International Journal of Parallel Programming (2024) 52:253–273
https://doi.org/10.1007/s10766-024-00770-3

High-Level Programming of FPGA-Accelerated Systems with Parallel Patterns

Björn Birath[1] · August Ernstsson[1] · John Tinnerholm[1] · Christoph Kessler[1]

Received: 3 November 2023 / Accepted: 1 May 2024 / Published online: 27 May 2024
© The Author(s) 2024

Abstract

As a result of frequency and power limitations, multi-core processors and accelerators are becoming more and more prevalent in today's systems. To fully utilize such systems, heterogeneous parallel programming is needed, but this introduces new complexities to the development. High-level frameworks such as SkePU have been introduced to help alleviate these complexities. SkePU is a skeleton programming framework based on a set of programming constructs implementing computational parallel patterns, while presenting a sequential interface to the programmer. Using the various skeleton backends, SkePU programs can execute, without source code modification, on multiple types of hardware such as CPUs, GPUs, and clusters. This paper presents the design and implementation of a new backend for SkePU, adding support for FPGAs. We also evaluate the effect of FPGA-specific optimizations in the new backend and compare it with the existing GPU backend, where the actual devices used are of similar vintage and price point. For simple examples, we find that the FPGA-backend's performance is similar to that of the existing backend for GPUs, while it falls behind in more complex tasks. Finally, some shortcomings in the backend are highlighted and discussed, along with potential solutions.

Keywords Algorithmic skeletons · Reconfigurable computing · FPGA · Single-source heterogeneous programming

✉ August Ernstsson
august.ernstsson@liu.se

Björn Birath
bjorn@birath.dev

John Tinnerholm
john.tinnerholm@liu.se

Christoph Kessler
christoph.kessler@liu.se

[1] PELAB, Department of Computer and Information Science, Linköping University, Linköping, Sweden

1 Introduction

For a long time, the trend in computer architecture has been the move to multi-core processors. Additionally, the use of accelerators such as massively parallel GPUs has increased, leading to many of today's systems being heterogeneous. An alternative accelerator to GPUs is the *field-programmable gate array* (FPGA). The strength of FPGAs is that they can be reconfigured and adapted for the type of algorithms to execute, mapping an algorithm one-to-one to the FPGA hardware. This involves "programming" the FPGA using a *hardware description language* (HDL) such as VHDL or Verilog [6]. The HDLs are used to generate a circuit description which is loaded onto the FPGA.

Historically, this process of programming FPGAs has required specialized training since HDLs lack many high-level constructs found in conventional programming languages, and use a parallel data flow model rather than a sequential one. There are also differences between FPGA platforms, leading to difficulties of reusing existing designs [12]. To alleviate these issues, there have been many attempts to create tools that utilize higher-level languages, such as C or C++, to automatically produce a circuit specification in a HDL. These *high-level synthesis* (HLS) tools allow developers to program FPGAs faster and without hardware expertise [20]. More recently, both Intel[1] and Xilinx[2] introduced HLS toolchains based on *OpenCL*, a framework for creating portable parallel programs targeting multiple types of platforms.

While these tools make it easier to program FPGAs, developers still need to handle the challenges of programming against heterogeneous processors. This includes communication, memory management and synchronization. Here, skeleton programming frameworks can provide a more high-level interface for the developer [11] by abstracting from some of the more complex interactions and specifics of a multiprocessor system. One such framework is SkePU,[3] an open-source skeleton programming framework for heterogeneous parallel systems. Today SkePU supports multi-core CPUs, GPUs and clusters. By adding support for FPGAs, it would allow developers to program FPGAs without the need for hardware expertise or deep knowledge of OpenCL.

To this end, we design and implement a new backend in SkePU targeting reconfigurable architectures by integrating an existing OpenCL HLS toolchain. This will allow SkePU to further accelerate the types of problems that FPGAs are particularly suitable for, such as problems that can take advantage of the high-throughput pipelines that FPGAs can create, while keeping full source-code portability with multicore CPU, GPU and cluster execution.

Overall, this paper makes the following main contributions:

- We present the design and implementation of a new OpenCL-based backend for FPGA, including FPGA-specific optimizations, atop Intel OpenCL SDK for FPGA, for the SkePU skeletons Map, Reduce, MapReduce, Scan, and MapOverlap.

1 https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html

2 https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/ehb1504034292718.html

3 https://skepu.github.io/

```
1  #include <skepu>
2
3  float mult(float a, float b) { return a * b; }
4  float add(float a, float b)  { return a + b; }
5
6  int main(int argc, char *argv[])
7  {
8      size_t const size = 100;
9      skepu::Vector<float> a(size), b(size);
10     auto dot_product = skepu::MapReduce(mult, add);
11     float res = dot_product(a, b);
12 }
```

Listing 1: SkePU program that calculates the dot product of two vectors.

– We evaluate the FPGA backend on an Intel Programmable Acceleration Card with an Arria 10 GX FPGA. We demonstrate the great performance benefit of applying FPGA-specific optimizations such as loop unrolling, register pipelining and skeleton fusion in the backend. We also identify challenges for future improvements of the FPGA backend, such as performance issues with complex kernels.

The remainder of this paper is organized as follows: Sect. 2 presents background on algorithmic skeletons and SkePU, and Sect. 3 on FPGAs. Section 4 discusses related work. Section 5 presents the implementation of the FPGA backend, Sect. 6 experimental results and discussion. Section 7 proposes future work and Sect. 8 concludes.

2 Skeleton Programming and SkePU

Skeleton programming is a programming model offering pre-built *skeletons*, generic programming constructs derived from higher-order functions that match different computation patterns for which parallel implementations are provided by the framework. Problem-specific user code is inserted as function arguments (*user functions*) into a skeleton to instantiate a complete algorithm [5].

SkePU [9] is a C++ open-source skeleton programming framework that provides an interface to create parallel computations with support for different backends: sequential, multi-core CPU (OpenMP), GPU (CUDA or OpenCL), cluster (StarPU-MPI), and combinations of those. SkePU programs define user functions that the skeletons use as operators. SkePU contains a source-to-source compiler that translates user functions for each backend and a runtime library that handles the scheduling, communication, and memory management between the host and backend [10]. Listing 1 shows an example SkePU program that calculates the dot product of two vectors, using the `MapReduce` skeleton and two user functions: `mult` and `add`.

