Concurrency Mapping to FPGAs with OpenCL: A Case Study with a Shallow Water Kernel

Concurrency Mapping to FPGAs with OpenCL: A Case Study with a Shallow Water Kernel

Moteb Alghamdi[1,2], Graham Riley[1], and Mike Ashworth[1]

[1] Department of Computer Science, The University Of Manchester, UK.
moteb.alghamdi@manchester.ac.uk
[2] College of Computer Science and Engineering, Taibah University, Saudi Arabia (mgamdi@taibahu.edu.sa)

Abstract. FPGAs have been around for over 30 years and are a viable accelerator for compute-intensive workloads on HPC systems. The adoption of FPGAs for scientific applications has been stimulated recently by the emergence of better programming environments such as High-Level Synthesis (HLS) and OpenCL available through the Xilinx SDSoC design tool. The mapping of the multi-level concurrency available within applications onto HPC systems with FPGAs is a challenge. OpenCL and HLS provide different mechanisms for exploiting concurrency within a node leading to a concurrency mapping design problem. In addition to considering the performance of different mappings, there are also questions of resource usage, programmability (development effort), ease-of-use and robustness. This paper examines the concurrency levels available in a case study kernel from a shallow water model and explores the programming options available in OpenCL and HLS. We conclude that the use of SDSoC Dataflow over functions mechanism, targeting functional parallelism in the kernel, provides the best performance in terms of both Latency and execution time, with a speedup of 314x over the naive reference implementation.

Keywords: FPGAs, OpenCL, HPC, HLS, Xilinx SDSoC

1 Introduction

Due to their promising performance and power efficiency, FPGAs have been explored for their potential to accelerate traditional HPC applications, including fluid computations [1–4]. The low-level nature of FPGAs has meant significant programmer effort is needed to achieve levels of performance comparable with traditional architectures. With the development of High-Level Synthesis (HLS) tools [5], such as Xilinx Vivado_HLS [6], Intel Altera SDK [7], Xilinx SDSoC [8] and Maxeler [9], the HPC community's interest in FPGAs as an accelerator has increased [10] [11]. These tools enable the generation of FPGA hardware configurations from high-level descriptions, such as C/C++, OpenCL and Java. The use of FPGAs is also emerging in supercomputers as an alternative accelerator to GPUs to improve performance and power efficiency on the path to Exascale computing as in the EuroEXA project [12].

OpenCL is a portable programming language [13] being explored by the HPC community for the acceleration of applications on FPGAs [14–16]. However, although OpenCL compilers for FPGAs generate functionally correct hardware designs, achieving high performance remains a challenge. HPC software developers already face a complex design problem when choosing how best to map the concurrency in their applications to the heterogeneous hardware resources in traditional HPC systems. Concurrent designs are typically mapped into high-level language extensions such as MPI to exploit parallelism over nodes, OpenMP to exploit shared memory parallelism within a node and OpenACC, OpenCL or CUDA to exploit GPUs. In each of these cases, the target hardware is essentially fixed. With the introduction of FPGAs, HPC developers can control the design of the actual hardware itself. This makes the concurrency mapping problem much more complex since the choices now include language options which control the *design* of the FPGA hardware.

In this paper, we explore the concurrency mapping problem using a simple shallow water model [17]. Ultimately, the target is a EuroEXA-like HPC system that consists of multiple nodes comprising CPUs with FPGA accelerators. The shallow water application (*shallow*) consists of several distinct *kernels*. This study focuses on the exploration of mapping the concurrency levels within a single representative kernel from shallow to a single FPGA node consisting of a multi-core CPU and an FPGA device, the Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit [18]. We explore the concurrency mapping options available using OpenCL in Xilinx SDSoC and characterise their use in terms of performance and FPGA resource usage. We also address questions related to development effort with the aim of providing insights to the traditional HPC software developer which can guide their future design choices.

This paper makes the following contributions:

1. An exploratory study of the use of HLS mechanisms with OpenCL and SDSoC for mapping instruction-level, data- and functional-parallelism in *shallow* to an Ultrascale+ FPGA.
2. An analysis of the performance, resource usage and programmability with the aim of supporting traditional HPC programmers to make good choices when targeting FPGAs.

The paper is structured as follows: Section 2 presents the background, while Section 3 details the related work. Section 4 discusses the concurrency mapping mechanisms available in OpenCL and SDSoC for host and FPGA device programming. Section 5 presents an overview of the software designs, while Section 6 contains the experimental results and discussion. Finally, Section 7 presents conclusions and future work.

2 Background

The Field Programmable Gate Array (FPGA) is a programmable device. Unlike other processing devices, such as GPUs and CPUs, which have pre-defined architectures, FPGAs enable the logic architecture to be fully customizable whereby

the programmer can tailor the hardware solution to the application needs [16]. Describing the required logic behaviour and implementing it onto an FPGA involves two high-level process steps. It starts with the description of the concurrent algorithm from a high-level language, in this case the OpenCL kernel, which is transformed by the HLS tool chain to a low-level hardware representation known as an Intellectual Property or IP block [16]. The IP block contains the functional logic of the kernel code, and is called from the OpenCL host code.

