A Comparison of Vivado HLS, SDSoC C++ and OpenCL for Porting a Matrix-vector-based Climate model mini-app to FPGAs

Document Version
Accepted author manuscript

Link to publication record in Manchester Research Explorer

Citation for published version (APA):
Alghamdi, M., Riley, G., & Ashworth, M. (Accepted/In press). A Comparison of Vivado HLS, SDSoC C++ and OpenCL for Porting a Matrix-vector-based Climate model mini-app to FPGAs. In PDPTA'21 - The 27th Int'l Conference on Parallel and Distributed Processing Techniques and Applications

Published in:
PDPTA'21 - The 27th Int'l Conference on Parallel and Distributed Processing Techniques and Applications

Citing this paper
Please note that where the full-text provided on Manchester Research Explorer is the Author Accepted Manuscript or Proof version this may differ from the final Published version. If citing, it is advised that you check and use the publisher's definitive version.

General rights
Copyright and moral rights for the publications made accessible in the Research Explorer are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Takedown policy
If you believe that this document breaches copyright please refer to the University of Manchester’s Takedown Procedures [http://man.ac.uk/04Y6Bo] or contact uml.scholarlycommunications@manchester.ac.uk providing relevant details, so we can investigate your claim.

Download date: 23. May. 2022
A Comparison of Vivado HLS, SDSoC C++ and OpenCL for Porting a Matrix-vector-based Climate model mini-app to FPGAs

Moteb Alghamdi\textsuperscript{1,2}, Graham Riley\textsuperscript{1}, and Mike Ashworth\textsuperscript{1}

\textsuperscript{1} Department of Computer Science, The University Of Manchester, UK. motel.alghamdi@manchester.ac.uk
\textsuperscript{2} College of Computer Science and Engineering, Taibah University, Saudi Arabia (mgamdi@taibahu.edu.sa)

Abstract. The High-Performance Computing (HPC) community’s interest in FPGAs as accelerators has been renewed due to the introduction of High-Level Synthesis tools (HLS). HLS tools hide the complexity of FPGA programming through raising the abstraction level for programmers. They offer environments where traditional HPC programmers can use high-level languages such as C/C++ and OpenCL to implement HPC application kernels on FPGAs. However the use of an HLS environment implies trade-offs between the achievable performance and programmer effort. This paper presents a comparative study between three HLS programming methodologies, Xilinx Vivado HLS and Xilinx SDSoC using both OpenCL and C++, all targeting the Xilinx Zynq UltraScale+ MP-SoC ZCU102. We use a matrix-vector kernel from the LFRic weather and climate model mini-app to compare the programming techniques, effort and resulting performance of an implementation using Vivado HLS with the higher level of abstraction provided by SDSoC C/C++ and OpenCL. We provide a comparative analysis of the design choices, scaling behaviour and peak performance. We find that Vivado HLS provides the highest performance due to the programmer’s ability to exploit low-level FPGA features in the manual construction of the hardware system design, but near equivalent solutions can be obtained with OpenCL and C++, with automatic design generation resulting in reduced programmer effort.

Keywords: HPC, HLS, FPGA, Xilinx SDSoC, Xilinx HLS, Vivado

1 Introduction

The introduction of High-Level Synthesis tools (HLS) has renewed the High-Performance Computing (HPC) community’s interest in Field Programmable Gate Arrays (FPGA) for accelerating HPC applications. HLS tools hide the complexity of FPGA programming through raising the abstraction level. They offer environments where traditional HPC programmers can use high-level languages such as C/C++ and OpenCL to implement kernels. The ability to program FPGAs at a high level of abstraction poses the following questions. What
is the trade-off between the performance and efficiency of the FPGA hardware solution and the ease-of-use of HLS tools for the traditional HPC programmer? And what are the techniques and understanding of FPGA hardware that are required by the programmer to achieve high performance?

This paper describes a comparative study between implementations produced using three programming methodologies, all targeting the same Xilinx FPGA hardware, to investigate the questions posed above. The first is an existing implementation, described in [1], exploiting multiple copies of a (restricted) C kernel processed using Vivado HLS with the resulting IP blocks integrated into a chip design manually using Vivado Design Studio; this will act as a reference implementation. The second and third methodologies produce, respectively, C++ and OpenCL kernels using Xilinx SDSoC, in which the FPGA chip design is generated automatically, thus representing a higher level of abstraction than Vivado HLS.

The case study is a single kernel from the LFRic weather and climate model described in section 2.1.

We are particularly interested in the trade-off between performance and programmer effort, which can be expected to be reduced by using high-level approaches. We address questions such as whether the use of high-level optimization techniques in C/C++ and OpenCL can match the benefits of manual optimizations carried out in a hand-tuned Vivado design. Comparisons of several aspects are reported based on both quantitative and qualitative metrics.

