



OpenMP on FPGAs—A Survey

Florian Mayer¹(✉), Marius Knaust², and Michael Philippsen¹

¹ Programming Systems Group,
Friedrich-Alexander University Erlangen-Nürnberg (FAU), Erlangen, Germany
{florian.andrefranc.mayer,michael.philippsen}@fau.de

² Zuse Institute Berlin, Berlin, Germany
knaust@zib.de

Abstract. Due to the ubiquity of OpenMP and the rise of FPGA-based accelerators in the HPC world, several research groups have attempted to bring the two together by building OpenMP-to-FPGA compilers. This paper is a survey of the current state of the art (with a focus on the OpenMP `target` pragma). It first introduces and explains a design space for the compilers. Design space dimensions include how FPGA infrastructure is generated, how work is distributed, and where/how target outlining is done. A table concisely condenses the available information on the surveyed projects which are also summarized and compared. The paper concludes with possible future research directions.

1 Introduction

OpenMP was originally intended to standardize the parallel programming of CPU-based SMP and NUMA systems. Prior to OpenMP 4.0, CPU-based systems were the only ones supported. Later, OpenMP 4.0 introduced the `target` pragma and allowed HPC programmers to exploit a cluster's heterogeneity by marking highly parallel regions of an algorithm to be offloaded to a more suited device (e.g. GPUs, FPGAs, etc.). Figure 1 illustrates the new situation and the typical approach to outline code for GPU and FPGA targets in the front-end of the compiler. Note that Fig. 1 simplifies. At least for GPUs there exists a compiler whose back-end builds code for both the host and the target. The thin dashed arrows show the traditional compilation pipeline prior to 4.0. In bold are the new challenges for the OpenMP implementer, as they now have to target both FPGAs and GPUs. Figure 1 also sketches the internals of an FPGA: Acc_1 through Acc_n denote hardware units doing actual calculations. The other blocks represent the infrastructure needed in order to run these hardware units on the FPGA. As almost

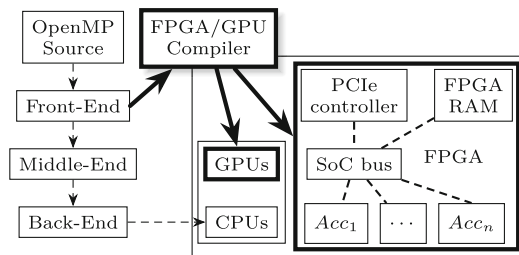


Fig. 1. OpenMP compilation.

As almost

everything in the FPGA can be configured arbitrarily, one of many possible configurations is shown. To make use of the accelerators, OpenMP compilers need to solve novel problems: How to transfer data from the CPU to GPUs/FPGAs and back? What kinds of handshaking to use? What hardware blocks to chose to make up the FPGA configuration? What FPGA-internal bus system to use? Over the years, several researchers answered some of those questions in various ways.

Here we survey these papers and cover the current state of the art. Section 2 sketches the design space of OpenMP-to-FPGA compilers. Section 3 discusses the published research using that design space.

2 Design Space

When mapping OpenMP code to FPGA-based accelerators, a tool chain has a variety of different design decisions to choose from. This section covers feasible approaches and identifies the dimensions that the next section uses to categorize published systems. Of course, categories are not always black-and-white.

In this paper the term *architecture* refers to all components a system is built from, how those components behave, and how they interact with each other. An FPGA (or FPGA chip) usually consists of both a reconfigurable part and a fixed ASIC part (for instance ARM cores, RAMs, etc.). This paper uses the term FPGA *fabric* (or just fabric) to denote the reconfigurable part. A fabric can emulate arbitrary hardware. Its *configuration* (the *bitstream*) encodes this hardware. Without the configuration there are not even connections to the static ASIC parts of the FPGA. Any real-world configuration thus must consists of two parts: First functional entities, also known as kernel IPs (Intellectual Property) that perform desired calculations from the regions inside the OpenMP program, and second, the infrastructure for getting data to/from those functional blocks, also known as Low-Level Platform (LLP). This part of the configuration enables internal and external communication.