OpenCL supports the development of functional and portable software by providing a programming language and a runtime API [13]. Low-level hardware abstractions are provided, such as platform, memory and execution models for exposing the underlying hardware capabilities in the high-level language. The OpenCL execution model defines the execution of kernels. There are two types of kernel in OpenCL: the *task* kernel and the *NDRange* kernel. The task kernel is executed as a single *work item*, while the NDRange kernel is executed within the concept of an *index space*, in the simplest case the index range of a `for` loop, where work items (loop iterations) may be grouped into *work groups*.

The OpenCL memory model defines the memory hierarchy that OpenCL applications can use and is translated for FPGAs as follows. *Host memory* (e.g. DDR) that resides outside the FPGA's fabric area. The allocation of memory buffers and data transfer is the responsibility of the host using OpenCL APIs. *Global Memory* can be represented either as shared off-chip memory or using distributed memories (e.g. BRAM) that reside within the FPGAs fabric area. *Local Memory* is on-chip device memory that is accessible only within one compute unit. This memory is implemented using registers or BRAMs. Further details are provided in Section 4.

The Shallow Water model (SWM) implements a set of partial differential equations to describe fluid flows, including oceanic and atmospheric flows in Numerical Weather Prediction (NWP) and Climate Modelling (CM). The SWM used here (*shallow*) is based on [17]. The SWM computes wind velocities (U, V) in the x and y directions, and potential pressure (P). It also calculates mass fluxes, (CU and CV), potential vorticity (Z), and the fluid potential surface height (H). A two-dimensional finite difference model is used to solve the shallow water equations over time and space. This application contains multiple levels of concurrency. In this paper we focus on a single kernel as an example to illustrate the options for mapping the available concurrency. The chosen kernel is called `L100`, which is shown in Figure 1. In this paper, we use a domain of size 64x64. A single iteration of the inner loop of the `L100` kernel contains 24 32-bit `float` operations. Thus, there are 98,304 `float` ops for the 64x64 domain for a single cycle. The kernel requires 23 `read` accesses, and 4 `write` accesses to the global DDR memory in each iteration, leading to a total of 368KiB input data and 64KiB output data for a single time-step.

Concurrency and Mapping Options: The `L100` kernel takes three arrays as input (U, V, P) and they are used to calculate four arrays (CU, CV, Z, H) in separate operations. There are several ways to express this algorithm in a high-level programming language such as C++ and this initial expression of the

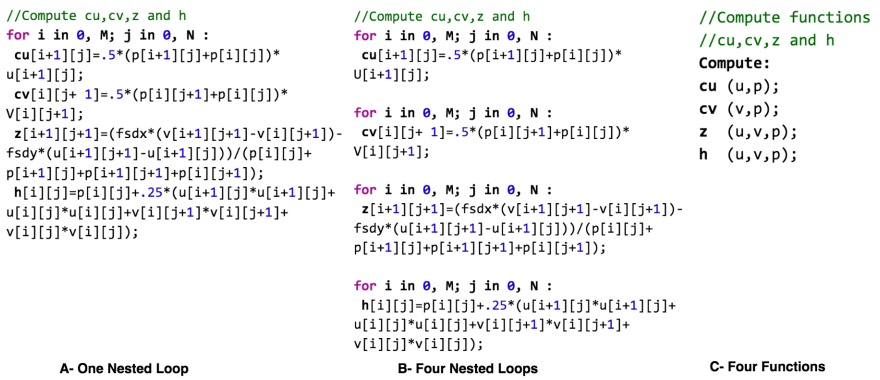

Fig. 1. Pseudocode for the *L100* kernel coding options. A- kernel operations in one *for* loop. B- each operation in a loop. C- each operation in a *function*

computations in code constrains the choices to exploit concurrency. Here, we consider three candidate coding options as represented in Figure 1. The calculations can be wrapped with one nested `for` loop as in Figure 1(a); the assignment to each variable (CU, CV, Z, H) can be computed in a separate loop (Figure 1 (b)); implement each of the assignments as a separate function (Figure 1 (c)). As is apparent in Figure 1, each assignment, and every computation of an element of each of CU, CV, Z and H, are all independent. That means that, theoretically, with an ideal machine design, all the operations could be executed in one step; there is an high degree of concurrency in this *pleasingly parallel* algorithm. There are data dependencies between the kernels in shallow but we use the L100 kernel to focus on the exploration of the concurrency mapping options. Handling such data dependencies is reasonably straightforward and leads to pipelining of the kernels. Three distinct types of concurrency may be identified in this algorithm: *Instruction-Level Parallelism* (ILP) which involves exploiting parallelism from the computation of each instance of the loop iterations (i.e. each *statement*) since each consists of several floating point operations. This parallelism is clear in the code options (a) and (b) in Figure 1. *Functional Parallelism* (FP) where each function in code option (c) in Figure 1 can be processed in parallel. *Data Parallelism* (DP) where the processing of the iterations in each loop in the algorithm can be carried out in parallel. In OpenCL terminology, each such iteration (or group of operations) may be considered as a work-item (WI).