SkePU implements a set of data-parallel skeleton patterns. The basic `Map` is a fundamental building block in SkePU programs, as it provides a flexible interface, e.g., with variadic input and output arity and optional use of non-trivial memory access patterns. `MapOverlap` and `MapPairs` are optimized extensions of `Map` for stencil computations and Cartesian-product patterns, respectively. Similarly, `Reduce` and

`Scan` are specialized patterns for *reductions* and *prefix sums*. SkePU offers efficient combinations when a reduction is used after a map-based pattern through `MapReduce` and `MapPairsReduce`. Each skeleton is instantiated with one or more user functions, which contain the program-specific code, executed in parallel according to the respective pattern semantics.

SkePU provides so-called *smart data-containers* [7] that manage memory and coherency between host and backends automatically. In the latest version of SkePU there are four different types of smart containers for four different dimensionalities: `Vector` (1D), `Matrix` (2D), `Tensor3` and `Tensor4`.

Smart containers are C++ objects in main memory and can therefore not be used directly in user functions. While this is not needed for element-wise access, some computations require access to all elements in a container. SkePU therefore provides *proxy containers* which can be used to access any element in a container inside a user function.

3 Field-Programmable Gate Arrays (FPGAs)

*FPGA*s are computer chips that can be programmed to implement different digital circuits. The term comes from the fact that FPGAs are programmable in-field even after deployment. They consist of an array of configurable logic and I/O blocks that are connected through a network with programmable switches. Modern FPGAs often also have *hard blocks*, blocks that cannot be configured but instead implement a specific functionality such as multipliers or Ethernet interfaces. Another common hard block is RAM, since implementing RAM using the configurable logic is much less area efficient [4].

Programming an FPGA is divided into three stages. First the desired hardware circuit is described in an HDL. This is then translated to logic gates via synthesis, which generates the physical design. Later, the place-and-route stage maps the physical design to a device. Constraint checks ensure that the design will fit on the chosen device and has no timing errors. Finally, a hardware configuration file, a *bitstream*, is generated which can be loaded onto the FPGA. The process can take hours or days to complete [2].

Hardware description languages (HDLs) describe hardware circuits that are programmed to FPGAs. While HDLs might look procedural, they are not. For example, rather than a sequential control flow model, HDLs use a data flow model where statements can run in parallel whenever the input is changed.

Since HDL code represents the hardware it synthesizes to, an understanding of circuit design is required to get the best use out of an FPGA. To make FPGA programming easier, there has been focused attention in the industry to create *high-level synthesis* (HLS) tools for converting code written in a high-level language to HDL code [2]. The generated HDL code can then be synthesized using the normal FPGA programming flow.

Since synthesis takes multiple hours, the standard OpenCL kernel just-in-time compilation cannot be used. Instead, the SDK compiles OpenCL into Verilog, which is passed to a synthesis program. The compiler is an extension of the LLVM compiler

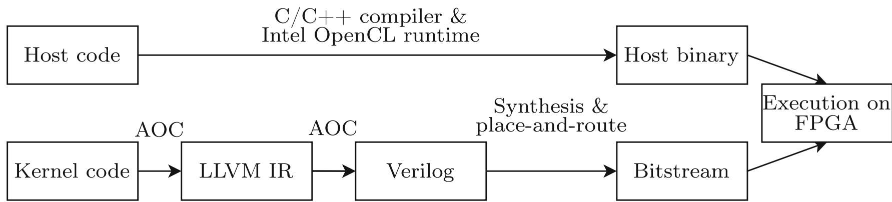

Fig. 1 Intel OpenCL SDK compilation flow. AOC is the OpenCL compiler in the SDK

which first produces a LLVM intermediate representation of the kernel, from which Verilog code is produced, followed by the normal FPGA programming flow of synthesis and place-and-route [6]. Figure 1 shows the compilation flow when using the SDK.

Czajkowski et al. [6] state that the reason for choosing OpenCL over a high-level language is the separation between the host and kernel. The kernel can be implemented as a highly performant hardware circuit while the host can handle the communication and programming of the FPGA. This means that the entire system can be implemented as opposed to other HLS tools that only generate HDL code for synthesis.

The SDK generates *hardware pipelines* based on the OpenCL code in the kernel. A pipeline's size is measured in terms of its *depth*, which is how many stages there are before the final output, and its *width* which is how many operations are done in parallel in each stage.

The SDK supports two execution models: NDRange and Single Work-Item. Both use pipelining to achieve parallelism, but differ in how they issue new data into the pipeline.

When using the *NDRange* model, the kernel is executed once for each work-item. This is the model most commonly used on GPUs, as multiple work-items can execute in parallel. Each region between subsequent barrier calls will generate an independent pipeline that is flushed at the barrier. The work-items are issued into the pipeline iteratively by a run-time scheduler generated in hardware. If the kernel references any of the indices in the NDRange or uses a barrier, the kernel is interpreted as a NDRange kernel [26].

The SDK's programming and best practice guides recommend two techniques for better performance: Kernel vectorization and compute unit replication. *Kernel vectorization* is achieved by using the `num_simd_work_items` attribute (as seen in Listing 2), which instructs the compiler to translate each scalar operator to a SIMD operation. This allows the programmer to increase the throughput of the kernel without any modifications to the kernel code or NDRange used in the invocation of the kernel [23]. The vectorization fails if the kernel contains code that the compiler deems "SIMD-unfriendly", such as thread-dependent branching. Vectorization can be done manually, as in Listing 3, but then the NDRange must be manually changed to match the number of items each kernel handles.

For *compute unit replication*, using attribute `num_compute_units(N)` instructs the compiler to replicate the full pipeline `N` times. Work-groups are split across all compute units, scheduling is handled by a hardware scheduler.

```
__attribute((num_simd_work_items(2)))
__kernel void sum( __global float* a, __global float* b,
__global float* result) {
int gid = get_global_id(0);
result[gid] = a[gid] + b[gid];
}
```

Listing 2: Example of kernel vectorization.

```
__kernel void sum( __global float* a, __global float* b,
__global float* result) {
int gid = get_global_id(0);
result[gid * 2 + 0] = a[gid * 2 + 0] + b[gid * 2 + 0];
result[gid * 2 + 1] = a[gid * 2 + 1] + b[gid * 2 + 1];
}
```

Listing 3: Manual kernel vectorization.

Both methods increase throughput by increasing the amount of hardware generated. It is recommended to first use kernel vectorization, as this generates coalesced memory accesses and less total hardware. The methods can also be combined, which can give better throughput depending on what type of work the kernel performs [17].