This paper makes the following contributions:

- The porting of an existing Vivado HLS version of the LFRic HPC mini-app to FPGAs using SDSoC C/C++ and OpenCL
- Comparison of three state-of-the-art approaches for utilizing FPGAs by traditional HPC programmers: Vivado HLS and SDSoC C++ and OpenCL
- A comparison and analysis of the essential techniques and differences in the three approaches which result in their performance and scalability

The paper is structured as follows: Section 2 presents the background and related work. Section 3 presents the reference Vivado design and the OpenCL and C++ designs, while Section 4 describes the study methodology. Section 5 presents performance results and an analysis of the three approaches. Section 6 discusses issues related to data movement in the three approaches and, finally, Section 7 draws conclusions and discusses future work.

2 Background

2.1 LFRic Weather and Climate Model

The LFRic model is a new atmospheric weather forecasting and climate simulation model that uses a cube-sphere grid to cover the globe see Figure 1. LFRic has been developed by the Met Office in the UK in partnership with universities and other research centres and builds upon the GungHo dynamical
core [3] with the aim of providing portable performance achieved using an innovative, architecture-independent domain-specific programming methodology implemented using PSyclone [2].

The kernel used in [1] is derived from a mini-app that computes pressure calculations and consists of several finite element double-precision matrix-vector multiplications on elements in columns of the atmosphere. Variants of this kernel are used extensively in the atmosphere dynamics computations, especially within the Helmholtz solver. For some example benchmark cases, the calculation can represent up to 50% of the model execution time on a CPU.

The test grid used in this study is a very coarse representation of the globe, in which the six faces consist of 12x12 finite-element cells, making 864 cells in the horizontal. Since columns of cells share edges, they cannot all be updated in parallel and a graph colouring scheme is used to identify parallelism [2]. Six colouring groups represent the 864 cells\(^3\). A single ‘colour’ has no dependencies and can be processed simultaneously. The mesh cells are distributed to four groups with 205 cells each plus a 32 cell group and a 12 cell group.

![Fig. 1. A cubed-sphere mesh as used in GungHo with 12×12 subdivisions per face, referred to as a C12 mesh. This gives 864 columns of cells. [2]](image)

### 2.2 LFRic matrix-vector multiplication Kernel (Matvec-Kernel)

Listing 1.1 shows the *restricted* C kernel of the matrix-vector multiplication extracted from the LFRic model and used in [1]. The kernel computes a set of 40 (NK) matrix-vector multiplications corresponding to the 40 finite element cells within a single vertical column. Each update consists of a *matrix* of size 8x6 and a 6-element right-hand-side vector, \(x\), producing a left-hand-side output vector of 8 elements, \(lhs\). Thus, there are \((8+6+48) \times 864 \times 40 \times 8B = 17\) MB of input data and \(8 \times 864 \times 40 \times 8B = 2\) MB of output data for the entire mesh. The size of the matrix is derived from the order of the finite element scheme used.

\(^3\) We note that subsequent work has reduced the number of colours required to four.
Listing 1.1. restricted C kernel for the Matrix-vector multiplication

```c
#define NDF1 8
#define NDF2 6
#define NK 40
#define MVTYPE double

int matvec_8x6x40_vanilla(
    MVTYPE matrix [NK][NDF2][NDF1],
    MVTYPE x [NDF2][NK],
    MVTYPE lhs [NDF1][NK]
) {
    int df, j, k;
    for ( k=0; k<NK; k++ ) {
        for ( df=0; df<NDF1; df++ ) {
            lhs [df][k]= 0.0;
            for ( j=0; j<NDF2; j++ ) {
                lhs [df][k]= lhs [df][k]+x [j][k] *
                    matrix [k][j][df];
            }
        }
    }
}
```

2.3 HLS Tools

FPGA acceleration is based on a host-kernel model in which host code running on the CPU launches compute-intensive kernels on the programmable logic of FPGA. The host code is responsible for controlling interactions with the accelerator, launching kernels, supplying data and bringing data back for further processing in the application.

Traditionally, implementing FPGA accelerated applications has been delivered through low-level hardware design languages such as VHDL [3], and Verilog [4]. These methods typically require extensive low-level hardware knowledge and significant development time and effort.

More recently, various academic groups and commercial vendors have developed High-Level Synthesis tools e.g. [5–7]. Examples include: Xilinx SDSoC [8], Xilinx Vivado HLS [9], Altera SDK [10], Maxeler [11] and OmpSS [12]. The target HLS tools for this paper are Xilinx Vivado HLS and Xilinx SDSoC.

**Vivado HLS** Vivado HLS is a compilation system which analyses a restricted form of C code. Compiler-automated optimizations are supplemented by programmer supplied HLS pragmas that invoke and guide a range of optimizations such as pipelining and unrolling and manage data placement and streaming. Vivado HLS writes Register Transfer Level (RTL) code which forms the basis
of an IP, *Intellectual Property*, Block that can be saved to an IP Repository for later inclusion in a full FPGA system design using Vivado Design Suite. This involves specifying a great deal of low level hardware details to configure the final generated system design, but also gives the application developer considerable fine control at this level.