FPGAs are a platform where a hardware solution can be tailored to fit the algorithm requirements. The question for the HPC scientific programmer is: how best to map the different concurrency types available in the `L100` algorithm to exploit the FPGA's potential using the mechanisms available in the OpenCL language and in the Xilinx SDSoC HLS tool? These mechanisms are discussed in detail in Section 4.

3 Related Work

Since the release of specialist compilers such as the Xilinx SDSoC development environment [8] and the Intel FPGA SDK for OpenCL [7], several research studies have been published. In [19] OpenCL optimizations methods for implementing stencil computations on FPGAs are explored. Their implementations of 1D and 2D convolution and Jacobi iteration kernels achieved roughly two orders of magnitude performance increase over the baseline kernels. In [20] the authors evaluated the use of the Altera OpenCL-to-FPGA compiler (now the Intel HLS Compiler) to compare FPGA performance with CPUs and GPUs for accelerating document filtering algorithm kernels. They showed that the FPGA performance per watt outperformed the CPUs and GPUs by five times. In [21], the authors propose a performance analysis framework that can shed light on performance bottlenecks and guide code tuning for OpenCL applications on FPGAs. In [22] OpenCL performance on FPGAs in terms of execution time and power consumption was compared with high-end GPUs. The evaluation was based on the implementation of several algorithms, and the authors concluded that FPGAs could sometimes provide more speedup than GPUs with a particular OpenCL programming style and efficient utilization of HLS directives. In [23] optimization techniques for OpenCL kernels on FPGAs under device-specific hardware constraints are explored. The techniques were evaluated on the OpenDwarfs benchmark suite in terms of performance improvement and resource utilization. They discussed techniques such as single work-item kernel versus NDRange kernel, manual and compiler vectorization, intra-kernel channels, and algorithmic refactoring. In contrast, this paper compares different methods within the OpenCL programming language and different mechanism available in the SDSoC compiler for mapping the kernel's concurrency to FPGAs.

4 Concurrency Mapping Mechanisms

Using OpenCL, there are two paths available for mapping the concurrency in a kernel to an FPGA. The first is through the use of the high-level OpenCL programming constructs in the host code, such as the type of the OpenCL kernel and the type of Command Queue (CQ, in-order or out-of-order). This approach often results in there being several, relatively simple, kernels that implement the algorithmic functions required by the application. The other, complimentary, way is to exploit concurrency using SDSoC-Specific OpenCL features in the kernels using HLS *attributes* to direct the FPGA compiler to generate the hardware solution. There are a number of key low-level optimisation strategies available from the SDSoC environment that are critical in achieving high performance from the kernels.

There are three critical mechanisms in OpenCL to control the mapping of the concurrency to the FPGA. The first is the kernel *type*; a kernel may be a *task* kernel or an *NDRange* kernel [16]. The second is the number of kernels in the design, reflecting the developer's choice as to how to implement the algorithmic operations of the application in terms of kernels. The execution of multiple

kernels can be controlled through OpenCL command queue(s), *in-order* or *out-of-order* [13] [24], and multiple queues may be used. OpenCL has (host) mechanisms to synchronise the execution of kernels: *barriers* and *events*. Use of a *Task kernel* allows the execution of a kernel with one work-group (WG) that contains only one work-item (WI). An *NDRange Kernel* exploits data parallelism using multiple work-items (WIs). NDRange organizes the WIs into WGs. WGs can be executed simultaneously [16].

The mapping mechanisms that are available in SDSoC for mapping concurrency are *Dataflow*, the use of *Multiple Compute Units*, *Loop pipelining* and *Loop unrolling*. The *Dataflow* attribute (`__attribute __((xcl_dataflow))`) allows the parallel execution of a kernel's functions or loops, decreasing the latency. The use of *multiple Compute Units (CUs)* increases the level of parallelism by utilising more FPGA resource to compute a kernel's operations. With NDRange kernels, multiple CUs can be created to execute the WGs [24]. The *pipeline* attribute `(__attribute__ ((xcl_pipeline_loop)))` can improve the latency of a kernel by overlapping loop iterations. The most important parameter with the pipeline attribute is the *initiation interval* (II), the number of clock cycles before the next iteration can start [24]. The *loop unrolling* attribute`(__attribute__-((opencl_unroll _hint(n))))` maps instruction-level parallelism. Unrolling increases the iterations in a loop body, reducing the loop trip count [24].

SDSoC tools provide low-level optimisations that can improve efficiency, including utilisation of the FPGA's memory hierarchy to improve data movement between the host and FPGA kernels:

Burst Mode: burst mode transfers large volumes of data from the host memory (DDR) to the FPGAs kernel local memories (e.g. BRAMs).

Max Memory Ports: The Zynq UltraScale+ has four DDR memory ports. This option increases the number of ports that the CUs can use [24].

Array Partitioning: BRAMs have a limited number of data ports. The *array partitioning* attribute, `(__attribute__((xcl_array_partition())))`, divides a data array over multiple physical BRAMs, increasing the numnber of ports. There are three types of array partitioning: block, cyclic and complete [24].