In *the Single Work-Item* model, the kernel is executed only once as a single work-item. High performance is achieved by pipelining loop iterations, mapping each outer loop to a separate pipeline. This allows multiple loop iterations to be computed in parallel, allowing a pipelined loop to finish faster than a non-pipelined loop. No run-time scheduler is used, instead the scheduling is determined at compile-time. The *initiation interval* II [1] is the number of clock cycles between two subsequent loop iterations being issued into the pipeline. For a pipelined loop with an input size of L and a depth of P, the total amount of clock cycles to complete is

$$T_{cycles} = P + II * (L - 1) \quad (1)$$

which can be converted to time in seconds by $T_{seconds} = T_{cycles}/f_{max}$, where f_{max} is the operating frequency of the FPGA. As f_{max} is often fixed for each FPGA, P not contributing much to the execution time and L being application dependent, II is the one parameter the developer can change with the largest impact on the performance of a single work-item kernel. The compiler always tries to pipeline loops so that the II is 1, but loop-carried dependencies like data dependencies or memory dependencies can cause the II to increase. The run-time II can also differ from the II determined at compile time due to stallable load and store operations or nested loops [25], so the actual execution time can be longer than the calculated $T_{seconds}$.

Parallelism similar to vectorization can be added to single work-item kernels by unrolling loops, increasing the width and depth of the pipeline at the cost of more hardware being used. shows an example of how loop unrolling is applied and the result of the unrolling.

```
float reg[SIZE + 1] = {0.0};

for (int i = 0; i < N; ++i) {
#pragma unroll
for (int j = 0; j < SIZE; j++)
// Shifts the content one step per iteration
reg[j] = reg[j + 1];
reg[SIZE] = reg[0] + input[i];
}
```

Listing 4: Creating and using a shift register in OpenCL.

If the length of the loop is known at compile time, the compiler can fully unroll the loop, otherwise it can be unrolled by a user-specified factor. On top of increasing parallelism, loop pipelining also allows the compiler to coalesce memory operations, reducing the amount of global memory accesses [24]. Equation 1 can be extended with loop unrolling:

$$T_{cycles} = P' + II * \frac{(L - N_p)}{N_p} \tag{2}$$

where P' is the new depth of the pipeline and N_p is the unroll factor. Assuming $L >> P'$, unrolling should result in a theoretical performance improvement of almost N_p times. This does, however, not take the run-time II into account, so in practice the performance improvement will not be as large.

Shift registers is a technique that can be used to relax some loop-carried dependencies in pipelined loops. Shift registers are implemented using the FPGA's registers, which have an access latency of one clock cycle [25], meaning they can be accessed without increasing the loop's II. An array in OpenCL will be implemented as a shift register if (1) the array size is known at compile time, (2) all accesses to the array are made with addresses known at compile time, and (3) all content in the array is shifted by a compile-time known amount in each loop iteration (see Listing 4).

The primary way to use a shift register is to increase the dependency distance for a variable, by writing values to one end of the shift register and operating on the other end of the shift register. This way operations that take more than one clock cycle to perform can be used without increasing the loop's II as different loop iterations will operate on different parts of the shift register. Shift registers can also be used for data sharing across loop iterations by reusing values across multiple iterations, creating a *sliding window* [16]. This is especially applicable to computations where a stencil is used, which otherwise requires multiple redundant memory accesses per loop iteration to read the input elements for the stencil. If the elements are instead stored in a shift register, one element being read per iteration, the entire stencil is accessible in a single clock cycle.

4 Related Work

We review three high-level frameworks similar to SkePU that target FPGAs, of which two use OpenCL and one is fully implemented in an HDL.

Melia [24] is a MapReduce framework for FPGAs that uses the Intel FPGA OpenCL SDK. The framework lets the user define a *map* and *reduce* function in the OpenCL C language which are then compiled, synthesized and used to configure an FPGA. Melia includes memory optimizations such as coalescing and ”private memory optimization” and applies some FPGA-specific optimizations: converting nested loops to a single loop, loop unrolling, and pipeline replication. Loop unrolling is the only optimization that is performed automatically on the user functions. The user is responsible for applying memory optimizations and must pass parameters to Melia for the pipeline replication and loop unrolling before synthesis. Since synthesis is a long process, Wang et al. [24] developed a cost model using the resource estimation tool included in the Intel FPGA OpenCL SDK. The cost model estimates the execution time of a given OpenCL kernel by multiplying the estimated hardware frequency and estimated amount of clock cycles needed to execute the kernel. Through testing they found that their model could closely predict the hardware frequency of a kernel, and generally capture the trend of the required clock cycles. Using the model, a user can experiment with different parameters in a matter of minutes instead of the hours it would take to complete a full synthesis. Applying the FPGA-specific optimizations to seven common MapReduce applications led to speedups of $1.4\times$ to $43.6\times$. The Melia implementations demonstrated high energy efficiency compared to CPU and GPU implementations and were not much slower than the GPU implementations.

OpenACC-to-FPGA [17] is a framework for translating OpenACC C programs to a hardware configuration file for running on FPGAs. It is an extension of the *Open Accelerator Research Compiler* (OpenARC) [18], an open-source compiler for OpenACC which supports CUDA and OpenCL as backend programming models using source-to-source translation. OpenACC-to-FPGA uses the OpenCL backend to generate OpenCL code, which is passed to the Intel FPGA OpenCL SDK to generate the hardware configuration file. To generate efficient OpenCL code for FPGAs, OpenACC-to-FPGA adds to OpenARC boundary check elimination and directives for controlling loop unrolling, kernel vectorization and compute unit replication (pipeline replication).

The OpenACC-to-FPGA runtime performs dynamic memory-transfer alignment of memory that will be transferred to maximize throughput and lowering latency when transferring data between the host and FPGA memory. Correctly aligned memory on both the host and device allows the Intel FPGA OpenCL runtime to use direct memory access (DMA) between host and FPGA, speeding up data transfers. By this method, Lee et al. [17] achieved a 100-fold speed-up in data transfers between the host and device, in both directions.

In follow-up work, Lambert et al. [16] extend OpenACC-to-FPGA by optimizations for single work-item kernels. First, they added FPGA-specific loop collapsing, changing the existing OpenARC loop collapsing to calculate the indexes of the collapsed loops without modulo and divisions operations, which are relatively expensive on FPGAs. A reduce-specific optimization was also added. It generates OpenCL code for reduce loops that uses shift registers to relax the data dependency that can occur in

reduce loops when using instructions that take more than one clock cycle. It also adds a new `window` directive which generates OpenCL code for creating a sliding window for stencil operations. Based on offsets used to access the stencil, it automatically generates a shift register large enough to store the stencil and the offsets to be used to access the window inside the shift register. Unlike Melia, OpenACC-to-FPGA does not provide any tools or models to alleviate the long synthesizing process.