2.1 Low-Level Platform

While kernel IPs are application specific, in general and in this paper, the LLP is composed from pre-built IPs that implement bus communication, memory management, etc. These static LLP IPs sometimes can be configured (e.g., a bus IP could be configured to host more than 4 bus masters). Conceptually the compiler could generate application-specific LLP IPs from the ground up to best fit the served kernel IPs, but we do not know of any such attempt.

There are 4 classes of LLP:

Generic-Static: Fixed and pre-built ahead of time for *all* OpenMP programs. Such LLPs typically include a memory controller for FPGA memory, a communication controller (PCIe), an on-chip bus system (AXI), and sometimes a softcore CPU that manages the overall system. Because the bus system is fixed

for all OpenMP programs, this type of LLP is limited to a constant number of accelerator blocks.

Specialized-Static: Specific for *one* OpenMP program and built at compile time. The compiler uses some static code analyses to compose LLPs of this type according to the needs of the OpenMP program at hand. For example, for a throughput-heavy OpenMP program the compiler would pick a different bus system than for compute-bound code. Similarly, an AXI streaming bus is not added to the fabric if the code cannot make use of it. Such tailoring saves fabric space that can be used for additional or larger kernel IPs. The LLP is static as it does not change after it has been configured to run on the FPGA.

Generic-Dynamic and *Specialized-Dynamic*: pre-built for *all*/specialized for *one* OpenMP program/s, but adapting based on runtime measurements. LLPs of these two *dynamic* classes adapt themselves depending on the current runtime requirements of the OpenMP program. They require a partially reconfigurable FPGA [38]. The LLP could for instance use different bus systems in different phases of the execution. After a throughput-heavy initialization, another bus system can be used, freeing space for additional computational kernels. To the best of our knowledge there is not yet an OpenMP-to-FPGA compiler that employs a dynamic LLP, neither a *generic* one (that fits all OpenMP programs) nor a *specialized* one that reconfigures itself from a tailored set of LLP-IPs.

2.2 Distribution of Work

OpenMP 4.0 allows computations to be distributed over all available computing devices. For the distribution decision, the compiler first assigns code blocks to the devices statically and decides how many copies of the code block to instantiate. We survey approaches that also decide statically where to execute the code blocks. A runtime system could optionally schedule them dynamically.

Dynamic scheduling is outside the focus of this paper as it is – if at all on FPGAs – used for `task` scheduling only [7]. Figure 2 lists some of the abbreviations used in this paper. While conceptually it is possible to let the programmer specify the static distribution explicitly or to use some sophisticated optimization routine to find a best-performing distribution at compile time, existing OpenMP-to-FPGA compilers make a rather simplistic choice and fall into either of the following two categories: CF_{hw} : Plain code, including the main thread, executes on the CPU, while for `target` pragma annotated code, FPGA hardware is synthesized that performs the calculation. This is called the *host-centric* approach. Note, that whenever possible the hardware synthesis tool makes use of available ASIC blocks on the FPGA (like DSPs).

C^x	CPU with instruction set x
G^x	GPU with instruction set x
F_{hc}	FPGA with a <i>hardcore</i> CPU
F_{sc}	FPGA with a <i>softcore</i> CPU
F_{hw}	FPGA with custom <i>hardware</i>
$F_{hw,sc}$	FPGA with custom <i>hardware</i> and a <i>softcore</i> CPU

Fig. 2. Abbreviations for devices.

$F_{hw,sc}$: In the *fpga-centric* approach, both the plain code (including the main thread) and the pragma code execute on the FPGA; the sequential code runs on a softcore CPU on the fabric. Again, the hardware synthesis tool makes use of ASIC blocks for both the softcore and the custom hardware.