5 Experimental Design Overview

The targeted FPGA is the Xilinx Zynq UltraScale+ MPSoC ZCU102 evaluation platform [18] which contains a multiprocessor system-on-chip system, including a 1.3 GHz ARM Cortex A53 quad-core CPU and a XCZU9EG-FFVB1156 FPGA. The board 4GB DDR memory in memory banks (4 access ports). This memory is shareable between the CPU and FPGA. There is 3.5MB of fast BRAM memory. The FPGA has 548,160 Flip-Flops (FFs), 274,080 Look-Up-Tables (LUTs), 912 Block RAMs (BRAMs) and 2,520 DSP Slices (DSPs). The ARM CPU runs Ubuntu 16.04.5, and we used Xilinx OpenCL SDK SDSoC 2018.2.

Concurrency mapping mechanisms are used to optimise the computation of operations. However, to produce an efficient kernel the optimisations in subsection 4 for improving data-movement are also required. Thus, we apply the

following optimisations, listed in order of programming complexity, in each experiment.

1. Enable the max memory ports, to reduce DDR latency.
2. Transfer data from the DDR memory to the FPGA BRAM to take advantage of their lower access latency.
3. Transfer the data to the BRAMs in burst-mode.
4. Use the array partitioning attribute with the pipeline, unrolling and dataflow attributes, to provide more BRAM ports.
5. Use the work-item pipeline attribute with NDRange kernels.

We present results for the single problem size of 64x64. This represents a typical domain size used on a single CPU in an MPI decomposition. For all experiments we use -O3 optimization, and a 200 MHz clock frequency (which was found to be the maximum that can be used in most cases. Higher frequencies caused failures to meet timing constraints in the hardware compilation process).

5.1 Reference Implementation

The first coding version for the `L100` algorithm described in Section 2 is the most straightforward in terms of the coding required for both the host OpenCL and the FPGA kernel.

In this version the whole of the algorithm's operations are wrapped within a single `for` loop, as shown in Figure 1(a). Such a loop-based kernel is represented in OpenCL as a *Task* kernel. We denote this basic implementation `L100-Seq` and it acts as the *reference implementation* for subsequent experiments. Table 1 summarizes the performance data. This implementation creates an IP block containing a single Compute Unit (CU). Since the kernel is implemented as a single task there is, by default, only a single work-item to be mapped to the CU by the host. The single work item task can be mapped to the FPGA using a single, simple OpenCL task queue on the host, requiring only a single call to enqueue the work-item. Listing 1.1 shows the key steps of the OpenCL Host code along with an outline of the OpenCL Kernel function code.

Listing 1.1. Reference implementation Host and kernel code

```
//command queue
cl::CommandQueue q(context, device);
//kernel call
kernel_L100(cl::EnqueueArgs(q,
    cl::NDRange(1, 1, 1),
    cl::NDRange(1, 1, 1)),
    buffer_u, buffer_v, buffer_p,
    buffer_cu, buffer_cv, buffer_z,
    buffer_h, fsdx, fsdy);
//Kernel function
__attribute__ ((reqd_work_group_size(1,1,1)))
outer_loop:for (i=0;i<local_n;i++) {
```

```
inner_loop: for (j=0;j<local_n;j++) {
..  }}
```

Xilinx provides analysis tools in the SDK, that support analysis of the L100 kernel performance, and latency reports generated during the build process. Table 1 shows that this reference implementation requires a latency of 4,714,496 clock cycles (CC) (188.4s in time) to execute. A closer look at the timeline analysis of the kernel's operations (available in a report from the build process) shows that one iteration requires 1,151 CC. These cycles are divided between the kernel's arithmetic (float) operations, `fmul, fadd and fsub` and read/write `gmem`, global memory access operations. In the unoptimised reference code the execution of these operations is seen to be carried out sequentially (one operation per clock cycle). Resource usage is given in Figure 2. This straightforward, unoptimised implementation's performance is poor, and the resource usage is low, only a small fraction of the available resources being utilised.

Table 1. The performance effects of each optimisation on the L100 kernel. (Seq) no optimisations; (Max) max DDR memory ports; (BRAMs) use of BRAM memories; (B) use burst mode; (AP) use array partitioning

Experiment Name	Latency (Clock Cycles)	Iteration Latency	Loop trip Count	Execution Time Seconds	Speedup
L100-Seq	4,714 ,496	1151	4096	188.4	-
L100-Max	35,457	853	4096	3.4	55.41 x
L100-Max -BRAM	97,344	1528	4096	2.29	82.27x
L100-Max BRAM-B	14,805	524	4096	0.66	285.48x
L100-Max BRAM -B-AP	14,804	524	4096	0.66	285.48x

Table 1 and Figure 2 also summarise the performance and resource usage results, respectively, of applying the basic optimisations described in Section 5.

The first optimisation we applied to `L100-Seq` is the use of Max Memory Ports which is `L100-Max` in Table 1. Enabling the Max Memory Ports flag in SDx instructs the `xocc` compiler to utilise the four DDR memory ports. This helped the compiler to apply `pipeline` with Initiation Interval (`II`)=4 automatically over the inner loop. The kernel latency reduced to 35,457 CC (132.96x), and the single iteration latency improved from 1151 CC to 853 CC. The execution time improved by 55.41x. The timeline analysis shows that the use of Max Memory Ports optimises the Global memory access operations. Multiple memory operations are now carried out per CC. The use of Max Memory Ports increased the resource usage compared to the reference implementation.