FPMR is a MapReduce framework for FPGAs [22], though unlike Melia and OpenACC-to-FPGA it does not use any HLS tools. Instead, the framework is implemented in an HDL, providing data synchronization, scheduling and handling communication between the map and reduce tasks. The user implements the *map* and *reduce* user functions by designing a *mapper* and *reducer* processor using the corresponding interfaces in FPMR.

During execution, a processor scheduler, which is implemented on the FPGA, is used to dynamically utilize the mapper and reducer processors using a set of queues for idle processors and tasks for both types of processors.

The framework uses three levels of storage: Global memory, local memory and register files inside each processor. Global memory is implemented using SDRAM modules, providing large capacity and high bandwidth. It is managed by a data controller responsible for transferring data between the host and device memory, dispatching requested data to the mappers, and storing output data from the reducers. An important feature of the data controller is the *common data path*, which allows the controller to overlap data transfer to multiple mappers at once. This is useful for applications where some data are the same for all mappers. The local memory stores the mappers' intermediate results before being passed to a reducer. It is implemented with on-chip RAM giving it low access latency, and if multiple RAMs are implemented, it can also be accessed simultaneously by mappers and reducers.

In a case study of the RankBoost algorithm using FPMR, Shan et al. [22] obtained 31.8× speedup over their CPU reference implementation, which was comparable to a manually designed FPGA implementation with 33.5× speedup.

5 SkePU FPGA Backend

The first part of the implementation was to integrate the Intel FPGA OpenCL SDK into SkePUs backend code generation. SkePU provides OpenCL files to the SDK and accepts generated bitstream files in return. The user is responsible for ensuring that an installation of the SDK and the relevant FPGA board packages are available.

The runtime was also extended to recognize Intel FPGAs and the Intel FPGA emulator as valid OpenCL devices, see Fig. 2. While this initial implementation worked, it used the OpenCL backend designed for GPUs, which resulted in subpar performance on FPGAs. Therefore, a new FPGA-specific backend was created, similar to the OpenCL backend for GPUs with specialized kernel code generation for the respective skeletons. Functionality such as device detection and memory management is shared with the OpenCL GPU backend.

An initial design decision was whether to generate *single work-item* (SWI) or *NDRange* kernels. Using NDRange kernels would have meant that the existing

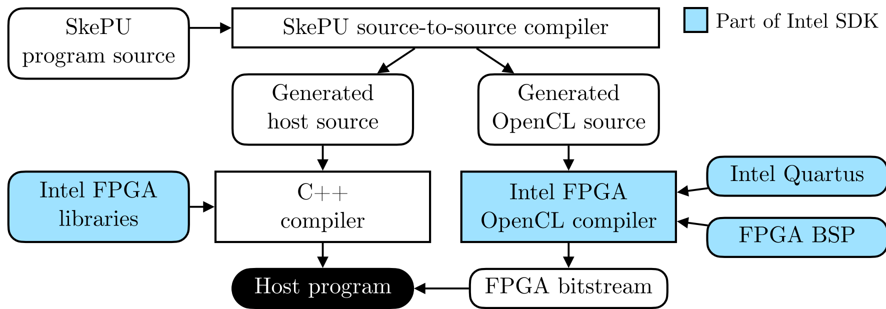

Fig. 2 FPGA backend compilation flow

OpenCL code generation could have been reused. However, those kernels use branches that depend on the global id of the work-items, meaning that automatic vectorization cannot be applied. This is a key optimization for NDRange kernels, so new code generation would have had to be done either way. Furthermore, Intel's general recommendation is to use SWI kernels, and they are faster for many types of problems [25, 26]. It was therefore decided to implement code generation of SWI kernels. The main goal for the skeleton implementations was to reach an II of 1 for the main loop and not reducing the maximum frequency, while still pipelining the main loop.

The optimization techniques used are taken from Intel's programming documentation[4,5] and papers that applied the techniques [12, 25, 26]. The compiler referenced in the following sections is the Intel FPGA SDK OpenCL Offline Compiler.

The Intel documentation recommends some general optimizations that can be applied to all kernels, mostly focusing on helping the compiler:

- Using the `restrict` keyword on pointer arguments which never alias to other pointers. This can prevent the compiler from assuming memory dependencies between read and write operations.
- Using the `const` keyword on any read-only buffers. This allows the compiler to perform more optimizations on load operations.
- Applying the `uses_global_work_offset(0)` attribute. Applying this attribute allows the compiler to not generate hardware for supporting kernel invocations with a non-zero `global_work_offset` (which is never not zero in the FPGA backend), reducing area usage.

SkePU's memory allocation code was modified to allocate 64-byte aligned memory for all host buffers. The dynamic memory-transfer alignment logic used in OpenACC-To-FPGA was also implemented to handle cases when the buffer is not aligned on the FPGA. This guarantees that both the host and device buffer will be aligned when transferring memory between them, allowing all data transfers larger than 64 bytes to use direct memory access.

[4] https://intel.com/content/www/us/en/docs/programmable/683521/21-4/introduction-to-pro-edition-best-practices.html

[5] https://intel.com/content/www/us/en/docs/programmable/683846/21-4

We now present the implementations of the Map, Reduce, MapReduce, Scan and MapOverlap skeletons in the FPGA backend.

Map The Map kernel was replaced by a for loop with a controllable unroll factor. As the loop contains no loop-carried dependencies, it did not require the use of a shift register or any other techniques to attain an II of 1.

Reduce Firstly, the generated kernel was changed to a single work-item kernel instead of an NDRange kernel. This meant the kernel could be reduced to a single for loop, which the compiler can pipeline. However, such reduce loops often result in a loop-carried data dependency on the reduce variable. If the user function is complicated or uses instructions that are expensive to execute on FPGAs, this cannot be done in a single clock cycle. This will increase the II of the loop to the number of clock cycles the user function takes.

The reduce kernel tries to relax such data dependencies by increasing the number of variables that store the intermediate result of the reduction using a shift register. In each iteration, the head of the shift register is read, and the partial result is written to the tail of the register. If the size of the shift register is equal to or greater than the number of cycles that the user function takes to execute, the data dependency can be eliminated.

Finally, parallelism is increased by adding partial loop unrolling. This could be applied to the main loop, but doing so acts as a multiplier to the latency of the loop [16], increasing the II. There are two ways to solve this: increasing the size of the shift register by the unroll count or performing manual loop unrolling as done by Zohouri [25]. Increasing the size of the shift register also increases the total area consumption, so for large unroll factors this can become impractical. Therefore manual loop unrolling was used in the reduce kernel. An example of a generated reduce kernel can be seen in Listing 5, where lines 17-22 show the manual loop unroll.