**Software-Defined System-On-Chip (SDSoC)** SDSoC is a development environment that provides an Eclipse-based IDE for C/C++ and OpenCL application development. SDSoC combines the processing system, accelerators, data movers, signalling and drivers under one infrastructure. This abstraction enables shorter FPGA development time and simplifies the developer’s view of the interface between the software and hardware. The environment contains two FPGA high-level compilers: sdsc/sds++ for C/C++ kernels and xocc for OpenCL [8]. Much of the design generation is automated, but, as with Vivado HLS, the programmer is able to guide the compiler through compiler pragmas or attributes.

**OpenCL** The OpenCL programming methodology provides a programming language and runtime API [13] to support functional and portable software development. It also provides low-level hardware abstractions like memory hierarchy, platforms and execution models to facilitate the use of underlying hardware capabilities at a high-level of abstraction [13]. The OpenCL execution model defines two types of OpenCL kernels: task kernel and NDRange execution. An OpenCL kernel is executed on a device within the concept of index space. The most appropriate index space for FPGAs is task kernel execution [14]. The OpenCL memory model defines a hierarchy of device memories and their behaviour. The FPGA’s memory hierarchy is translated within the OpenCL environment as follows: Host-Kernel shared DDR memory resides outside the FPGA’s fabric area. OpenCL Global Memory can be represented either as shared off-chip memory or distributed memories (BRAM) in the FPGA’s fabric area. Local memory and private memory can be implemented using registers or BRAMs [13].

**C/C++** The SDSoC sdsc/sds++ compilers accept standard C/C++ application code and provide pragmas such as pipeline, dataflow and unrolling. Data movement in the C/C++ approach is specified through data motion networks [8] that manage the data movements. A data motion network has three components that programmers can control with pragmas for a better choice that suits the target kernel design. The most important component is the data mover engine which is an FPGA IP block for transferring data between the CPU and the FPGA accelerator. Two data mover engines are suitable for this study which are the Scatter and Gather (SG) and simple data memory engines (DMAs) [8].

### 2.4 Target FPGA Board

The target platform for this work is the Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit [15]. It contains a multiprocessor system-on-chip system with a
1.3 GHz ARM Cortex A53 quad-core CPU, and a Zynq UltraScale XCZU9EG-FFVB1156 FPGA. The board has 4GB DDR shared memory, consisting of four memory banks with four access ports. The ZU9 FPGA has programmable logic resources consisting of 548,160 Flip-Flops (FFs), 274,080 Look-Up-Tables (LUTs), 912 Block RAMs (BRAMs) and 2,520 DSP Slices (DSPs). We use Ubuntu 16.04.5 on the CPU and Xilinx SDK SDSoC 2018.2.

2.5 Related Work

Across almost all scientific areas in HPC applications, linear algebra operations are ubiquitous. FPGA implementations have focused mostly on matrix-matrix (MXM) and sparse matrix-vector (SMV) multiplication and simple iterative stencil-based solvers. For MXM, the authors in [16] have proposed a block design that enhances data locality and re-usability considering the local storage and I/O limitations in FPGAs. Another example is [17] which provides two FPGA accelerator designs that support IEEE 754 double-precision floating-point matrix multiplication on Virtex-5 FPGA. For SVM solution, the authors in [18] proposed a novel optimized sparse matrix-vector FPGA design that exposes parallelism across rows with low usage of on-chip memory. In contrast, the authors in [19] present a scalable kernel design that efficiently utilizes the available memory bandwidth and FPGA resources. Our work focuses on the trade-off between performance and programmability from the perspective of programmers coming to FPGAs from a traditional scientific background. To our knowledge, there is a little published work on this aspect of FPGA acceleration. For example, the authors in [7] conducted a survey and proposed an evaluation methodology for the use of HLS tools. Another example is from in [5] who present a comparison study between the AutoPilot HLS tool and an optimized hand-coded design for a sphere decoder kernel to show that the HLS solution can improve design productivity. A third work presented in [20] outlined a common underlying design philosophy between Xilinx SDAccel and Intel OpenCL HLS tools. The study aims to show how to overcome the differences in OpenCL HLS tools to enhance portability.

