The ORKA-HPC Compiler — Practical OpenMP for FPGAs

Florian Mayer,[1] Julian Brandner,[1] Matthias Hellmann,[2] Jesko Schwarzer,[3] and Michael Philippsen[1]

[1] Friedrich-Alexander University Erlangen-Nürnberg (FAU)
Programming Systems Group
{florian.andrefranc.mayer,julian.brandner,michael.philippsen}@fau.de
[2] University of Cologne, Regional Computing Centre (RRZK)
hellmann@uni-koeln.de
[3] Systemberatung Schwarzer, orka-hpc@schwarzers.de

Abstract. ORKA-HPC is a new and downloadable OpenMP-to-FPGA compiler that is easy to set up, easy to use, and easy to extend. It targets a variety of different FPGA-boards, and is distributed with a "batteries included" runtime and development environment.
Starting from a set of properties that such a compiler must possess, we derive how ORKA-HPC achieves these, reason about the underlying decisions, and evaluate ORKA-HPC's current state of development. The paper concludes with future work and provides a download link.

1 Motivation

While some research groups attempted to build OpenMP-to-FPGA compilers [10], none of them made it into wide-spread use. Without commercial interest by industry, research funding often only suffices for building prototypes that address the posed research question but does not make the project's software valuable for other research groups. Even if there is open source code, it is hardly useful once the PhD students have left or the funding has ended. The ORKA-HPC OpenMP-to-FPGA compiler tries to avoid this fate by avoiding the following common mistakes and showstoppers.

1) Cross-platform Issues. While mainstream programs are a part of a Linux distribution and available as pre-built binaries, research software in general does not make it into Linux distributions. Hence, there is only the source code plus instructions on how to compile and build. Interested parties thus have to ensure every build-time dependency, often by having to install specific versions of other tools or libraries. As sometimes the authors did not properly document all the dependencies, this makes it even harder to achieve a successful installation and thus to reproduce published results. In contrast, ORKA-HPC is easy to install on different Linux platforms and avoids dependency issues.

2) Shifting Grounds Issues. Where well-funded Linux distributions provide a stable execution environment with fixed APIs and libraries for a long time, others change more rapidly. Projects that are under active development can adapt their build- and run-time dependencies to such changes. However, once developers have left a project, their code often stops working after a short while. ORKA-HPC employs techniques to enhance its lifetime.

In Sec. 4 we discuss how ORKA-HPC achieves these general goals (portable cross-platform distribution and longevity). In addition, there are goals for FPGA projects and for compiler projects.

3) Stand on the shoulders of giants. To use an FPGA one needs to synthesize a Bitstream that describes the FPGA hardware. In general, special compilers (1) produce a tree representation for a C/C++ input, (2) generate an intermediate representation (IR), (3) optimize it, (4) generate a VHDL/Verilog hardware description from it, and (5) use a VHDL compiler to produce the Bitstream. Since FPGA vendors typically keep their hardware architectures undisclosed, little can be done in step 5 (hardware synthesis). In the past, researchers often reinvented all or some of the steps 1–4 (High-Level Synthesis, HLS). However, they were in competition with the vendors' industry-grade HLS tools and their developers. The ORKA-HPC compiler only builds crucial components from scratch and makes use of the best performing industry tools for both HLS and hardware synthesis.

4) Do not put all the eggs in one basket. Published OpenMP-to-FPGA compilers often generate FPGA hardware for one specific FPGA board. However, there are two issues here. First, different FPGA architectures are good for different computing tasks. Second, specific FPGA boards quickly become outdated. Hence, the ORKA-HPC compiler is extensible/portable for new devices and benefits from future advances of FPGA hardware. For the same reason the ORKA-HPC compiler is agnostic with respect to the board vendors. While APIs, options for both the HLS and the hardware synthesis, pipeline feedback, etc. are often at least upwards compatible for all the boards of a specific vendor, ORKA-HPC needs extra layers to gloss over differences between vendor tools.

Finally, there are goals for any research compiler, independent of targeting FPGAs. It must must be **(5) easy to use** ideally as a drop-in replacement for Clang/GCC without modifications to the build system and it must be **(6) easy to extend** with regards to its internals. It is written in a way that it minimizes the effort to change the internals or to add new analysis/optimization algorithms.

In Sec. 3 we show how ORKA-HPC achieves these goals (reuse of optimized components as building blocks, retargetability to various FPGA types and brands, ease of use, and extensibility). Sec. 5 reports on some experiments to demonstrate that ORKA-HPC is operational.

2 Related Work

We discuss works mentioned in the survey by Mayer et al. [10], but exclude approaches that either require extensive hardware development knowledge (e.g. [2]), or that do not focus on offloading the OpenMP `target` pragma to the FPGA (e.g. [15]) but map other constructs.

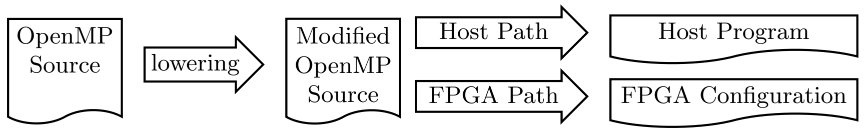

Fig. 1. Bird's eye view of ORKA-HPC's compilation process.