MapReduce The MapReduce kernel used the same approach as the Reduce kernel, with an added call to the Map user function in the manual loop unroll and in the ramp-up phase. In the normal OpenCL backend, MapReduce is a two-phase kernel where the second phase performs a final reduction. This is not needed in the FPGA version, as the full reduction is performed in a single kernel execution, making the call to the reduce-only kernel unnecessary.

Scan The Scan kernel uses a shift register and loop unrolling. It begins with a prelude to populate the shift register, and after that it reads a single element from the input each iteration and applies it to the user function together with an element from the shift register, the result being stored in the end of the shift register. The shift register is used to avoid the memory dependency that would otherwise be created when immediately accessing the previously read element in the next loop iteration. To also avoid a data dependency if the user function latency is larger than one, the shift register is accessed using an offset `OFFSET` that should be equal to or larger than the latency. This means that there will be `OFFSET` loop iterations between an element being read from main memory and that element being accessed from the shift register. This relaxes both the memory dependency and potential data dependency, assuming the offset is large enough, and allows the outer loop to attain an II of 1.

The results stored in the shift register are not complete when initially written, since the offset also needs to be taken into account. To compensate for this, a scan

```
__kernel void reduce(__global float const* restrict input,
__global float* restrict output,
unsigned long size)
float shift_reg[LATENCY + 1];
#pragma unroll
for (int i = 0; i < LATENCY; i++) {
shift_reg[i] = input[i];
}
int exit = (size
(size / UNROLL) :
(size / UNROLL) + 1;
for (int i = 0; i < exit; i++) {
float partial_result = (
LATENCY <= i * UNROLL && i * UNROLL < size
) ? input[i * UNROLL] : (float) {0};
#pragma unroll
for (int j = 1; j < UNROLL; j++) {
int index = i * UNROLL + j;
partial_result = (index < size) ?
user_func(partial_result, input[index]) :
partial_result;
}
shift_reg[LATENCY] = user_func(shift_reg[0], partial_result);
#pragma unroll
for (int j = 0; j < LATENCY; j++) {
shift_reg[j] = shift_reg[j+1];
}
}
float result = shift_reg[0];
#pragma unroll
for (int i = 1; i < LATENCY; i++)
result = user_func(shift_reg[i], result);
output[0] = result;
```

Listing 5: Example of a generated reduce kernel.

is performed on the first OFFSET elements in the shift register before writing the result to the output. This starts after the shift register has shifted the first input enough times, which depends on the size of the shift register, offset and if the scan is inclusive, according to $delay = size - offset + inclusive$. The outer loop will need extra iterations to write all elements to the output, but assuming the input size is large, it will not affect the performance.

As found by Lambert et al. [16], to allow the outer loop to be unrolled while not increasing the II, the shift register size should be $newsize = size * unrollfactor$. This does lead to a high area usage since the user function will be unrolled both by the outer loop and the final scan loop. As a result, the size of the unroll factor is limited compared to the other skeletons when using floating-point types, since these multiple unrolls of the user function quickly consume all hard-blocks used for floating-point arithmetic.

MapOverlap MapOverlap was the most complicated skeleton implemented on the FPGA backend. Therefore only the one-dimensional version was implemented for

Table 1 Single-skeleton test programs with user functions used in the evaluation

Test program	Skeleton	User function
Adding squares	Map	$f(a, b) = a^2 + b^2$
Global sum	Reduce	$f(a, b) = a + b$
Dot product	MapReduce	$f_M(a, b) = a * b$, $f_R(a, b) = a + b$
Prefix sum	Scan	$f(a, b) = a + b$
Overlap average (1D stencil)	MapOverlap	$f(region) = average(region)$

now, supporting the `Vector` variant and both the row- and column-wise `Matrix` variants. All edge handling modes are supported.

The implementation uses a shift register for all three variants, while loop unrolling is only applied to the main loop in the `Vector` variant due to too high resource usage in the other variants. The edge handling modes are implemented in a single kernel for each variant, which is one of the culprits for the larger hardware usage, as each unroll needs to include the logic needed to handle the different modes.

6 Experimental Results and Discussion

The FPGA evaluations were performed on Intel Devcloud[6] using a Programmable Acceleration Card (PAC) with an Arria 10 GX FPGA, with 1150000 logical elements and 1518 DSP blocks, connected via PCIe. Version 19.4.0 of the Intel FPGA OpenCL SDK was used to compile the FPGA kernels. The GPU benchmarks were run on a NVIDIA Tesla V100 SXM2 32GB GPU connected via PCIe. Both devices have a similar release date and price point and were therefore deemed to be a fair comparison.

6.1 Single Skeleton Performance

The FPGA backend was first evaluated for single skeleton calls. A program was created for each skeleton type supported by the FPGA backend, with a simple, typical user function for that skeleton type. Each program was compiled and run with 3 different unroll factors: 1, 8, and 16, and all used the `float` data type. The execution time of each variation was recorded with different input sizes from 10^6 to 10^7 elements, in increments of 250000, and with 10 runs for each input size. Memory transfer time in each direction was included in the execution time. Each skeleton was invoked once before the measured invocations to remove the time to reconfigure the FPGA. Table 1 lists the skeletons and user functions that were evaluated. The skeletons were also evaluated with the OpenCL backend on both an FPGA and on a GPU. All benchmarks used a one-dimensional `Vector` as input container. The FPGA kernels were compiled using the `-fast-compile` flag, which significantly speeds up the compilation speed by reducing the compiler's optimization efforts.