There are other design choices. For example to use multiple FPGAs, to employ an ASIC hardware CPU if it is available, to also offload code to a GPU, etc. To the best of our knowledge these choices have not yet been explored.

2.3 Outlining

To the best of our knowledge, all OpenMP 4.0 compilers that support target offloading to FPGAs so far implement this as follows: They replace the marked code with function calls (that may be bundled into a stub). Some of the functions handle the communication of data between the host and the accelerator. One function initiates the execution of the payload code on the accelerator that implements the marked code block. To construct this payload, the compiler *outlines* the marked code block into a separate function that is then fed into an accelerator-specific tool chain. This can be a compiler for a GPU or a high-level synthesis tool (HLS) in the case of an FPGA as shown in Fig. 3. Some GPU compilers spit out GPU code in their back-ends. But as it still has to be explored if this is a better choice for FPGA code, we focus on the front-end outlining options in this survey. It is common practice to outline each `target` region individually. The design choice is whether to execute the outlining on the level of the abstract syntax tree (AST) or to do it on the immediate representation (IR) of the code.

As in general, the host and the accelerator do not share memory, data needs to be shipped to the accelerator (and back) so that the kernel IP can access it. Hence, the compiler and the runtime system must solve three problems. First, *identify the values that need to be passed to the outlined code*. Used techniques range from naively copying all the data in the scope to relying on compiler analyses or programmer specifications (data `map` clause) to limit the amount of moved data and to thus gain performance. Second, *create a parameter list for the payload function (fed into the accelerator tool chain)*. Used techniques range from naively creating one parameter for each value, to bundling values in structs or arrays. For FPGAs, fewer parameters result in fewer bus ports in the generated FPGA hardware which saves valuable resources on the fabric that then can be used for the functional entities. Despite the importance, most papers do not reveal how they generate parameter lists. Third, *generate API calls to transfer values to the accelerator*. Value transfer routines can be asynchronous (non-blocking) or synchronous (blocking). What works best depends on the accelerator hardware. Both techniques are used to transfer data to/from computing devices. For FPGAs, their ASIC devices constrain what transfer method works best for any given application. While conceptually it is possible to tailor the transfer routines to fit the LLP (and vice versa), to the best of our knowledge there is not yet an OpenMP-to-FPGA compiler that exploits this option.

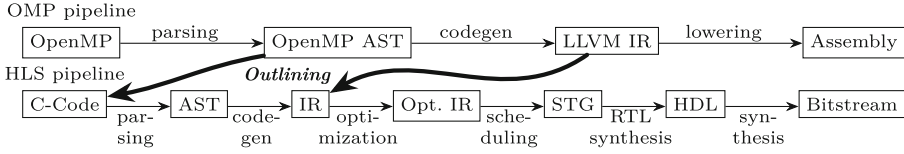


Fig. 3. Front-end outlining options in an OpenMP-to-FPGA compiler.

Ability to Compose Streams: It is state-of-the-art to outline each code block individually. This implies that modified data is shipped back from the accelerator to the host CPU even if the next outlined code block uses the same data. In this case, regardless of the type of accelerator, transfer cost can be saved. On FPGAs, this optimization idea may have an even larger impact than on GPUs as two subsequent `target` regions could use a streaming design to exploit pipeline-style parallelism for better performance. To the best of our knowledge this has not yet been explored.

2.4 Supported Pragmas

When distributing/mapping a code block to the FPGA, compilers may or may not be able to exploit OpenMP pragmas. For the discussion, we distinguish between OpenMP pragmas that are defined in the standard and HLS-specific ones defined by tool vendors. OpenMP-to-FPGA projects that use HLS-tools often *pass through* the latter to the HLS tool chain when they outline the code according to Fig. 3.