Bosch et al. [1] conduct target offloading for a programming model that is partially derived from OpenMP. They generate hardware from an annotation scheme closely resembling OpenMP's task based parallelism.

Sommer et al. [16] extract target regions and pass them to the TaPaSCo LLP backend. While being conceptually similar to ORKA-HPC, the tool is restricted to said backend and thereby inherits its platform dependencies.

In contrast to the above mentioned works (including ORKA-HPC) Knaust et al. [7] achieve offloading by passing an IR representation of the input region to a vendor tool. While providing a remarkably simple workflow, the method comes with a significant drawback, as it relies on an undocumented interface of one specific vendor tool.

Huthmann et al. [5] also use IR-level representations for Bitstream generation. But instead of passing them to commercially available tools their system explores the possibilities of using an open academic HLS infrastructure for target offloading. In contrast, ORKA-HPC exploits highly optimized vendor tools.

Nepomuneco et al. [12] focus on multi-FPGA environments and evaluate their speedups for just one specific accelerator platform.

To the best of our knowledge, none of the published systems gives details on how to achieve portability across Linux platforms, how to achieve longevity, how to exploit optimized tools as building blocks, how to target various brands and types of FPGA boards, and how to achieve ease of use and extensibility. ORKA-HPC tries to stay as accessible as your command line operated C compiler.

3 The ORKA-HPC OpenMP-to-FPGA Compiler

Here we introduce the building blocks of the ORKA-HPC compiler. We discuss the general architecture, the design options, and the reasons for ORKA-HPC's choices to achieve the goals above. We discuss distribution issues in Sec. 4.

The bird's eye view of the ORKA-HPC compiler in Fig. 1 shows its main processing paths and intermediate results. After lowering the OpenMP pragmas in the input source down to an intermediate form at a lower level of abstraction, the compiler then processes the modified OpenMP code by two distinct tool chains: An FPGA path offloads work to one or more FPGA devices and a host path generates the main program that manages the distribution of work.

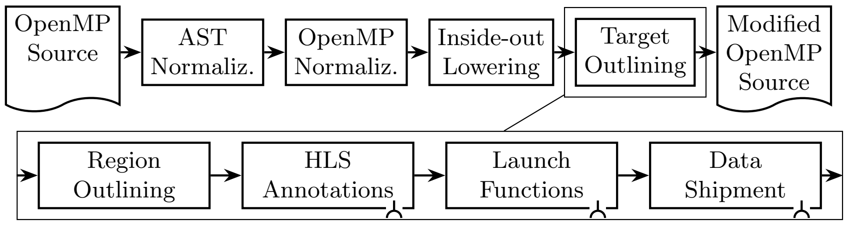

Fig. 2. AST-level OpenMP lowering. Hooks to plug in vendor-specific modules.

3.1 OpenMP lowering

Which pragmas to offload to the FPGA? When building an OpenMP-to-FPGA compiler one has to decide which OpenMP pragmas to leave unchanged and to forward to a regular OpenMP compiler for a (shared memory) CPU (i.e., the host path), and which pragmas to transform so that the affected source code turns into FPGA hardware. By leaving pragmas for the host path that were designed for shared-memory systems with operating systems that provide the necessary synchronization primitives one can tap the general purpose compiler's optimization capabilities. There have been attempts to map OpenMP's synchronization primitives and shared-memory behavior to FPGAs, but they never matured beyond the proof-of-concept state, probably because there is a too wide gap between the fundamentally different computation models of FPGAs and CPUs. For instance, FPGAs neither permit easy thread synchronization nor access to the same shared memory of the CPU. For these reasons, ORKA-HPC only offloads the OpenMP `target` and `target data` pragmas to the FPGA and lowers pragmas that are nested inside the regions that both pragmas annotate.

How to Lower? In general, compilers (1) convert their input to an Abstract Syntax Tree (AST), (2) transform this AST, (3) map/lower the AST to a simpler intermediate representation (IR), and (4) generate assembly from it. Adding OpenMP support to an existing compiler requires additional AST node types and extensions to the steps 1, 2, and maybe 3. Compiler writers have two choices. First, they can replace all new OpenMP nodes with bundles of regular AST nodes. This so-called AST-level lowering only affects steps 1 and 2. Second, they can translate OpenMP nodes to intermediate form, just like all other AST nodes. This so-called AST-to-IR lowering also requires work on step 3.

ORKA-HPC uses an AST-level lowering and here is why: Offloading must transform a pragma into calls of runtime library functions that control the FPGA. On AST-level this is as simple as adding a function call node to the AST. After the transformation, the AST still represents regular C/C++ and can – after some pretty-printing – be fed into any vendor's HLS systems.