3 Design Descriptions

This study seeks to replicate in higher-level programming models an implementation of the Vivado HLS kernel and design described in [1]. The focus of that design is exploiting spatial parallelism on the FPGA using multiple matrix-vector IP blocks each with its own external BRAM. Also, as stated in [1], in any full port of the LFRic mini-app, the aim would be that the matrices are generated on the FPGA and the data kept in the FPGA plane and so will not need to be repeatedly transferred to or from the host. The host CPU code fills the BRAMs directly with data for a number of cells (matrix and x arrays) ahead of the execution of the kernels. When the kernels execute, the matrix and x array data are accessible to be read from the BRAM blocks to local BRAM inside the kernel.
blocks for the calculation process of each cell. A kernel processes a single cell at a time. Many options were explored for a genuine multicell block with Vivado HLS, but none performed as well as multiple single-cell blocks. The Vivado design is created based on three objectives. First: to stream the input and output data to the IP Block, targeting one 64-bit word per clock cycle. Second: to pipeline and overlap of the arithmetic operations targeting 64-bit multiplications and 64-bit addition operations every clock cycle. Third: to minimize FPGA resource usage. The objectives were carried out through several optimization decisions which can be classified into three groups: coding changes to the code in listing 1.1, computational optimizations and data movement optimizations\(^4\).

The code changes that were made are as follows: The loops in Listing 1.1 were reorganized to put the most extended loop, the \textit{k-loop} over levels, innermost. Data arrays were transposed so that the \textit{k-index} loop data are organized sequentially in the memory; this ensures that the innermost loop is sequential with a length of 40 elements. The kernel only computes the matrix-vector product. The update of the \textit{lhs} data array is computed on the host ARM CPU.

The computational optimizations that were applied are: \textit{unroll} the innermost loops and \textit{pipeline} the outer loops in the changed code. In addition, \textit{partition} the kernel local arrays to provide more memory access ports (and, hence, higher bandwidth). The best partitioning factors were found to be 60 for the \textit{matrix} and \textit{xl} arrays and 40 for \textit{lhs} array.

The data movement optimizations that were applied were: the \textit{matrix} and \textit{x} arrays data are first transferred from the DDR memory to the kernel blocks’ BRAM storage. Those data are communicated from the outside BRAM storage to local \textit{BRAM\_18K} logic elements in \textit{burst mode}. The \textit{x} data array is copied to the kernel only once to the local \textit{BRAM\_18K}, as it is constant for all the iterations of the outer \textit{df-loop}. Slices of the \textit{matrix} data are transferred to the local kernel array each \textit{df-loop} iteration, and output columns of the \textit{lhs} data array are copied out to the outside BRAM blocks at each iteration of the loop in \textit{burst mode}. Upon the end of the kernel execution, the \textit{lhs} array is copied from the outside BRAMs back to the DDR memory. The reason for this data movement design is for isolating the host(DDR memory)-to-kernel data load/store time from the kernel execution time [1].

In the OpenCL and the C++ designs, we attempted to replicate the objectives and optimizations of the Vivado HLS approach where possible. The code changes in the kernel code and the basic computational optimizations were replicated successfully, requiring similar effort in all approaches, though the syntax of the pragmas etc. required in each approach are different. However, for the data movement optimizations, we found that both C++ and OpenCL SDSoC provided no useful support for the equivalent of BRAM block creation outside of the kernel block [8] - see Section 6 for further details. Therefore, for the C++ and OpenCL approaches we implemented a solution where the kernels read the input data for the block of cells to be processed from the DDR memory into

\(^4\) Details of these optimizations are in [1].
local BRAM \textit{inside} the kernel itself, before performing the computations in a loop over the cells in the block\(^5\). The kernel in this case is a \textit{multicell} kernel.

4 Methodology

Our methodology is based on creating OpenCL and C/C++ FPGA solutions that seek to replicate the basic optimizations and design decisions applied in the coding and manual tuning of the Vivado HLS kernel described in Section 3. Performance is explored with different numbers of FPGA kernel IP Blocks, and with blocks processing different numbers of cells, each cell representing a vertical column of the atmosphere. We compared a number of scenarios, including trade-offs between performance scaling, the number of kernels that can be implemented on the FPGA, and the number of cells that can be accommodated in the BRAMs associated with each block.

We use the following metrics: \textbf{Performance}, using \textit{flops per second} for arithmetic computation and \textit{bytes per second} for data movement, and the overall \textit{runtime (seconds)} for the version; \textbf{Resource usage}: the percentage of FPGA resources consumed; \textbf{Hardware design}: how close to the Vivado design can the OpenCL/C++ approaches get, and why; \textbf{Data movement options}: what are the available options for exploiting the memory hierarchy, and how does the choice affect performance; \textbf{Development effort}: a qualitative review of the steps and effort involved in developing the OpenCL/C++ versions; \textbf{Level of hardware expertise required}: how much hardware knowledge the programmer needs.

5 Performance Analysis

In order to make a fair like-for-like comparison, we replicated, as far as possible, the same design and coding decisions in the OpenCL and C++ versions as were used in the Vivado version. In this section we present the results for each version and discuss the differences in the designs arising from the different approaches.