6 https://intel.com/content/www/us/en/developer/tools/devcloud/overview.html

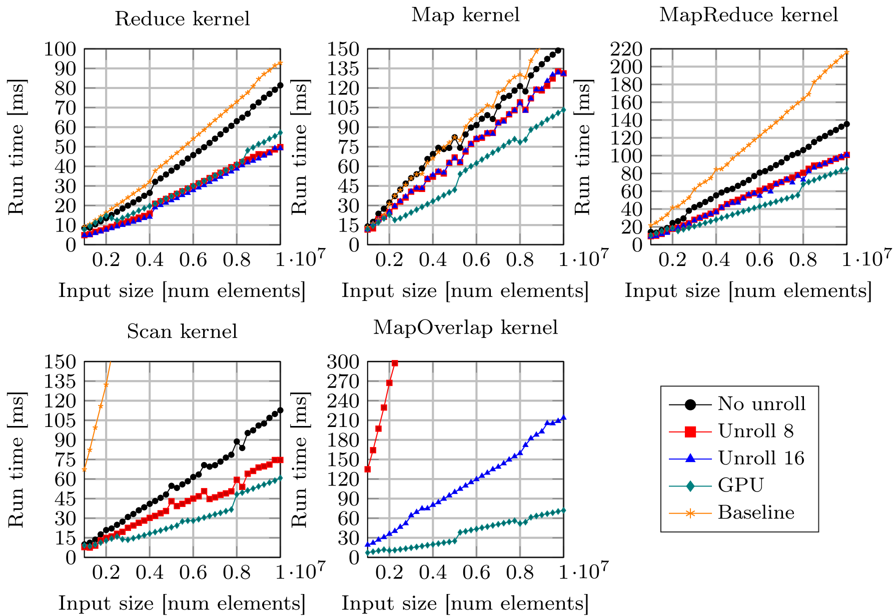

Fig. 3 The median execution times for the single skeleton evaluation (lower is better)

Figure 3 shows the performance of the single skeleton calls. The *No Unroll* line was the FPGA backend run with the unroll factor set to 1 for each skeleton, effectively disabling the unrolling, while *Unroll 8* and *Unroll 16* set the unroll factor to 8 and 16. The *Baseline* and *GPU* lines show the execution time of using kernels generated by the (preexisting) OpenCL backend where *Baseline* is the execution time on the FPGA and *GPU* is the execution time on the GPU. Scan is missing *Unroll 16* because the design generated by this unroll factor did not fit on the FPGA. It ran out of DSP blocks, which are used for float operations. For MapOverlap, a baseline is missing due to a bug in the SDK's library, preventing the benchmark from running. Finally, we omitted *No Unroll* for MapOverlap due to taking too long time: 420ms for the smallest size and 4.2s for the largest. These results show that the kernels generated by the FPGA backend are faster than the ones generated by the OpenCL backend when run on the FPGA, even with no unrolling. With unrolling, they are close to the GPU execution time and faster in the Reduce kernel case.

Figure 3 shows that the new kernel generation with FPGA-specific optimizations in the FPGA backend gives a noticeable performance gain over the OpenCL backend. The largest performance gain is observed for Scan, where the baseline implementation performed poorly. One cause of the poor performance could be the many barriers used in the baseline kernels, which force the pipeline to be flushed before moving to the next section. Map is the skeleton with the least performance gain. This is likely because the FPGA Map kernel does not use any FPGA-specific techniques besides loop unrolling, meaning both implementations are similar and are largely memory-bound. The Map

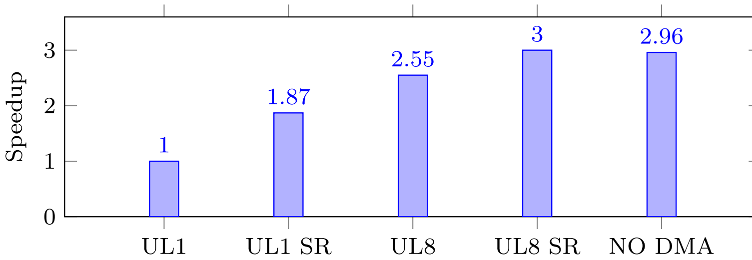

Fig. 4 Result of FPGA-specific techniques (UL*N* means an unroll factor of *N*)

kernel would likely gain from being implemented as an NDRange kernel instead, as the hardware scheduler can help alleviate memory bottlenecks.

The experiments also show that increasing the unroll factor does not improve performance past a certain point, as almost all benchmarks have similar execution times. The one outlier is the MapOverlap kernel, where the execution time for the largest input size using unroll factor 16 is 6 times faster than that with an unroll factor of 8.

Finally, we see that the fastest FPGA execution time for all but MapOverlap is close to the GPU execution time. These results are promising, given the maturity of the skeleton implementations used by the OpenCL backend.

6.2 FPGA-Specific Optimizations Performance

The effect of the two major FPGA-specific optimizations used in the skeleton implementations, loop unrolling, and shift registers, were evaluated by executing the same skeleton kernel compiled with different parameters. The Reduce skeleton was used with the same user function as in the single skeleton evaluation. Four combinations of parameters (No unrolling / Unroll by 8, and No shift register / Shift register of size 8) were used to compile four variants of the skeleton kernel.

The variant with both unrolling and shift register was evaluated once with the memory alignment and dynamic DMA transfers turned off to evaluate its impact. The total execution time was evaluated for each variant with an input size of 10^7. Each variant was run 10 times, and the median execution time was recorded. Figure 4 shows the impact of FPGA-specific optimizations. UL*N* means the kernel was compiled with an unroll factor of *N* and *SR* means it used a shift register. It shows that both loop unrolling and using a shift register gives performance benefits. The shift-register had a larger impact when not using unrolling, giving an 87% speedup, but when combined with unrolling the performance boost is only 17% compared to just using loop unrolling. Lastly, turning off the dynamic memory-transfer alignment logic had a small impact on the execution time. (NO DMA in Fig. 4 uses the UL8 SR settings and disables DMA).

The evaluation of the FPGA-specific optimizations makes it clear that they are worth implementing as they give a large speedup without the user needing to do anything. The compiler reports for the four variants show that II is 3, and the maximum frequency is 98 MHz without the shift register compared to 1 and 240 MHz with the shift register.

```
float add(float a, float b) { return a + b; }
float multiply(float a, float s) { return a * s; }
float combined(float a, float b, float s) { return (a + b) * s; }
```

Listing 6: The user functions used for evaluating multiple skeleton calls.

This is the case both with and without loop unrolling. According to Eqs. 1 and 2, decreasing the II from 3 to 1 and more than doubling the maximum frequency should give close to six times better performance. Nevertheless, we see that this does not hold in practice, as the evaluated speedup is close to half the theoretical speedup. Similarly, an unrolling factor of 8 gives a theoretical speedup of almost eight, but the evaluated speedup is far from that. Even so, the equations are still useful in guiding what parameters to change to get performance benefits.

Furthermore, the finding that turning off the memory alignment and transfer logic did not impact performance was surprising, given that Lee et al. [17] reported over 100 times faster transfer times when using this method. Profiling the kernel to get the exact memory transfer times shows that the two variants only differ by a few nanoseconds. This can mean three different things: DMA is not used in either variant, DMA is used in both variants, or DMA does not affect the memory transfer time.