In contrast, AST-to-IR lowering has three problems. First, it needs the HLS system to be be usable on IR-level but until lately [7], Intel's or Xilinx's HLS systems did only accept C/C++ inputs. Of course, the lowering could generate

IR, optimize it, and convert it back to C/C++, but it is hard to decompile IR which is why LLVM for example has removed support for it in an early version [9]. Thus, OpenMP Fortran cannot yet be run on FPGAs, as this either requires IR-capable HLS systems or IR decompilers. Today, the situation seems to be a bit better as Vivado's HLS is now open source. Second, it requires an OpenMP runtime library in IR format (a set of IR-level routines) that have to be created somehow. Third, the AST-to-IR approach potentially inherits implementation details of vendor tool chains (e.g., the LLVM IR version that the tools support) which makes it hard to support multiple vendors and their FPGA boards. [21].

Lowering Details. Fig. 2 zooms into the first arrow in Fig. 1. To make an OpenMP-to-FPGA compiler easy to extend, both normalization steps are mandatory. Moreover, nested pragmas need to be lowered inside-out. Finally, to benefit from proven compiler tooling, existing generic outliners can be adjusted to work with separate memory domains. The following paragraphs explain the details.

AST Normalization. When an AST is first constructed from an input program, it often reflects the syntactic sugar that the programmer has used. For example, there are `if`-statements with a missing `else` branch or with an empty block; there are variable declarations with or without an initialization to the default value; there are `for` statements with a single statement in their bodies (`for() s();`) or with a block that holds the same statement (`for(){s();}`). Even though the semantics are the same, the AST representations differ and the compiler would need different cases in all its steps. It is thus common practice to normalize the AST before processing it. For example, branches of an `if` can always be turned into blocks of statements, variable declarations can always have an explicit initialization, etc. This cuts down on the number of cases in all compiler steps downstream and makes them easier to extend. The ORKA-HPC compiler follows this common practice and first reduces the complexity of the input AST.

```
#pragma omp target
#pragma omp teams
#pragma omp parallel
#pragma omp for
// equal to
#pragma omp target teams
#pragma omp parallel
#pragma omp for
// equal to
#pragma omp target teams
#pragma omp parallel for
```

Fig. 3. Combined pragmas.

OpenMP Normalization. OpenMP Normalization simplifies the AST w.r.t. OpenMP pragmas and makes an OpenMP-to-FPGA compiler easier to extend. The idea again is to normalize the input to a single representation that has all the default values set explicitly to avoid special cases for syntactic variants that have the same semantics.

In OpenMP several pragmas may or may not be combined. An example is `target teams parallel for` [14]. As illustrated in Fig. 3, the same can also be expressed in four lines with one pragma each, or in fewer code lines that hold two or more of these four pragmas. In essence, the combined form is a concatenation of the pragmas retaining their original order. The ORKA-HPC compiler brings all pragmas into an expanded normal form according to the rules of OpenMP. This yields fewer cases and reduces the complexity of writing the compiler and hence make it easier to extend.

Similar to a variable declaration that receives an explicit initialization to a default value when the programmer has not specified a different value, OpenMP normalization also adds default values to the AST nodes. Defaults are specified for most optional OpenMP clauses. Later on, the compiler can lower an AST node without knowing about default values. This separation of concerns is common practice applied to OpenMP translation.

```
// original code:
int a[];
{ /* code that uses a */ }
// after outlining:
int a[];
out(&a);
void out(int **a) {
  /* code that uses *a */
}
```

Fig. 4. Standard Outlining.

Target Outlining. An OpenMP-to-FPGA compiler must partition the original program to offload parts of it to an FPGA. C-based HLS systems only accept entire C functions as input [6,19] and most HLS systems require these functions to be annotated with vendor specific pragmas or attributes. Moreover, the CPU code and the FPGA do not share memory.

Partitioning of programs for offloading is not new. OpenMP offloading for GPUs uses so-called outlining that wraps the piece of code under the `target` pragma in a new function and replaces the original region with a call of this function. ORKA-HPC strives to not reinvent the wheel. Instead of starting from a GPU outliner, as they are tailored to the demands of GPU vendor tool chains, it starts from a generic AST-level outliner that, however, is meant for a shared-memory situation. Fig. 4 illustrates what a generic outliner would do. All the data that an outlined block of code accesses is passed to the outlined function as a set of pointers, i.e., with one level of indirection. This works well as long as caller and callee can access the same shared memory, but on an FPGA pointers to data in the host memory are meaningless, the HLS synthesis in general cannot handle them.

The ORKA-HPC compiler therefore extends a generic outliner with an address space cleanup that replaces infeasible pointer indirections with API calls that access the data in the other address space, depending on what the programmer has specified in the map clauses. See *Data Shipment* in Fig. 2.

Since the outlined function cannot simply be called after the synthesis turned it into FPGA hardware, the ORKA-HPC outliner inserts API calls to *ship data* to and from the FPGA and to *launch* the function's hardware. To achieve portability across different board types and vendors, the launch and the shipment APIs are generic. Most HLS systems require function annotations or special pragmas to produce valid and well-performing hardware. Despite the vendor-specific demands, the ORKA-HPC compiler still achieves portability by bundling the insertion of these annotations into another pluggable module, specific for each vendor but hidden behind a generic *annotation* API.