\textbf{Vivado Design} Figure 2 shows the performance of the FPGA-based computation phase of the Vivado design for a range of number of blocks and cells-per-block. As discussed in Section 3 the Vivado design has the advantage of providing BRAM memory that is external to the kernel IP blocks. The data is directly transferred from the host CPU DDR memory to BRAM associated with each block. BRAM access is much faster than access to DDR. The Xilinx Ultrascale+ board supports up to a 600 MHz clock and a single matrix-vector IP block can run at 435 MHz, but when integrating 12 blocks in a multi-block design, the maximum clock speed is reduced to 310 MHz to meet timing constraints. All data in Figure 2 is at 310 MHz.

\(^5\) This was found to be more efficient than reading data from DDR on a cell-by-cell basis.
The best performing Vivado implementation has 12 blocks and 13 cells-per-block and delivers 4.98 Gflop/s, see Figure 2 and Table 2. The scaling with number of blocks is good, at twelve blocks the parallel efficiency is 88%. Increasing the number of cells per block initially delivers improved performance, as additional cells hide some of the latency costs, but quickly saturates. The matrix-vector kernel executes 2 flops per cycle (one dmul and one dadd). At 310 Mhz and for 12 blocks, this leads to a theoretical peak performance figure of 7.44 Gflop/s. Thus, the Vivado HLS design achieves 67% of peak.

**OpenCL Design** Figure 3 shows the OpenCL performance for three IP blocks and a range of cells-per-block. Unlike the Vivado case, the data in the OpenCL implementation is located in DDR RAM and kernel loads data into local BRAM before starting the computation. The kernel processes multiple cells in a call, whereas Vivado processes only a single cell in each call. See the data movement discussion in Section 6 for more details.

The highest clock rate we were able to use in the OpenCL design is 200 MHz, above which the hardware design generated by SDSoc did not meet timing constraints. The highest performance achieved is with 1 block with 120 cells, at 2.02 Gflop/s. OpenCL implementations with different numbers of blocks scale quite well with increasing cell numbers, but the more blocks created, the fewer cells each block can have due to BRAM resource limitations. The resources used for the best performing implementations are shown in Table 1. The highest number of OpenCL blocks was only three, regardless of the number of cells, due to either lack of resources or failure to meet timing constraints. This is believed to be due to additional complexity in the auto-generated design due to the number
Fig. 3. Performance of the OpenCL Matrix-vector kernel designs at 200Mhz as the number of blocks and cells-per-block varies.

Table 1. Resource usage of the best performing implementations for the matrix-vector block and for the total design.

<table>
<thead>
<tr>
<th>Resource Usage</th>
<th>Design</th>
<th>FF</th>
<th>LUT</th>
<th>DSP</th>
<th>BRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usage Per IP Block</td>
<td>Vivado 12-Blocks 13-Cells</td>
<td>278388 (50.78%)</td>
<td>81396 (29.69%)</td>
<td>120 (5%)</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td>OpenCL 1-Block 120-Cells</td>
<td>74845 (13%)</td>
<td>31174 (11%)</td>
<td>224 (8%)</td>
<td>588</td>
</tr>
<tr>
<td></td>
<td>C++ 1-Block 180-Cells</td>
<td>22895 (4%)</td>
<td>27563 (10%)</td>
<td>28 (1%)</td>
<td>781.5</td>
</tr>
<tr>
<td>Total Design</td>
<td>Vivado 12-Blocks 13-Cells</td>
<td>304118 (55%)</td>
<td>178574 (65%)</td>
<td>120 (5%)</td>
<td>816</td>
</tr>
<tr>
<td></td>
<td>OpenCL 1-Block 120-Cells</td>
<td>94101 (17.16%)</td>
<td>48880 (17.83%)</td>
<td>224 (8%)</td>
<td>622.5</td>
</tr>
<tr>
<td></td>
<td>C++ 1-Block 180-Cells</td>
<td>49987 (9%)</td>
<td>37100 (14%)</td>
<td>28 (1%)</td>
<td>827</td>
</tr>
</tbody>
</table>

of DDR memory paths required. This problem is avoided in the Vivado design, in which each block has a dedicated path to BRAM.

The best Vivado design has 12 blocks with 13 cells in each block. The 12 blocks are executed concurrently, leading to the processing of 156 cells per 12-kernel execution. The best OpenCL design has 1 block with 120 cells, a slightly lower number of cells.

Analysis of the system design reveals that in the compute intensive phase, there are a maximum of 32 flops per cycle (16 dmuls and 16 dadds). At 200 MHz this leads to a theoretical peak performance figure of 6.4 Gflop/s (compared with 7.44 Gflop/s for Vivado HLS). Thus, the OpenCL design achieves 31.5% of peak performance (Vivado achieves 67%).
**Fig. 4.** Performance of the C++ Matrix-vector kernel designs at 150Mhz, as the number of blocks and cells-per-block varies.