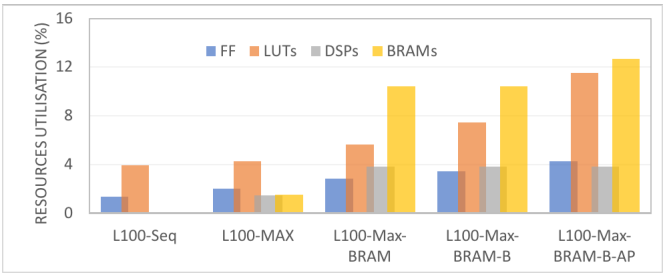

Fig. 2. Effects of basic optimisations on resource utilisation

The next optimisation, applied on the `L100-Max` implementation, is utilising the BRAM memory, this is `L100-Max-BRAM` in Table 1. We introduce local BRAM storage for the kernel input data, then transfer data from the DDR memory to the BRAMs using a `for` loop. The kernel will access the data from the BRAM buffers, which have much lower access latency. After the calculations finish, we transfer the data back from the BRAMs to the DDR memory. The timeline analysis shows that the data movement between the BRAMs and the DDR memory took 1404 CC, and the kernel's operations take only 124 CC with `II`= 1. With the utilisation of the BRAM memory, the compiler perfectly pipelined the inner-loop of the kernel's operations. This implementation took 2.29s to finish, which improved the execution time 82.27x compared to the reference implementation. The resource usage for the `L100-Max-BRAM` implementation is shown in Figure 2.

To improve the data-movement from the DDR memory to the local BRAMs, we used the `burst` mode which is triggered through the use of the pipeline attribute `xcl_pipeline_loop` on the data read/write loops. In Table 1 `L100--Max-BRAM-B` the use of burst mode has improved the performance further and the execution time decreased to 0.66s. The use of `burst` mode reduced the cost of data movement from 1404 CC to 400 CC. That has improved the kernel's latency to 14,805 CC, a 285.48x improvement compared to the reference implementation. `Burst` mode has increased only the use of FFs (3.44%) and LUTs (7.44%).

The final optimisation strategy is the use of array partitioning. A local memory BRAM has only two access ports. Array partitioning provides more BRAM access ports. In Table 1 this implementation, `L100-Max-BRAM-B-AP`, shows that no performance improvement to the implementation `L100-Max-BRAM-B` is noticed with the use of array partitioning. The pipeline `II` is already equal to 1, meaning that the BRAM's access ports were sufficient.

6 Results and Discussion

This section presents the results of experiments exploiting the available concurrency options in Xilinx OpenCL (for host code) and SDSoC HLS (for FPGA device code). Our aim is to provide traditional scientific HPC software devel-

opers with insight into how best to exploit FPGAs in their applications. The results in this section build on top of those of the `L100-Max-BRAM-B-AP` version which includes the low-level optimisations discussed in the previous section.

Exploiting Instruction-Level-Parallelism: Instruction-level parallelism is exploited through the use of the pipeline and unrolling mechanisms for the `L100` kernel's operation loops. The use of -O3 instructs the `xocc` compiler to automatically apply the pipeline mechanism to the inner-loop of the kernel's operations. Thus, in subsection 5 for the `L100-Max-BRAM-B-AP` version the compiler has automatically pipelined the inner-loop.

In Section 2 two loop-based coding options for the kernel were shown. In Figure 1(a) a single loop nest contained all the operations, while in Figure 1(b), a loop nest was used for each operation. The SDSoC build reports reveal that the use of the pipeline attribute on the outer loops explicitly in both these cases does not improve the latency. Pipelining the outer loop means that we are trying to pipeline outer iterations that each have 64 inner iterations. The SDSoC compiler failed to translate this design to a register transfer level (RTL) file and the compilation failed.

The use of a loop per operation, as in implementation L100-P (b) in Table 2, increased execution time to 0.97s. Pipelining separate loops for each operation in sequence is not as efficient as pipelining all operations in a single loop. The

Table 2. ILP mapping performance. (P) pipelining, (a) or (b) code options (a) or (b) in Figure 1; (U) unrolling.

Experiment Name	Latency (Clock Cycles)	Iteration Latency	Trip Count	Execution Time Seconds	Speedup
L100-P (a)	14,805	524	4096	0.66	285.48x
L100-P (b)	29,991	816	4096	0.97	194.22x
L100-U	15,508	542	1024	0.68	277.05 x

performance differences between L100-P (a) and L100-P (b) come from the fact that for a single loop nest (a) pipelining of the execution of all the kernel's operations is performed. In contrast, with one loop nest per operation (b), the operations are executed in sequence. The resource usage of the L100-P (b) was lower that that of L100-P(a).