Inside-Out Lowering. So far we assumed that there was no nested pragma in a `target` region. To limit the engineering effort and to make the ORKA-HPC compiler easy to extend for new pragmas, it is best to process nested pragmas inside-out. At the deepest nesting level, a pragma only affects a region of plain, pragma-free code. At the deepest level, a pragma can be lowered without special cases for enclosed pragmas because there are none. To understand that such special cases are problematic, think of a target region that uses the variable `a`. If there can still be inner pragmas like for instance `parallel shared(a)`, the target outlining would need special cases for different flavors of variables. By lowering the pragmas inside-out, compiler writers only need to reason about one lowering template per pragma. Such a template replaces the code block affected by the pragma with a pragma-free code block. In the example, the `shared(a)` is long gone, when the outliner works on the `target` region. Inside-out lowering also eases experiments with research pragmas, as they can be lowered in isolation (as any potentially nested pragmas will already have been transformed away). Note that by climbing up the nesting levels of pragmas, the generated replacement code may be modified repeatedly, once per pragma layer.

To see why inside-out lowering works for non-trivial pragma situations, consider that in general a pragma only affects the statement or the scope directly below it. Inside-out lowering cannot handle clauses that affect a preceding line, i.e., a parent note in the AST. We do not know of such pragmas or clauses but could handle them with a pre-processing pass that adds attributes to the AST before the inside-out transformation.

The compiler needs to reject pragma combinations that cannot be mapped to an FPGA (e.g. there cannot be a `target` pragma inside another `target` pragma). While this check is straightforward to implement as part of the inside-out transformation, an outside-in approach would require a full subtree traversal that queries the structure of the pragmas nested below.

3.2 FPGA Path

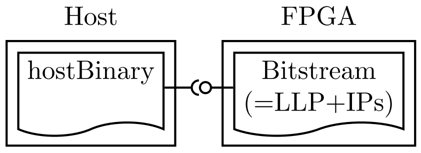

Fig. 5. LLP API.

According to Fig. 1, the OpenMP-to-FPGA compiler produces two artifacts geared to seamlessly work together. The FPGA must be configured to perform the offloaded work and the host program must make use of the FPGA, see Fig. 5. An FPGA is configured with a so-called Bitstream, i.e., a binary file that represents all the wires and logic gates of the circuit. In addition to these circuits that represent the offloaded function (called IP in FPGA lingo), it is common practice to capture infrastructure hardware in a so-called Low Level Platform (LLP). The LLP is used to interface with the host, e.g., via a PCIe interface, and it also contains the bus system to communicate with or among offloaded functions.

It is infeasible to generate these Bitstreams from scratch in the FPGA path, because the format of the Bitstream is proprietary. FPGA vendors typically provide hardware synthesis tools that offer many pre-built hardware blocks (e.g., for PCIe communication) and thus ease the development process. Such a tool is

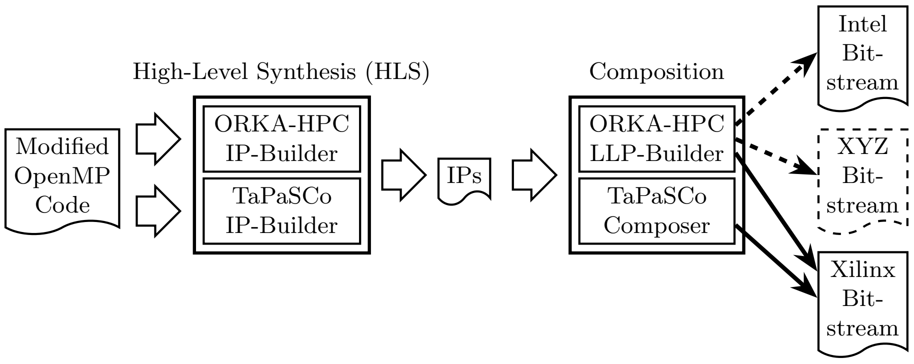

Fig. 6. FPGA Path. Dashed means ongoing work.

still impractical to use from a compiler, because it usually interprets a kind of abstract recipe to generate the Bitstream. For that reason LLP backends such as in ORKA-HPC or TaPaSCo [8] have been developed. As shown in Fig. 6, they get a set of C functions, build standalone hardware entities (IPs) for each of them, and write recipes that build an LLP plus IPs to form a fully functional FPGA. They also provide an LLP API to launch the execution of the work in the Bitstream from the host side, plus the necessary data shipment routines.

As these LLP backends are still mostly vendor specific, ORKA-HPC generalizes LLP functionality so that users can specify which LLP backend to use per `target` pragma. Different `target` regions can thus be offloaded to different FPGAs of a system. New LLP backends can also be added easily.

3.3 ORKA-HPC LLP-Backend

The ORKA-HPC LLP-backend provides LLPs for FPGA boards. In contrast to other such backends (like TaPaSCo), it supports multiple FPGA vendors. It consists of multiple parts: The IP-builder (see Fig. 6) builds IPs starting from the C-codes emitted by the ORKA-HPC compiler. The LLP-builder integrates these IPs and generates Bitstreams for the targeted hardware. For host-to-FPGA communication it also includes an abstraction layer and C API.