**C++ Design** Figure 4 shows the performance of the C++ implementations as the number of blocks and cells varies. The highest performance with C++ is 1.78 Gflop/s with one block and 160 cells (see Table 2). As with the OpenCL design, the C++ performance figures include the data transfer costs from DDR to local BRAM. Another issue affecting the C++ performance is the choice of the Scatter-Gather (SG) DMA engine. SG is the slower DMA option, but it can handle the larger volumes of data required for execution with multiple cells. This issue is discussed further in the data movement subsection in Section 6. For the C++ designs, the performance scales reasonably well with the increase in cell numbers, but the more blocks used, the fewer cells can fit in the FPGA due to resource limitations. For small numbers of cells, up to four blocks can be used. With more blocks, the highest clock rate we were able to use in the C++ design is 150 MHz. Both the clock rate and the maximum number of blocks are limited by timing constraints in the auto-generated design. Figure 4 shows poor scaling with number of blocks. This is as a result of the bandwidth of the DDR memory ports being shared between the blocks in the C++ design.

**Time-based comparison** Here we discuss the performance in the context of each phase of the overall algorithm: preparing and loading data onto the FPGA, Load, computing, Comp, and returning data back to the host Store. This enables us to put the performance rates into the broader context of the whole application and provides further insight into the different data movement and cells-per-block strategies that were employed in the three approaches. Note that the Load and Store operations are not equivalent between the three approaches due to the different data movement methods used, as discussed in Section 6.

Table 2 shows the execution times for the best implementations and for two special cases (marked with an asterisk in the table) where the overall time is
Table 2. Performance data for the best implementations. (*: performance results for a smaller, 1 block/26 cell versions)

<table>
<thead>
<tr>
<th>Design</th>
<th>Load Time(s)</th>
<th>Comp Time(s)</th>
<th>Store Time(s)</th>
<th>Overall Time(s)</th>
<th>Performance (Gflop/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vivado</td>
<td>0.0204</td>
<td>0.0006</td>
<td>0.0137</td>
<td>0.0349</td>
<td>4.98</td>
</tr>
<tr>
<td>12-Blocks 13-Cells</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenCL</td>
<td>0.0231</td>
<td>0.0049</td>
<td>0.0188</td>
<td>0.0469</td>
<td>2.02</td>
</tr>
<tr>
<td>1-Block 120-Cells</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*OpenCL</td>
<td>0.0187</td>
<td>0.0049</td>
<td>0.0195</td>
<td>0.0381</td>
<td>0.67</td>
</tr>
<tr>
<td>1-Block 26-Cells</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++</td>
<td>0.0223</td>
<td>0.0019</td>
<td>0.0033</td>
<td>0.0274</td>
<td>1.78</td>
</tr>
<tr>
<td>1-Block 180-Cells</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*C++</td>
<td>0.0125</td>
<td>0.0025</td>
<td>0.0032</td>
<td>0.0183</td>
<td>1.31</td>
</tr>
<tr>
<td>1-Block 26-Cells</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

better. For the OpenCL and C++ designs we found that the highest Comp performance does not necessarily lead to the fastest overall time, e.g. execution of 1 block with 26 cells delivered a better overall time compared to that of the best performing versions. The 26-cell case delivers faster overall time because the Load and Store times are faster, while we note that the kernel is called more times than in the 120-cell OpenCL version and 180-cell C++ version, because it deals with fewer cells per call. This effect is not seen with the Vivado design.

With C++, the Load and Store times are fastest with 1 block and 26 cells. Data for 26 cells at a time is prepared by the host and written to shared DDR memory. Load time in the three best implementations is fairly similar, but the Store time is faster in the C++ implementations. Vivado HLS Load and Store times are faster than OpenCL. For Vivado, data for 13 cells for each of 12 blocks is prepared on the host in DDR memory and then written directly to the BRAM associated with each block. In contrast, for OpenCL, data for 120 cells at a time is prepared on the host for the single block, and the OpenCL buffer write call executed (enqueueWrite()) [21].

Table 3. Rates of data movement for the best implementations

<table>
<thead>
<tr>
<th>Design</th>
<th>Load (MB/s)</th>
<th>Store (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vivado</td>
<td>1085.8</td>
<td>108.2</td>
</tr>
<tr>
<td>12-Blocks 13-Cells</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenCL</td>
<td>643.6</td>
<td>116.6</td>
</tr>
<tr>
<td>1-Block 120-Cells</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++</td>
<td>666.9</td>
<td>676.7</td>
</tr>
<tr>
<td>1-Block 180-Cells</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*C++</td>
<td>1189.6</td>
<td>673.9</td>
</tr>
<tr>
<td>1-Block 26-Cells</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Vivado Comp time is faster than that of both OpenCL and C++, for which the compute kernels are reading data from DDR and compute multiple cells per call, whereas in the Vivado design the compute kernels operate with data from BRAM and process one cell per call. The OpenCL whole application performance is competitive with the Vivado design, while the C++ whole application performance is faster in both the best implementation case and the 1 Block and 26 cell case; mainly as a result of the faster Store times.