The use of unrolling (version `L100-U` in Table 2) provides similar performance to that of the pipelined `L100-P (a)`. The kernel execution time is 0.68s (with fewer but larger loops). Unrolling results in arithmetic operations from successive iterations being executed in parallel, however, performance can be impacted due to conflicts when accessing the BRAMs. Conflicts may be avoided by increasing the array partitioning factor to provide more BRAM access ports. The best unrolling factor we found was 4, with an array partitioning factor of 8. `L100-U`

utilised more resources compared with the pipelined version, as can be seen in Figure 3. In terms of coding, we only need to add the pipeline or unrolling

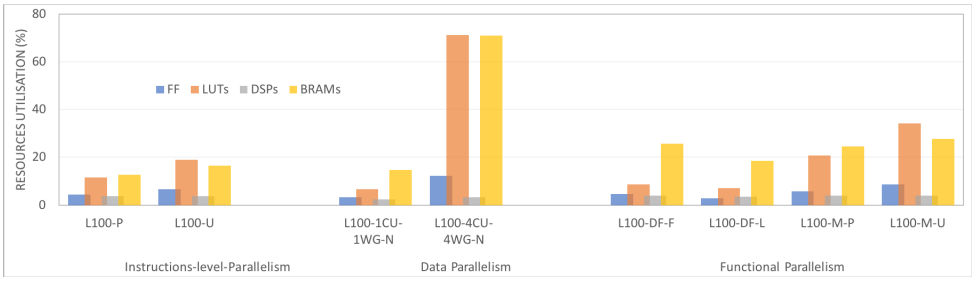

Fig. 3. Bar graph showing the resource utilisation of the optimised concurrency mapping implementations listed in Section 6

attribute to the appropriate loop in the kernel. The host code follows the coding style in Listing 1.1

Exploiting Data Parallelism: Data parallelism is available because the operations of the L100 kernel are entirely independent as discussed in Section 2. The OpenCL NDRange kernel mechanism is suitable for mapping this concurrency type because multiple WIs can be initiated to execute those operations in parallel.

Listing 1.2 shows the coding side of the NDRange mechanism. This coding style is different from that of the Task kernel.

Listing 1.2. NDRange kernel implementation

```
//Host kernel call
kernel_L100(cl::EnqueueArgs(q,
    cl::NDRange(64, 64, 1),
    cl::NDRange(32, 32, 1)),
    buffer_u, ... , fsdx, fsdy);
//NDRange Kernel. 1024 WIs with 4 WGs
__attribute__((reqd_work_group_size(32,32,1)))
//read data in burst mode to Local BRAMs
__attribute__((xcl_pipeline_workitems)) {
loops: l_u[][]=; l_v[][]= l_p[][]=;
}
int j = get_global_id(0);
int i = get_global_id(1);
// WIs Calculate their work
    if(j < N && i < M) {
__attribute__((xcl_pipeline_workitems)) {
        Loops: CU[]=; CV[]=; Z[]=; H[]=;
        }
```

```
    }
// WIs write data back in burst mode
  __attribute__((xcl_pipeline_workitems)) {
    loops: =local_u[][]; =local_v[][];
           =local_p[][];
  }}
```

There are no loops, and we use `xcl_pipeline_workitems` attribute to enable burst-mode and to pipeline the execution of WIs. In the host code, the global and local work size of the kernel is defined using the `cl:NDRange` OpenCL function.

The first, straightforward, NDRange implementation is to create one WG that contains all 4096 WIs which can be executed in parallel. The NDRange functions provide the size and coordinate information within the global work size. The first `NDRange` design is `L100-N-1CU-1WG` in Table 3. This has a kernel with (4096) WIs and one WG that is mapped to one CU.

This kernel's iteration latency has improved 17.9x (64 CC compared to the reference implementation 1151 CC). The SDSoC build reports show that 64 read accesses to local memory are carried out in parallel in one CC, and the next 64 read accesses happen in the next CC. In addition, the kernel's arithmetic operations are pipelined with `II` = 1. The use of `xcl_pipeline_workitems` and multiple WIs has improved the kernel's performance significantly. However, in Table 3, the kernel's total latency did not improve to the same extent. This is because data transfers between DDR memory and the local BRAMs add a high overhead. 29,582 CC out of 38,758 CC were needed for data-movement, a result of congestion as many WIs are trying to access the DDR memory through only 4 DDR ports. The execution time of `L100-N-1CU-1WG` design is 1.17s. For resource usage see Figure 3.

Table 3. Data parallelism: NDRange. (N) NDRange kernel; (CU) No. of compute units; (WG) No. of work-groups.

Experiment Name	Latency (Clock Cycles)	Iteration Latency	Loop trip Count	Execution Time Seconds	Speedup
L100-N-1CU-1WG	38758	64	4096	1.17	161.02x
L100-N-4CU-4WG	15219	396	1024	1.14	165.4x

NDRange data parallelism can be exploited at the WI and WG levels. The kernel's global data size is divided into multiple local work groups. As the L100 algorithm's operations are independent, the WGs can be executed in parallel by mapping each WG to a separate CU. The creation of the WGs is driven by the OpenCL API function `cl:NDRange(32, 32, 1)` in the host where we define the global and the local work size. We specify the number of CUs required through the application project settings in the SDSoC SDK environment.

The implementation `L100-N-4CU-4WG` in Table 3 exploits data parallelism at the WG level, where the global work size, $64 * 64$, is divided by 4 and 4 CUs are requested. The kernel's latency has improved compared to the `L100-N-1CU-1WG`. However, no execution time improvement has resulted. The SDSoC build report shows that there are multiple read/write operations to BRAMs carried out in parallel, and the kernel's arithmetic operations are pipelined, but the latency of accessing the DDR memory has degraded the performance. The 4 CUs consume a high number of LUTs and BRAMs, see Figure 3.