Tailored to the functionality and performance necessary for the ORKA-HPC compiler, the LLP-backend provides the tools necessary to generate Bitstreams and run OpenMP on FPGAs without any user input. While easily extendable in capability and fully configurable by the user, predefined and hand-optimized LLPs are provided for ease of use. With *partial reconfiguration* support, it allows for even more logic to be deployed on the FPGAs (sequentially) and reduces compile times. Its support of FPGAs from various vendors enables ORKA-HPC to distribute OpenMP workloads to a wide range of hardware configurations using one or multiple communication interfaces, including PCIe, Ethernet and USB concurrently – a unique feature of ORKA-HPC not available in any other tool. In order to exploit the full potential of the FPGAs it internally uses the vendor's tool chains to generate IPs and Bitstreams. Since vendors typically update their tools alongside new hardware releases, this also facilitates adding compatibility to future devices and helps extend ORKA-HPC's lifetime.

The ORKA-HPC Generic Driver encapsulates device drivers and exposes a unified and vendor independent API for host-to-FPGA communication. FPGA selection can thus be done at runtime *on the fly* without the need to recompile library calls in the host binary, no matter which of the supported boards (brands) are targeted and how they are connected (e.g., locally via PCIe or *remotely* via TCP/IP). The *remote*-option is in fact a driver virtualization that enables driving FPGA boards through the internet or cloud solutions.

3.4 Host Path

We have discussed above that as part of the outlining, the ORKA-HPC compiler adds library calls to the host program that launch the offloaded functionality on the FPGA and that perform the data shipment to and from the FPGA. Because of its design goal to be useful for various types and brands of FPGAs, ORKA-HPC decouples the host binary both statically and dynamically from the various FPGA-specific LLPs. While decoupling is well known, to the best of our knowledge ORKA-HPC is the first tool that employs it that way.

For a static decoupling ORKA-HPC uses a generalized LLP API when lowering a `target` region. Adding a new type of FPGA to the ecosystem then only takes some glue code between the generalized LLP API and the concrete API.

For a dynamic decoupling ORKA-HPC loads concrete LLPs as plugins (at load-time). This allows to select an FPGA board per `target` region and it also allows to bind to a different LLP without recompiling the host code.

One particularly helpful LLP plugin in the ORKA-HPC environment is the Dummy LLP plugin. It only pretends to talk to an FPGA. In reality it executes every data shipment and IP control request locally and emulates the behaviour of the offloaded region on an FPGA. This eases and expedites the debugging of control messages as it saves the slow synthesis of FPGA hardware and drastically reduces the round-trip time (from hours to seconds).

ORKA-HPC therefore supports the use of a mixture of different LLPs in the *same* program and enables the host binary to switch between different LLPs, including the Dummy LLP, without recompilation. It is the latter feature that makes the ORKA-HPC compiler useful for LLP researchers as they can easily plug in experimental versions of their codes. To demonstrate the extensibility of the ORKA-HPC solution, we plugged in the TaPaSCo Composer [8] in addition to the ORKA-HPC LLP backend.

4 Deployment

For easy installation of a compiler on a variety of platforms, both the compiler and the build process for the compiler must run on different platforms. An OpenMP-to-FPGA compiler is harder to get up and running than a regular C compiler as the former in general relies on specific versions of other components (e.g., LLP backend, FPGA tool chain, C compiler, etc.).

Well-known challenges of a portable build process are to manage the build-time dependencies and to ensure them on other systems. There are popular solutions that unfortunately limit the ease of using ORKA-HPC. First, using a Virtual Machine to distribute a piece of software is equivalent to shipping a 1:1 image of the developer's computer. The disadvantages are: The VM approach only works as long as the VM can be executed. There is almost no extensibility. In general, the size of such a VM is enormous (several gigabytes). Above all, team development is severely restricted, since a change of the VM requires a copy of the entire VM to be shipped to other team members. Second, dependency managers such as Nix [13] add complexities for both the developers and the build-process. Third, custom shell scripts that tailor each build environment to fit a given Linux distribution take effort to write and are hard to maintain.

ORKA-HPC avoids the above disadvantages by using Docker [3] for deployment. Instead of having to ship a 1:1 image, the ORKA-HPC compiler provides a small recipe, the so-called Dockerfile, that describes the Docker containers that act like Virtual Machines in which the specified environment for build- and run-time are set up correctly. Although we recommend the Docker tools to build the ORKA-HPC environment automatically from the Dockerfile, users can also simply follow those human-readable instructions to build ORKA-HPC themselves. Compiled programs also do not need the Docker container.

5 Evaluation

target
target data
target teams$\star$
target distribute$\star$
target teams distribute$\star$
target parallel$\star$
target parallel for$\star$
target teams distribute simd$\star$
target update
target enter data$\diamond$
target exit data$\diamond$
declare target
end declare target

Fig. 7. Supported OpenMP.

In this section we report on the current state of an ongoing project. We list the OpenMP pragmas that ORKA-HPC currently supports, give performance numbers on an embarrassingly parallel Mandelbrot experiment, and discuss the current functional completeness on a set of benchmarks that was never designed to be offloaded to FPGAs.