As the surveyed projects are still prototypes, they ignore most of the regular OpenMP pragmas. Tables 1 and 2 hold a positive list of those pragmas that they support, in the sense that a pragma somehow affects the code they build during outlining and that they feed into the HLS tool chain. As mentioned before, data shipment between host and FPGA matters. Hence, Tables 1 and 2 also cover whether a system supports the `map` clauses of the `target` pragma.

2.5 Optimization Techniques

OpenMP compilers also differ w.r.t. the (few) optimization technique they apply along their pipelines. This is outside the scope of this survey.

2.6 High-Level Synthesis

As shown in Fig. 3, the OpenMP-to-FPGA tool chain uses a High-level synthesis (HLS) tool to transform C/C++ code into FPGA hardware. The design

spaces comprises three different types of HLS: *Data path* based (DP), *finite state machine* based (FSM), and *hybrid* HLS [31]. A DP-based HLS produces the best hardware for C/C++ code that is highly data parallel. It does not work well for code that has many branches in it [16]. FSM-based HLS tools can translate most programs (with the exception of programs that use recursion, malloc, or function pointers). Unfortunately, FSMs in hardware in general suffer from a higher latency than DP designs. Hybrid HLS tools are most commonly used because they (try to) combine the advantages of the two pure types: Vivado HLS [40], Intel OpenCL SDK for FPGA [18], or Intel Quartus Prime (previously Altera Quartus) [20] are well known hybrid commercial tools. LegUp [9] is a alternative from the research community. There also is a commercial fork available [25]. CoDeveloper [17] is a special HLS that only accepts the Impulse-C language. It does not fit into any of the above categories.

Which type of HLS to use for each target region is a design space decision. All surveyed projects treat *every* region the same way and use the same hardware synthesis for it, even though (at least conceptually) the decision can be made on a per-region basis as the amount of parallelism varies among them.

3 Survey

Tables 1 and 2 illustrate which design space decisions existing approaches took. The columns are ordered with the latest system first. As most authors have not named their systems, we use the name of the first author instead. We did not find more recent papers than the cited ones. There are references to code archives if systems are available for download. However, unfortunately none of the systems ran out-of-the box for us. The rows of the table are grouped according to the design space discussion in Sect. 2. Some areas of the table give more details on the design space aspects. In the HLS section the table lists which tools have been used. The *Misc.* rows mention which compiler frameworks and libraries have been used to build the system (e.g., Clang [24], LLVM [28, 36], Mercurium [2], GCC [35], Nanos++ [3], or libomptarget [42]), for which FPGA boards they can be used, how the structure of the target systems looks, and whether the system offers a complete workflow that does not require any intermediate manual work along the tool chain. Simplified block diagrams sketch the target structure. Here M stands for memory, B for bus interconnect, C for CPU, G for GPU, A for application specific kernel IP, S for synchronization core, D for hardware debugger, and T for timer. Lines represent physical connections and parts in cyan live on the fabric. The superscripts give additional information on a component (e.g., C^{xeon} for a Xeon CPU). The subscripts show instance counts or memory sizes (e.g., C₄ for a 4-core CPU, A_n for n application specific kernel IPs, M_{2gb} for a memory block with 2 gigabytes of storage).

Table 1. Project overview (2019–2014)

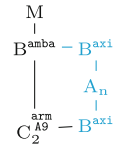

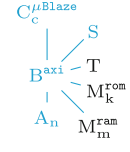
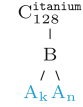