Exploiting Functional Parallelism: Two functional parallelism mapping mechanisms are available as discussed in Section 4. These are `Data-Flow` and `The number of kernels`. These allow each kernel operation to be mapped to a separate compute unit (CU). `Data-Flow` can be applied over functions or loops. Code options 2 and 3 in Figure 1 are suitable for this mechanism. In code option 2, the attribute `xcl_dataflow` can be specified in the kernel and applies to the multiple loops which are executed in parallel. In code option 3 the same attribute applies to the functions.

Table 4 shows the implementations with `Data-Flow` over functions and loops, `L100-DF-F` and `L100-DF-L` respectively. In each implementation 13 CUs has been created by the compiler, 7 CUs for read/write operations and 4 CUs for computing the operations `CU, CV, Z, H`. The kernel's execution timeline in SDSoC shows that the kernel is executed in three pipelined steps: `load, Compute and Store`. In the Compute step the four operations are carried out in parallel.

Table 4. Data parallelism: Dataflow. (DF) dataflow ; (F) apply DF to function code style; (L) apply DF over loops; (M) A kernel per operation; (P) pipeline; (U) unrolling; N NDRange kernel.

Experiment Name	Latency (Clock Cycles)	Iteration Latency	Loop trip Count	Execution Time Seconds	Speedup
L100-DF -F	12,861	-	-	0.60	314x
L100-DF -L	16971	-	-	0.67	281.19x
L100-M-P	56158	88	4096*4	1.62	116.29x
L100-M-U	29974	104	1024*4	1.61	117.01x

The `L100-DF-F` implementation delivered better performance with latency $12,861$ CC. The `L100-DF-L` implementation had $16,971$ CC. `L100-DF-F` and delivered the best execution time, 0.60 seconds, of all the implementations in this paper. The resource usage in Figure 3 shows that the `L100-DF-F` implementation utilised more resources, especially DSPs and BRAMs, than `L100-DF-L`. The performance differences can be related directly to the resource usage.

Table 4 also shows the implementations for the use of `Multiple kernels`. We have created a kernel for each of the operations in L100. The kernels can be `Task`

kernels as in the design in subsection 6, or `NDRange` kernels as in subsection 6. Given the resource usage figures in subsection 6, where one NDRange kernel with 4 WGs mapped to 4 CUs consumed more than 70% of the available LUTs, the use of multiple NDRange kernels is not an option. Therefore, the implementations `L100-M-P` and L100-M-U use Task kernels. There are 4 kernels with either the pipeline or unroll attribute applied on the kernel's operation loop. The use of multiple kernels needs synchronisation from the host, where we use an out-of-order command queue. This command queue issues the execution of the 4 kernels in the same clock cycle. The compiler creates 4 CUs and each kernel is mapped to a specific CU. The compiler reports show that the latency is high, however, and execution time is higher in both implementations (`L100-M-P` and `L100-M-U`) compared to the Data-Flow mechanism.

With both `Data-Flow` and `Multiple kernels` L100's operations are carried out in parallel. However, with Multiple kernels, we increase the data transfer latency. The data transferred to the local BRAM are U, V and P for one kernel, and for four kernels ten data memory transfers are required from the DDR memory. These data transfers are inefficient and there are only four DDR memory ports to carry ten memory access requests. In the Data-Flow mechanism, only three memory data read accesses are carried out in parallel, which helps explain the performance difference. This overhead has an impact on latency figures in Table 4 for the `L100-M-P` and L100-M-U implementations. However, in terms of computational iteration latency, `L100-M-P` and L100-M-U report 88 CC and 104 CC compared to that of the reference implementation with 1151 CC. `L100-M-P` has lower resource usage than L100-M-U, as shown in Figure 3.

Performance and resource usage: The performance of the different mechanisms was presented in terms of latency and execution time in Section 6. In addition, we have observed the iteration latency in different implementations. Different ways were explored for mapping the concurrency in the L100 kernel to the target FPGA. The results show that using *Data-Flow over functions*, targeting the functional parallelism in the kernel, provided the best performance in terms of both latency and execution time. Further, this approach utilises fewer FPGA resources compared to the use of multiple kernels or the multiple WG NDRange mechanism.

Code option 3 in Figure 1 proved to be the better choice to map the L100 concurrency. Since the kernel's operations are independent, processing as many in parallel as possible is the best method to achieve good performance. `L100-DF-F` has latency of just under 13,000 CC and execution time of 0.60s, a speedup of 314x over the reference. The Data-Flow over functions mechanism assigned each operation in the kernel to its own CU and executed them in parallel after filling the local BRAMs with the required input data. Exploiting instruction level parallelism with the pipeline mechanism (`L100-P(a)`) came second with nearly 2000 CC extra latency and execution time 0.66s and speedup 285.48x, with similar resource usage, except that Data-Flow used more BRAMs.

In terms of iteration latency (a measure of the level of parallelism), the use of the NDRange mechanism with 1CU and 1WG provided a speedup of 17.9x.