Pragma Coverage. Fig. 7 lists all the pragmas that ORKA-HPC currently supports for FPGA offloading. ORKA-HPC only covers `target` pragmas or pragmas nested below them. The underlying generic OpenMP compiler deals with all other pragma situations. There are three caveats to Fig. 7. We correctly parse the combined `target` constructs marked with $\star$, generate AST nodes, and normalize them; we do not yet exploit their semantics to generate efficient parallel FPGA structures, i.e., there is not yet an automatic mapping for instance of a `parallel for` to the HLS directives that would cause parallel FPGA hardware.

Table 1. Mandelbrot performance. Runtimes are given in milliseconds. The Xilinx VCU118 has the following resources available: $FF_{avail} = 2,364,480$. $LUT_{avail} = 1,182,240$. $DSP_{avail} = 6,840$. $BRAM_{avail} = 2,160$.

Unroll factor	1			8	32	64	128
Frequency	100MHz	200MHz	400MHz	400MHz	400MHz	350MHz	50MHz
FF_{all}	6.09%	6.09%	6.09%	7.23%	11.06%	16.16%	26.36%
LUT_{all}	9.49%	9.49%	9.53%	10.44%	13.25%	17.11%	24.95%
DSP_{all}	0.57%	0.57%	0.57%	4.25%	16.89%	33.73%	67.41%
$BRAM_{all}$	5.19%	5.19%	5.19%	5.19%	5.19%	5.19%	5.19%
FF_{mandel}	5.3%	5.3%	5.3%	20.1%	47.8%	64.3%	78.1%
LUT_{mandel}	3.0%	3.0%	3.3%	11.7%	31.5%	47.3%	63.7%
DSP_{mandel}	92.3%	92.3%	92.3%	99.0%	99.7%	99.9%	99.9%
$BRAM_{mandel}$	2.2%	2.2%	2.2%	2.2%	2.2%	2.2%	2.2%
FPGA runtime	2592.0	1296.0	0648.0	0109.6	0033.4	**0023.0**	0102.8
transf. to/alloc	0000.8	0000.8	0000.8	0000.8	0000.8	0000.8	0000.8
transf. from	0003.6	0003.6	0003.6	0003.5	0003.6	0003.6	0003.6

For the pragmas marked with ◇ we do not yet emit the library routines. Finally, all translation units must be compiled with ORKA-HPC before linking. Kernels and library code compiled with other compilers cannot be called in target regions.

Mandelbrot Performance. We wrote a Mandelbrot code with an explicit loop unroll parameter to study the performance of the FPGAs that ORKA-HPC generates. Per unrolled loop iteration there is a sub-block of FPGA hardware that implements the loop body for a parallel execution. Our code first transfers *all* data to the FPGA, triggers the Mandelbrot calculation, and finally transfers back *all* the data. The first four rows of Table 1 show the overall utilization of our VCU118 FPGA board, according to the Vivado Bitstream synthesis report. The columns hold various unroll factors and FPGA clock frequencies. The rows report on the fraction of FlipFlop (FF), Look-up-tables (LUT), Digital Signal Processor (DSP) blocks, and BRAM modules of the full board that our generated IPs populate.[4] There are four rows that show the resource consumption for *all* components of the Bitstream, i.e., the LLP and the Mandelbrot IP. The *mandel* rows tell how much of those resources the Mandelbrot IP uses. The upper shaded area shows the significant fixed cost of the LLP plus one Mandelbrot IP. Unrolled versions of this IP amortize the LLP cost. The lower shaded area holds the cost of one Mandelbrot IP. There again are fixed costs plus variable unrolling costs for FF, LUT, etc. The variable costs have different growth rates as can be seen in the columns of the higher unroll factors.

The last three rows show our runtime performance measurements for each Bitstream. Here, we show the pure FPGA runtime, "transf. to/alloc" gives the

[4] A LUT can be configured to behave like an arbitrary n-to-1 logic function and FFs are usually grouped to resemble n-bit registers. Most high-end FPGAs provide DSP blocks as ASIC components to accelerate floating-point heavy tasks.

Table 2. Mandelbrot performance on CPUs.

Processor	Xeon		i7	
Num. Threads	1	6	1	6
Runtime (secs.)	0.02451	**0.01872**	0.01184	**0.00634**
Std. Deviation	0.00152	0.00376	0.00185	0.00096
Flags	-O3	-O3 -fopenmp	-O3	-O3 -fopenmp
GCC Version	7.5.0		7.5.0	

Xeon = Intel(R) Xeon(R) Gold 5220 CPU (2.20GHz, 72 log. cores).
i7 = Intel(R) Core(TM) i7-4770 CPU (3,40GHz, 8 log. cores).

time needed for the data shipment plus the allocation, and "transf. from" is the duration for the data to return. Shipping the full data to/from the FPGA is neither affected by the unroll factor nor by the FPGA's clock frequency. All runtime durations were calculated by timing measurement routines that ORKA-HPC automatically placed into the host binary. All durations given are the averages of ten independent runs of the same program on one Intel(R) Core(TM) i7-4770 CPU (clocked at 3.40GHz, consisting of 8 logical cores). The standard deviations of all durations range from $1.08 \cdot 10^{-6}$ (min) and $4.49 \cdot 10^{-5}$ (max).