In summary we find that the Vivado design achieved the best performance for the compute phase of 4.98 Gflop/s, while the C++ design achieved the best overall time. Although the OpenCL and C++ designs operate at a lower clock rate than the Vivado design, and the data transfer time is from DDR rather than BRAM, the extra parallelism they exploit by processing multiple cells per call means that the OpenCL design provides an competitive overall runtime, with a compute performance of 2.02 Gflop/s, and the C++ design delivered the best overall time though with only 1.78 GFlop/s.

6 Discussion

Data Movement

The Vivado design isolates data transfer from kernel execution by providing external BRAM blocks for each kernel block. Data transfer occurs in a separate step to the kernel call. The input data is read by the kernel from external BRAM into local BRAM, one cell at a time. The performance in Figure 2 excludes the data transfer time. Table 3 shows that the Vivado load rate (1085.8 MB/s) is higher than the load rates in the best OpenCL and C++ implementations, due to the manually configured DDR-to-BRAM connection.

The load rates for the best OpenCL (120 cells) and C++ (180 cells) implementations in Table 3 are slow compared to the Vivado load rate but, as discussed in Section 5, these rates reflect only the preparation of cell data and the writing of the data to DDR, rather than to the BRAMs in the Vivado design. However, for the C++ 1 Block/26 cells (which delivered the best overall time) load and store rates are better than those of the Vivado and the OpenCL rates since the DDR buffers are shared between the host and device.

In OpenCL, local BRAMs are populated at the start of the kernel execution and this data is read during execution. Thus, the data transfer overhead is counted in the OpenCL performance. We explored a number of different methods in an attempt to isolate the OpenCL kernel execution time from this data movement time. For example, we created three kernels: a load kernel, a comp kernel and a store kernel and used OpenCL pipes [22], which act as local memory attached to the comp kernel, to connect them. The Load and store kernels are responsible only for transferring the data to/from the pipes before and after kernel execution. With this design, we were able to isolate the data movement time from the kernel execution time. However, the compute time did not improve; performance is limited to the data rate that the pipes can provide. Attempts to increase the data width of the pipes were inconclusive and we leave a thorough investigation to future work.
Another solution we explored was the creation *On-chip global memory* out of BRAMs. This would enable precisely the same solution as in the Vivado design. However, Xilinx stopped supporting this feature in 2017.

Similarly to OpenCL, in C++ the local BRAMs are populated at the start of the kernel and then read from the local BRAM during the kernel execution. This has added overhead to the C++ performance in the same way as in OpenCL. In addition, C++ design performance has been impacted by the DMA engines choices, as described in Section 5.

**Development Effort** This study shows that the Vivado approach is rather programmer unfriendly as it required the highest amount of development effort and hardware design experience. This is because the approach requires the programmer to take care of low-level concerns to produce an FPGA solution, including the manual creation of the FPGA hardware system design which requires the explicit configuration of the data-movement and connections between the IP blocks, address manipulation, setting widths of data paths and managing the execution of the kernel IP blocks through setting and examining the start and stop bits of the kernel. In addition, the programmer has to design and implement the managing of the preparation and transfer of data to the FPGA BRAM blocks explicitly in the host code. This all requires significant effort. The writing and optimization of kernel code requires a similar level of effort to other methods however, only the syntax of the pragmas required for pipelining and unrolling, etc. change.

In contrast, the OpenCL and C++ approaches hide most of the complexity of the system design steps required by the Vivado approach and assign it to the SDSoC compilers. In these two approaches, the programmer needs only to know the SDSoC design flow and have a good understanding of the kernel code pragmas for pipelining and unrolling, etc., and knowledge of their effect on the performance. The C++ approach requires also an understanding of the data-movement engines available in the system and their advantages. However, the resulting automatic designs are not as efficient as can be achieved manually. This is presumably because of the generality of the design solution, but this generality can lead to issues with timing in designs, limiting the maximum clock rate.

**FPGA vs. CPU Comparison** The authors in [1] provided performance figures for the matrix-vector kernel implementation on a state-of-the-art Intel Broadwell E5-2650 v2 2.60 GHz CPU with eight cores. The CPU’s eight cores are exploited by using OpenMP. Table 4 shows a performance comparison between the CPU implementation and the three best implementations we report in this paper. The CPU can deliver a peak performance of 332.8 Gflop/s as it can process 16 flops/cycle multiplied by eight cores with 2.6 GHz frequency. The ZCU102 FPGA peak Performance is 600 Gflop/s as stated in [23]. In [1] it was reported that FPGA performance for the double-precision matrix-vector kernel is 5.34 Gflop/s which is 54% of that achieved on an 8-core Intel Broadwell CPU.