	Knaust	Bosch	Ceissler	Sommer	Podobas	
Year	2019	2018	2018	2017	2014–2016	
Papers	[21]	[7]	[12]	[34]	[30,31]	
Code	✗	[5]	[11]	✗	✗	
LLP	generic-static	specialized-static	generic-static	specialized-static	specialized-static	
Work distrib.	<i>host-centric</i> CF _{hw}	<i>host-centric</i> CGF _{hw}	<i>host-centric</i> CF _{hw}	<i>host-centric</i> CF _{hw}	<i>fpga-centric</i> F _{hw,sc}	
Pragmas	OpenMP	target (with map)	parallel, parallel for, target (with map), target data, declare target	target (with map)	parallel, task, single, taskwait	
	HLS	✓	✓	✗	✓	
Outlining	Level	IR	AST	n.a.	AST	
	HLS	Hybrid: Intel OpenCL	Hybrid: Vivado HLS	n.a.	Hybrid: Vivado HLS	Configurable from FSM-to DP-based
	Streaming	✗	✓	✗	✗	✗
Misc.	Compiler Toolkit	Clang, LLVM	Mercurium, Nanos++	Clang, LLVM, libomptarget	Clang, LLVM, libomptarget, TPC [22,23]	Custom C89 Compiler
	Target system(s)	Intel Board with Arria 10 GX	Xilinx Board with Zynq Ultrascale+	Amazon AWS F1, Intel HARP2	Xilinx VC709 Board	Altera ED5 Board with Stratix V
	Target Structure					
	Complete workflow	✓	✓	✗	✓	✓

3.1 Projects

Below we describe the essence of the systems and their main contributions. Readers may skip the lines in fine print that detail the corresponding cells of the table. In bold is the name of the addressed cell of Tables 1 and 2. Most papers evaluate their compilers. Where appropriate, we summarize the evaluation results with respect to the benchmarks used, the method of comparison, and the main evaluation results obtained.

Knaust's host-centric prototype uses Clang to outline `omp target` regions at the level of the LLVM IR and feeds them into Intel's OpenCL HLS to generate a hardware kernel for the FPGA. This approach relies on an undocumented IR interface of the HLS. For the communication between host and FPGA, Knaust uses Intel's OpenCL API. It is unique how this work exploits a state-of-the-art commercial HLS with low transformation efforts.

Table 2. Project overview (2014–2006)

	Filgueras	Choi	Cilardo	Cabrera	Leow	
Year	2014	2013	2013	2009	2006	
Papers	[15]	[13]	[14]	[8]	[26]	
Code	[4]	[10]	✗	✗	✗	
LLP	specialized-static	generic-static	specialized-static	generic-static	n.a.	
Work distrib.	<i>fpga-centric</i> , (<i>host-centric</i>) $F_{hw,hc}$, (CF_{hw})	<i>fpga-centric</i> $F_{hw,sc}$	<i>fpga-centric</i> $F_{hw,sc}$	<i>host-centric</i> CF_{hw}	<i>fpga-centric</i> F_{hw}	
Pragmas	OpenMP	parallel, target (with map), task, taskwait	parallel for (with reduction), parallel, master, critical, atomic	Complete OpenMP 3.1	target (with map), task, block	threadprivate, barrier, for, parallel, parallel for, master, critical, atomic, single, section, for section, ordered, flush
	HLS	✓	✗	?	✗	✗
Outlining	Level	AST	IR	AST	n.a.	n.a.
	HLS	Hybrid: Vivado HLS	Hybrid: LegUp	Impulse CoDeveloper	n.a.	FSM-based
	Streaming	✗	✗	✗	✗	✗
Misc.	Compiler Toolkit	Mercurium, Nanos++	LLVM-GCC	Custom OpenMP, Xilinx EDK	Mercurium, GCC, SGI RASClib	C-Breeze [27]
	Target system(s)	Xilinx Board with Zynq-7000	Altera Board with Stratix IV	Boards supported by Xilinx EDK	SGI RASC 2.2 Board with 2 Virtex 4	Celoxica RC100 Board with Spartan II
	Target Structure					
	Complete workflow	✓	✓	✓	✗	✓

LLP: The internals of the LLP cannot be classified exactly because Intel’s SDK is proprietary. However, the *Floorplan Optimization Guide* of the SDK mentions that the LLP is loaded only once and that partial reconfiguration is used to hot-swap kernel IP bitstreams at runtime [19]. **Pragmas:** Knaust passes the `unroll` pragma to the underlying HLS. From the map clauses of the `target` pragma, only *array sections* are unsupported.