For the first three columns, we left the unroll factor fixed at 1 (i.e., one Mandelbrot IP) but varied the IP frequencies from 100MHz to 400MHz. As expected, the FPGA utilization did not change much, but the execution speed doubled each time we doubled the frequency. Our code did not meet hardware timing requirements for frequencies above 400MHz. The next four columns show higher unroll factors and the maximal possible frequency (in multiples of 50MHz). For instance, when the unroll factor is 8, our offloaded function fetches eight pixels from the memory of the VCU118 via a DMA transfer, processes each pixel in parallel, and writes 8 pixels back into the FPGA memory (also via DMA). The best execution speed for an unroll factor of 64 and a clock frequency of 350MHz is in **bold** in Table 1. Unfortunately, for an unroll factor of 128, the code did not compile beyond 50MHz due to hardware timing issues.

Table 2 shows the performance of the same Mandelbrot code on off-the-shelves computers without FPGAs. Since generic C compilers ignore the HLS pragmas, we added a `parallel for` pragma to the unrollable loop so that is can use six cores of our hardware. Our FPGA runtimes in Table 1 are about on par.

Multi-Board Support. The Mandelbrot code runs on multiple boards. We can configure ORKA-HPC to pick the ORKA-HPC LLP backend for the Arty board [11]. For the Vivado VCU118 [20] board, we can pick both the ORKA-HPC and the TaPaSCo LLP backend. Data transfers to/from the Arty board happen via Ethernet. The communication with the VCU118 uses the PCIe interface.

SPEC ACCEL. We used ORKA-HPC with the seven unmodified SPEC ACCEL benchmarks [17] that employ target offloading. Only for 514.pomriq we had to manually expand `#include "computeQ.c"` to fix a pre-processor issue. The

Table 3. Functional verification with SPEC ACCEL.

Benchmark Stage	503. postencil	504. polbm	514. pomriq	552. pep	554. pcg	557. pcsp	570. pbt
Frontend	✓	✓	✓	✓	✓	-[1]	-[1]
Mock with Dummy LLP	✓	✓	✓	✓	✓		
HLS for x functions	1✓	2✓of 3[2]	2✓	5✓	13✓of 16[3]		
FPGA	✓		✓	✓			
Time for Mock	6.13s	18.26s	4.23s	11.71s	19.91s		
Time for HLS	55.21s	119.81s	102.56s	314.21s	967.87s		
Time for FPGA	7065.63s		6928.39s	9836.89s			

[1] Uses the `[:]`-Syntax in a map clause which we do not yet support.
[2] One offloaded function used a pointer cast that the HLS does not support.
[3] HLS aborts with unknown error for three offloaded functions.

benchmarks are not meant to be used with FPGA offloading as they use `floats` or `doubles` which is not among the strength of FPGAs as the HLS synthesis of standard C floating point data types takes up considerable amounts of resources [4]. This is why this section is only a report on the functionality of ORKA-HPC, not on the performance gained.

Table 3 is a work-in-progress report. Its upper part illustrates the current state of the ORKA-HPC pipeline at the time of writing this paper. For each of the seven benchmarks it shows which stage of the compilation it successfully passes. The Frontend stage includes all ORKA-HPC steps that transform the input code into a host binary that can interact with an FPGA via different LLP backends. The footnotes show the current limitations. Codes that pass the Mock stage run with the Dummy LLP plugin that simulates an FPGA. They produce the same results as a host-only compilation without `-fopenmp`. A check mark in the HLS row tells that we can successfully translate the offloaded functions to hardware blocks. For two benchmarks there are issues in Vivado HLS 2018.2 that prevent synthesis for 1 or 3 functions. A newer version of Vivado's HLS fixes some of these bugs, but at the time of writing we still have to migrate to it. The FPGA row reports on a successful generation of the full Bitstream.

In the lower part of Table 3 we show how long the pipeline phases take (wall clock time) on a Intel(R) Xeon(R) Gold 5220 CPU (clocked at 2.20 GHz, consisting of 72 logical cores) with 496 GB of available RAM. As we are actively working on the benchmarks, the table will have more ✓ in the final version.

At the time being, the benchmarks that run with FPGAs are not as fast as CPU-only versions. (And because of the floating point computations they cannot be expected to.) The runtimes can be off by a factor of up to 900. We identified the top two reasons. First, we currently neither automatically generate an explicit unrolling nor the pragmas that would tell the HLS where to use parallel hardware. The Mandelbrot experiment shows the importance of this.

Second, the HLS generates an independent memory request to an off-chip RAM for *every* array access, without any caching. Adding suitable HLS pragmas for this is also in our future work.

6 Contributions and Future Work

We contribute ORKA-HPC, a new OpenMP compiler for FPGAs that has several novel properties. First, it is portable across different Linux platforms and uses Docker to ease the build process (of the compiler) despite of version updates in the infrastructure tools. Second, ORKA-HPC is designed to reuse many optimized building blocks provided by vendors while also hiding vendor-specific details behind APIs. This enables ORKA-HPC to use various types and brands of FPGAs. Third, AST-level transformations that lower pragmas inside-out (after some normalizations) make it easy to add support for more pragmas and to hide low-level compiler construction details. Therefore, ORKA-HPC lowers the bar for future FPGA-based OpenMP research. The ORKA-HPC distribution – including all sources and benchmarks[5] – is available from `https://github.com/ORKA-HPC/orkadistro`

Currently, we are working on two main issues. First, as they can drastically decrease the resource consumption on FPGAs, we make bit-accurate data types available for programmers [4]. Second, because offloading unmodified C code to an FPGA in general does not lead to well-performing hardware, we automatically insert and tune HLS annotations to improve resource utilization on FPGAs.