---

6 The peak performance computation does not take into account the FPGA data precision; probably, it is an overestimate for 64-bit precision.
The CPU implementation outperforms the FPGA implementations. The ideal would be for the FPGA “accelerator” to outperform the CPU. However, consideration of the comparison between power consumption and price between these two devices is critical in an overall comparison with CPUs and other accelerators, such as GPUs. CPUs are much more power hungry than FPGAs. The authors in [24] report that GPU power efficiency is up to 20 Gflop/s/W, with price efficiency varying from 0.07 to 0.12 €/Gflop/s. GPUs are used as accelerators in multi-CPU systems as their efficiencies exceed those of CPUs. The authors state that a mid-class FPGA’s power efficiency exceeds 70 Gflop/s/W, with a price efficiency of 0.29 €/Gflop/s.

Table 4. Comparison of ZU9 FPGA double-precision Vivado, OpenCL and C++ matrix-vector performance implementations with Intel multicore CPU performance

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Performance (Gflop/s)</th>
<th>Peak performance (Gflop/s)</th>
<th>Percentage peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Broadwell E5-2650v2 2.60 GHz 8-core CPU</td>
<td>9.86</td>
<td>332.8</td>
<td>3.0%</td>
</tr>
<tr>
<td>ZCU102 FPGA (Vivado implementation)</td>
<td>4.98</td>
<td>600</td>
<td>0.83</td>
</tr>
<tr>
<td>ZCU102 FPGA (OpenCL implementation)</td>
<td>2.02</td>
<td>600</td>
<td>0.33</td>
</tr>
<tr>
<td>ZCU102 FPGA (C++ implementation)</td>
<td>1.78</td>
<td>600</td>
<td>0.29</td>
</tr>
</tbody>
</table>

7 Conclusions

The performance gap between the three approaches is seen to be relatively small. While Vivado HLS provides a compute phase performance of 4.98 GFlop/s, OpenCL 2.02 GFlop/s and C++ 1.78 GFlop/s, the best overall execution times for one execution over the full mesh of 864 elements in this (small) version of the mini-app was 0.0183s (181.29 Mflop/s) for the C++ with 1 Block/26 cells, whereas the overall time for the Vivado HLS design with 12 Blocks/13 cells was 0.0349s (95.06 Mflop/s) and the best OpenCL overall time, with 26 cells, was 0.0381s (87.08 MFlop/s). The performance for the Vivado design benefited from not including the data transfer overhead. However, when we include the data transfer time, the C++ overall time was better than the Vivado design, and the OpenCL overall time is competitive.

The Vivado HLS design takes advantage of kernel-related BRAM, rather than DDR memory, to store input (and output) data prior to execution of the compute-intensive phase of the kernels; a clear performance advantage. This facility is not readily available in the SDSoC C++ and OpenCL approaches, and the consequent data access from DDR has a performance impact. In addition, the
absence of this facility has limited the scalability of multi-block performance in
OpenCL and C++ due to the sharing of DDR memory bandwidth between the
blocks. However, the processing of multiple cells in a call allows the exploitation
of some extra parallelism, providing a performance benefit over the Vivado HLS
design which processes a single cell per call.

The abstraction level in the OpenCL and the C++ approaches is higher than
in Vivado HLS and leads to higher programmer productivity but provides less
control of the system and low-level design than Vivado HLS, which can lead to
extracting better performance from the FPGA resources.

Acknowledgements

Moteb Alghamdi is sponsored by Taibah University, Madinah, Saudi Arabia. Work on the Vivado design was supported by the EuroExa project (grant agree-
ment no. 754337), funded by the European Union’s Horizon 2020 Research and
Innovation Programme. Work on the OpenCL and C++ designs was supported
by the ESIWACE2 project (grant agreement no. 823988), funded by the Euro-
pean Union’s Horizon 2020 Research Infrastructures Programme.

References

1. M. Ashworth, G. D. Riley, A. Attwood, and J. Mawer, “First steps in porting the
lfric weather and climate model to the fpgas of the euroexa architecture,” Scientific
2. S. V. Adams, R. W. Ford, M. Hambley, J. Hobson, I. Kavčič, C. M. Maynard,
T. Melvin, E. H. Müller, S. Mullerworth, A. Porter et al., “Lfric: Meeting the chal-
genews of scalability and performance portability in weather and climate models,”
5. J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, “High-
level synthesis for fpgas: From prototyping to deployment,” IEEE Transactions
6. D. Koch, F. Hannig, and D. Ziener, FPGAs for software programmers. Springer,
2016.
7. R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao,
S. Brown, F. Ferrandi et al., “A survey and evaluation of fpga high-level synthesis
tools,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and
documentation/sw_manuals/xilinx2018_2/ug1278-sdsc-programmers-guide.pdf,
2021, [Online; accessed 12-January-2021].
documentation/sw_manuals/xilinx2018_1/ug1270-vivado-hls-opt-methodology-guide.pdf,
2021, [Online; accessed 12-January-2021].