This implies a potential to exploit a high level of parallelism, since 4096 WIs are set to be executed in parallel. However, the pressure on DDR memory bandwidth to support 4096 WIs severely degrages the performance in this case. In terms of resource usage, Figure 3 shows that the multiple WGs NDRange kernel and multiple kernels implementation consumed the highest percentage of resources. Utilising a large amount of resources does not guarantee high performance, mainly due to memory access overheads. However, use of fewer resources can lead to lower power use, though we did not pursue this here.

7 Conclusions

This paper explored the mechanisms available to scientific programmers in a relatively high-level language, OpenCL, to map the concurrency in their algorithms to FPGAs. The trade-offs in the different approaches in terms of performance and resource usage were discussed along with issues related to progamability.

Data movement is a critical issue. The optimisation strategies in Section 5 are aimed at improving the data-movement. However, the results show that a more in-depth study is needed in future. There are a number of data movement optimisation strategies that can explored, including: utilising on-chip global memory, exploiting the full width of DDR ports and better use of the DDR memory banks.

Acknowledgment

Moteb Alghamdi is sponsored by Taibah University, Madinah, Saudi Arabia. Support was also from IS-ENES3 (grant agreement no. 824084), funded by the European Union's Horizon 2020 Research Infrastructures Programme.

References

1. R. Dimond, S. Racaniere, and O. Pell, "Accelerating large-scale HPC Applications using FPGAs," in *Computer Arithmetic (ARITH), 2011 20th IEEE Symposium on.* IEEE, 2011, pp. 191–192.
2. N. Brown, "Exploring the acceleration of the met office nerc cloud model using fpgas," in *International Conference on High Performance Computing.* Springer, 2019, pp. 567–586.
3. 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.
4. L. Gan, H. Fu, C. Yang, W. Luk, W. Xue, O. Mencer, X. Huang, and G. Yang, "A highly-efficient and green data flow engine for solving euler atmospheric equations," in *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on.* IEEE, 2014, pp. 1–6.
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 on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.

6. Vivado-HLS, "Xilinx Vivado SDK," https://www.xilinx.com/products/design-tools.html, 2020, [Online; accessed 01-September-2020].
7. Intel, "Intel FPGAs SDK for OpenCL," 2-https://www.intel.co.uk/content/www/uk/en/software/programmable/sdk-for-opencl/overview.html, 2020, [Online; accessed 01-September-2020].
8. Xilinx, "Xilinx SDSoC Environment User Guide UG1027 (v2019.1)," https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1027-sdsoc-user-guide.pdf, 2019, [Online; accessed 01-September-2020].
9. Maxeler, "Maxeler SDK," https://www.maxeler.com/products/desktop/, 2020, [Online; accessed 01-September-2020].
10. 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 Programming*, vol. 2019, 2019.
11. M. Vestias and H. Neto, "Trends of cpu, gpu and fpga for high-performance computing," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2014, pp. 1–6.
12. E. E. Projects, "European Exascale Projects," http://exascale-projects.eu/, 2020, [Online; accessed 01-September-2020].
13. A. Munshi, B. Gaster, T. G. Mattson, and D. Ginsburg, *OpenCL programming guide*. Pearson Education, 2011.
14. L. Solis-Vasquez and A. Koch, "A case study in using opencl on fpgas: Creating an open-source accelerator of the autodock molecular docking software," in *FSP Workshop 2018; Fifth International Workshop on FPGAs for Software Programmers*. VDE, 2018, pp. 1–10.
15. H. R. Zohouri, "High performance computing with fpgas and opencl," *arXiv preprint arXiv:1810.09773*, 2018.
16. H. M. Waidyasooriya, M. Hariyama, and K. Uchiyama, *Design of FPGA-based computing systems with OpenCL*. Springer, 2018.
17. R. Sadourny, "The dynamics of finite-difference models of the shallow-water equations," *Journal of the Atmospheric Sciences*, vol. 32, no. 4, pp. 680–689, 1975.
18. Xilinx, "Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit," https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html, 2020, [Online; accessed 01-September-2020].
19. Q. Jia and H. Zhou, "Tuning stencil codes in opencl for fpgas," in *2016 IEEE 34th International Conference on Computer Design (ICCD)*. IEEE, 2016, pp. 249–256.
20. D. Chen and D. Singh, "Using opencl to evaluate the efficiency of cpus, gpus and fpgas for information filtering," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2012, pp. 5–12.
21. Z. Wang, B. He, W. Zhang, and S. Jiang, "A performance analysis framework for optimizing opencl applications on fpgas," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 114–125.
22. F. B. Muslim, L. Ma, M. Roozmeh, and L. Lavagno, "Efficient fpga implementation of opencl high-performance computing applications via high-level synthesis," *IEEE Access*, vol. 5, pp. 2747–2762, 2017.
23. V. Anshuman, A. E. Helal, K. Krommydas, and W.-c. Feng, "Accelerating workloads on fpgas via opencl: A case study with opendwarfs," *Computer Science Technical Reports*, 2016.
24. Optimisation-Guide, "Xilinx SDAccel optimisations guide," https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1207-sdaccel-optimization-guide.pdf, 2020, [Online; accessed 01-September-2020].