6.3 Multiple Skeletons Performance

To evaluate the performance of calling multiple different skeleton instances in a single SkePU program two programs were created: One chaining two `Map` calls, and one calling a single `Map` instance explicitly fusing the two user functions, as shown in Listing 6. The total execution time of the program was recorded for an input size of 10^6 elements. Moreover, the time to reconfigure the FPGA for each skeleton call was measured using Intercept Layer for OpenCL Applications,[7] a tool for profiling OpenCL applications. Both benchmarks were run 10 times to reduce the effect of timing variations, with the median execution time and time spent reconfiguring the FPGA being recorded. The results are presented in Table 2. *Total time* is the median total amount of time spent to run the benchmark, *Reconfiguration time* is the time spent on reconfiguring the FPGA before each new kernel invocation, and *Execution time* is the time spent to run the computations, including memory allocation and transfers.

The difference in total time between the variants is stark: *Chained* spends 99% of the total time just reconfiguring the FPGA@. The reason why the *Merged* variant does not need to reconfigure the FPGA during the benchmark is that this is done when instantiating the merged skeleton, which is not part of the benchmark. This is of course also the case for other variants, but since two skeletons are instantiated, they "overwrite" each other, forcing the FPGA to be reconfigured for the second skeleton call in the benchmark every time.

[7] https://github.com/intel/opencl-intercept-layer

Table 2 Running times of the two variants evaluated

Variant	Total time (s)	FPGA Reconfiguration time (s)	Execution time (s)
Chained	7.253	7.241	0.012
Merged	0.018	0	0.018

The results of running multiple skeletons show the large overhead of reconfiguring the FPGA between each skeleton invocation[8] adds. Hence, any SkePU program that calls multiple skeleton instances will suffer large performance penalties when running on the FPGA backend, which is needed for many computations. Therefore, as many skeletons as possible should be fused when targeting the FPGA backend. In the current version of SkePU, there is no automatic fusion of skeletons, even for cases such as a chained Map and Reduce [8], so manual fusion by the user must be applied. A potential solution to this is to generate a single large kernel that only needs to be configured once, though this could lead to issues with resource usage on the FPGA if large loop unrolling factors are used or the user functions are non-trivial.

6.4 Complex User Function Performance

To test more complex user functions, such as functions with loops, we used the matrix multiplication code from SkePUs set of example programs. It uses the `Map` skeleton with the user function shown in Listing 7. Two versions were evaluated: One with an unroll factor of 1 and one with an unroll factor of 8.

For a comparison with the performance achievable on an FPGA, a handwritten Matrix Multiplication OpenCL kernel from Boyi's [14] collection of OpenCL FPGA kernels[9] was also benchmarked. The version used was the NDRange kernel with 64 as an unroll factor, a SIMD factor of 8 and 2 kernel replications (`u164_simd8_cu2`). All kernels were compiled with the same flags[10] and executed with 2048×2048 matrices. Like the previous evaluations, all kernels were run 10 times, and the median execution time was recorded.

The results in Fig. 5 clearly show that both SkePU variants run on the FPGA (*Unroll 1* and *Unroll 8*) are much slower than the handwritten variant. Furthermore, the SkePU variant with a higher unroll factor is slower than with the lower one, in contrast to our results in Sect. 6.1. The reason appears to be a failure to pipeline the *Unroll 8* variant's main loop because of the inner loop in the user function. Without the unroll, this failure does not occur.

While the results in Fig. 5 are expected, the difference in performance is an issue if the FPGA backend is used with complex user functions. The main issue is loops in the

[8] The FPGA will cache the kernel between program executions, so if the same kernel is run multiple times, only the first execution will require a reconfiguration.

[9] https://github.com/jjiantong/Boyi/tree/fpga20

[10] The handwritten kernel used an extra compilation flag (`-no-interleaving=default`) to store the two matrix buffers in different areas of the FPGA's memory. The FPGA backend does not use this optimization technique, so the flag was not used for those kernels.

```
1 T mmmult(const skepu::MatRow<int> ar, const skepu::MatCol<int> bc)
2 {
3 T res = 0;
4 for (size_t k = 0; k < ar.cols; ++k)
5 res += ar(k) * bc(k);
6 return res;
7 }
```

Listing 7: Matrix multiplication user function in SkePU.

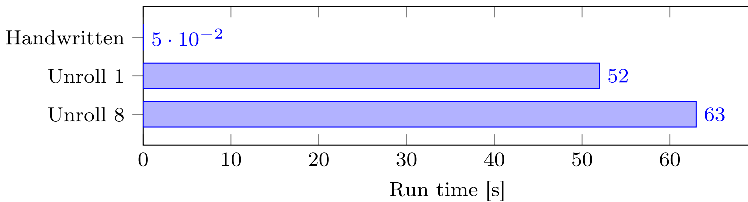

Fig. 5 Result of the complex user function evaluation

user function, as these often require modifications to not cause the outer loop's II to increase. According to Intel's documentation [13] such nested loops should preferably be fully unrolled or collapsed.[11]

7 Future Work

Future work could include a quantitative comparison with other high-level programming approaches, e.g. FPMR [22], Melia [24], OpenACC [17] or SYCL.

Our experimental setup for this work did not provide the possibility to conduct direct energy or power measurements. However, evaluation work on similar workloads has been performed in earlier work [15, 19, 21]. Kestur et al. [15] evaluate low-level FPGA implementations of BLAS kernels, which are similar workloads as in the pattern evaluations in our work. Their results show the FPGA to outperform native C, MKL, and CUDA kernels in terms of sustained energy consumption. FBLAS [19] is a high-level synthesis tool for streaming FPGA programs, and therefore closer to SkePU in terms of programming abstraction level. Their evaluation demonstrates up to 30% reduction in whole-board power consumption from using the FPGAs as compared to CPU reference implementations. While these results cannot be guaranteed to directly translate to our implementation, as the SkePU compilation and runtime system may introduce additional overhead, we consider these as guidelines or at least upper bounds on what can be expected from the FPGA backend.

[11] Automatic loop unrolling and collapsing could be added by directives to the backend compiler, but is currently not supported by the prototype.

8 Conclusion

We presented a new backend targeting reconfigurable architectures, specifically FPGAs, for the skeleton programming framework SkePU. The new backend implements many features also supported in other backends in SkePU and implements FPGA-specific optimizations for better performance. We evaluated the new FPGA backend and compared it to one of the GPU backends. Our results show that performance is close for simpler tasks and highlight the importance of using FPGA-specific optimizations. However, the backend falls behind for more complicated tasks. While the speedup by the FPGA-specific optimizations varied depending on the task, all skeletons saw performance benefits. Changing the kernels to be single work items with shift registers gave a speedup compared to running the OpenCL code by the existing OpenCL backend, from 1.06× for Map to 6.10× for Scan. Adding more optimizations improved speedup in all cases, ranging from 1.25× for Map to 9.23× for Scan.