Acknowledgments

The authors acknowledge the financial support by the Federal Ministry of Education and Research of Germany in the framework of ORKA-HPC (project numbers 01IH17003C and 01IH17003A). We also thank Marius Knaust (Zuse Institute Berlin, ZIB) for recommending the SPEC ACCEL benchmark and the ROSE team for the ROSE [18] framework with its OpenMP 3.0 lowering support.

References

1. Bosch, J., Tan, X., Filgueras, A., Vidal Piñol, M., Mateu, M., Jiménez-González, D., Álvarez, C., Martorell, X., Ayguadé, E., Labarta, J.: Application Acceleration on FPGAs with OmpSs@FPGA. In: Proc. Intl. Conf. on Field-Programmable Technology (FPT'18). pp. 70–77. Naha, Japan (Dec 2018)
2. Ceissler, C., Nepomuceno, R., Pereira, M.M., Araujo, G.: Automatic offloading of cluster accelerators. In: Proc. Intl. Symp. Field-Programmable Custom Computing Machines (FCCM'18). p. 224. Boulder, CO (Apr 2018)
3. Docker, Inc.: Docker, `https://www.docker.com`

[5] We do not include the SPEC ACCEL benchmarks as they require a SPEC license.

4. Finnerty, A., Ratigner, H.: Reduce Power and Cost by Converting from Floating Point to Fixed Point (Mar 2017), https://www.xilinx.com/support/documentation/white_papers/wp491-floating-to-fixed-point.pdf
5. Huthmann, J., Sommer, L., Podobas, A., Koch, A., Sano, K.: OpenMP Device Offloading to FPGAs Using the Nymble Infrastructure. In: Proc. Intl. Workshop On OpenMP, (IWOMP'20). pp. 265–279. Austin, TX (Sep 2020)
6. Intel Corporation: Intel Quartus Prime, https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html
7. Knaust, M., Mayer, F., Steinke, T.: OpenMP to FPGA Offloading Prototype using OpenCL SDK. In: Proc. Intl. Workshop High-Level Parallel Progr. Models and Supportive Env. (HIPS'19). pp. 387–390. Rio de Janeiro, Brazil (May 2019)
8. Korinth, J., Hofmann, J., Heinz, C., Koch, A.: The TaPaSCo Open-Source Toolflow for the Automated Composition of Task-Based Parallel Reconfigurable Computing Systems. In: Proc. Intl. Symp. on Applied Reconfigurable Computing, (ARC'19). pp. 214–229. Darmstadt, Germany (Apr 2019)
9. LLVM Team: LLVM 3.1 Release Notes, https://releases.llvm.org/3.1/docs/ReleaseNotes.html
10. Mayer, F., Knaust, M., Philippsen, M.: OpenMP on FPGAs—A Survey. In: Proc. Intl. Workshop On OpenMP, (IWOMP'19). pp. 94–108. Auckland, New Zealand (Aug 2019)
11. National Instruments Corporation: Artix-7 FPGA Development Board, https://reference.digilentinc.com/reference/programmable-logic/arty/start
12. Nepomuceno, R., Sterle, R., Valarini, G., Pereira, M., Yviquel, H., Araujo, G.: Enabling OpenMP Task Parallelism on Multi-FPGAs. arXiv:2103.10573 [cs.DC] (Mar 2021)
13. NixOS Contributors: Nix, https://nixos.org
14. OpenMP Architecture Review Board: Device data environments, https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf
15. Podobas, A., Brorsson, M.: Empowering OpenMP with automatically generated hardware. In: Proc. Intl. Conf. Systems, Architectures, Modeling and Simulation (SAMOS'16). pp. 245–252. Agios Konstantinos, Greece (Jan 2016)
16. Sommer, L., Korinth, J., Koch, A.: OpenMP device offloading to FPGA accelerators. In: Proc. Intl. Conf. Application-specific Systems, Architectures and Processors (ASAP'17). pp. 201–205. Seattle, WA (July 2017)
17. SPEC: SPEC ACCEL, https://www.spec.org/accel/
18. The ROSE Team: ROSE Compiler, http://rosecompiler.org/
19. Xilinx: Vitis High-Level Synthesis, https://www.xilinx.com/html_docs/xilinx2021_1/vitis_doc/introductionvitishls.html
20. Xilinx: Xilinx virtex ultrascale+ fpga vcu118 evaluation kit, https://www.xilinx.com/products/boards-and-kits/vcu118.html
21. Xilinx: Xilinx Vitis HLS LLVM 2020.2, https://github.com/Xilinx/HLS

All URLs referenced above were accessed on September 29, 2021.