Further details about the implementation and results can be found in the first author's recent master thesis [3]. A fork of SkePU with the FPGA backend implementation is available at https://github.com/Birath/skepu/.

Acknowledgements A.E. and C.K. acknowledge partial funding by ELLIIT, project GPAI. A.E. also acknowledges funding by Swedish National Graduate School in Computer Science (CUGS). We thank NSC for providing access to the Sigma cluster (LiU-gpu-2021-1).

Author Contributions B.B. performed implementation and evaluation work and wrote an initial manuscript. A.E., J.T., and C.K. supervised the work and edited the manuscript for publication. All authors reviewed the journal manuscript.

Funding Open access funding provided by Linköping University.

Declarations

Conflict of interest The authors declare no conflicts of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Allan, V.H., Jones, R.B., Lee, R.M., Allan, S.J.: Software pipelining. ACM Compu. Surv. (CSUR) **27**(3), 367–432 (1995)
2. Bacon, D.F., Rabbah, R., Shukla, S.: FPGA programming for the masses. Commun. ACM **56**(4), 56–63 (2013)

3. Birath, B.: Skeleton computing for reconfigurable architectures. Master thesis LIU-IDA/LITH-EX-A–23/005–SE, Department of Computer and Information Science, Linköping University, Sweden. To appear (2023)
4. Boutros, A., Betz, V.: FPGA architecture: Principles and progression. IEEE Circuits Syst. Mag. **21**(2), 4–29 (2021)
5. Cole, M.I.: Algorithmic skeletons: structured management of parallel computation. Pitman London (1989)
6. Czajkowski, T. S., Aydonat, U., Denisenko, D., Freeman, J., Kinsner, M., Neto, D., Wong, J., Yiannacouras, P., and Singh, D. P.: From OpenCL to high-performance hardware on FPGAs. In 22nd Int. Conf. on Field Programmable Logic and Applications (FPL), pp 531–534. IEEE (Aug 2012)
7. Dastgeer, U., Kessler, C.: Smart containers and skeleton programming for GPU-based systems. Int. J. Parallel Prog. **44**(3), 506–530 (2016)
8. Ernstsson, A.: Pattern-based Programming Abstractions for Heterogeneous Parallel Computing. PhD thesis, Linköping University (2022)
9. Ernstsson, A., Ahlqvist, J., Zouzoula, S., Kessler, C.: SkePU 3: Portable high-level programming of heterogeneous systems and HPC clusters. Int. J. Parallel Prog. **49**(6), 846–866 (2021)
10. Ernstsson, A., Li, L., Kessler, C.: SkePU 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. Int. J. Parallel Prog. **46**(1), 62–80 (2018)
11. González-Vélez, H., Leyton, M.: A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. Softw.: Pract. Exper. **40**(12), 1135–1160 (2010)
12. Hill, K., Craciun, S., George, A., and Lam, H.: Comparative analysis of OpenCL vs. HDL with image-processing kernels on Stratix-V FPGA. In 26th Int. Conf. on Application-specific Systems, Architectures and Processors (ASAP), pp 189–193 (Jul 2015)
13. Intel. Intel FPGA SDK for OpenCL Pro Edition: Programming Guide
14. Jiang, J., Wang, Z., Liu, X., Gómez-Luna, J., Guan, N., Deng, Q., Zhang, W., Mutlu, O.: Boyi: A systematic framework for automatically deciding the right execution model of OpenCL applications on FPGAs. In: Proc. Int. Symposium on Field-Programmable Gate Arrays, FPGA'20, pp. 299–309. ACM (2020)
15. Kestur, S., Davis, J. D., and Williams, O.: BLAS comparison on FPGA, CPU and GPU. In 2010 IEEE Computer Society Annual Symposium on VLSI, pp 288–293 (2010)
16. Lambert, J., Lee, S., Kim, J., Vetter, J. S., and Malony, A. D.: Directive-based, high-level programming and optimizations for high-performance computing with FPGAs. In Proc. Int. Conf. on Supercomputing, ICS'18, pp 160-171. ACM (2018)
17. Lee, S., Kim, J., and Vetter, J. S.: OpenACC to FPGA: A framework for directive-based high-performance reconfigurable computing. In Int. Parallel and Distrib. Processing Symposium (IPDPS), pp 544–554 (2016)
18. Lee, S., and Vetter, J. S.: OpenARC: open accelerator research compiler for directive-based, efficient heterogeneous computing. In Proc. int. symp. on high-performance parallel and distributed computing, HPDC '14, pp 115–120. ACM (2014)
19. Matteis, T. D., Licht, J. d. F., and Hoefler, T.: FBLAS: Streaming linear algebra on FPGA. In SC20: International conference for high performance computing, networking, storage and analysis, pp 1–13 (2020)
20. Nane, R., Sima, V.-M., Pilato, C., Choi, J., Fort, B., Canis, A., Chen, Y.T., Hsiao, H., Brown, S., Ferrandi, F., Anderson, J., Bertels, K.: A survey and evaluation of FPGA high-level synthesis tools. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **35**(10), 1591–1604 (2016)
21. Qasaimeh, M., Denolf, K., Lo, J., Vissers, K., Zambreno, J., and Jones, P. H.: Comparing energy efficiency of CPU, GPU and FPGA implementations for vision kernels. In 2019 IEEE Int. Conf. on Embedded Software and Systems (ICESS), pp 1–8 (2019)
22. Shan, Y., Wang, B., Yan, J., Wang, Y., Xu, N., and Yang, H.: FPMR: MapReduce framework on FPGA. In Proc. int. symp. on field programmable gate arrays, FPGA '10, pp 93–102. ACM (2010)
23. Wang, Z., He, B., Zhang, W., and Jiang, S.: A performance analysis framework for optimizing OpenCL applications on FPGAs. In 2016 IEEE international symposium on high performance computer architecture (HPCA), pp 114–125 (2016)
24. Wang, Z., Zhang, S., He, B., Zhang, W.: Melia: a mapreduce framework on OpenCL-based FPGAs. IEEE Trans. Par. Distr. Syst. **27**(12), 3547–3560 (2016)
25. Zohouri, H. R.: High performance computing with FPGAs and OpenCL. PhD thesis, Tokyo Institute of Technology (2018)

26. Zohouri, H. R., Maruyama, N., Smith, A., Matsuda, M., and Matsuoka, S.: Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs. In Proc. SC16 Conference, pp 409–420. IEEE (2016)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.