Evaluation: Two Sobel filters (unoptimized and optimized for FPGAs) run on a $4096 \cdot 2160 \cdot 8$ bit matrix. The CPU-only version is compiled without `-fopenmp`. The pure optimized kernel for the FPGA is $4\times$ as fast as one CPU core, but this can hardly amortize the cost of transfer and initialization.

OmpSs@FPGA by *Bosch* et al. improves and generalizes the work by Filgueras et al. Memory on the accelerator is used for data sharing (streaming). This is the only system in the surveyed set that not only outlines code to

the FPGA but also addresses the GPU. Moreover, the `tasks` are *dynamically* scheduled onto the devices.

LLP: The authors do not describe the structure of the LLP in detail. However, in contrast to Filgueras et al. there are hints that it falls into the specialized-static class.

Evaluation: On three benchmarks (matrix multiplication, n-body, Cholesky decomposition) the authors compare the baseline runtime (measured on a $C_4^{\text{ARM-A52}}$ with 4 GB of shared memory) with their FPGA versions. For the Cholesky decomposition, the performance drops by about $2\times$. For n-body, the FPGA version is $15\times$ faster. The matrix multiplication on the FPGA achieves $6\times$ the GFLOP/s.

Ceissler et al. propose HardCloud, a host-centric extension for OpenMP 4.X. There is no outlining of code blocks. Instead, HardCloud makes pre-synthesized functional units for FPGAs easier to use in existing OpenMP code.

LLP: While the authors do not describe the internals of their LLP, the first figure in [12] suggests it to be generic-static. **Complete Workflow:** Users need to manually design hardware and synthesize it to a kernel IP as there is no outlining. HardCloud automates the data transfer and device control.

Evaluation: The authors claim to have achieved speed-ups on the HARP 2 platform between $1.1\times$ and $135\times$. However there is no further information about the context or the benchmark codes.

Sommer et al. use Clang to extract `omp target` regions from the source program (at AST-level) and feed them into the Vivado HLS that then generates kernel IPs. Calls to their Thread Pool Composer (TPC) API (now called TaPaSCo) injected into the program implement the host-to-FPGA communication. The strength of the prototype is that it fully supports `omp target` (including its `map` clause). This project is the first that integrated `libomptarget`.

LLP: TPC assembles a specialized-static LLP from the following set: the kernel IPs, configuration files describing the IPs, and an architecture configuration file describing for example what bus system to use (only AXI in their work).

Evaluation: For 6 benchmarks from the Adept benchmark suite [1], the authors compare the runtime of -O3-optimized i7 CPU code (4 cores) to their FPGA-only version (with HLS pragmas). The CPU outperforms the FPGA version by $3\times$ to $3.5\times$ (without the HLS pragmas: $6\times$ to $9\times$).

In the system by *Podobas* et al. the compiler extracts `task`-annotated functions and synthesizes a specialized system on a chip (SoC) for them. It rewrites the main program to use these units and compiles it to run on a softcore CPU that is also placed on the FPGA. While their first system builds isolated FPGA hardware per `task`, the authors later fuse `task` kernel IPs for resource sharing. To do so they use Gecode [32] to solve constraint programs in which the constraints express what to share.

LLP: Altera Quartus builds the specialized-static LLP. It connects the kernel IPs and assigns an own address space on a shared Avalon bus to each of them. It also connects auxiliary blocks and the softcore to that bus. **Pragmas:** The behaviour of the pragmas `parallel` and `single` slightly differ from the OpenMP specification. If both pragmas are written consecutively in the source code, the system replaces them by a function call that initializes the LLP. The behaviour of just one pragma is left open. **HLS:** The authors use the custom hardware synthesis tool fpBLYSK. Depending on the command line flags, their HLS can generate purely FSM-based designs that execute one instruction per state, or it can combine several instructions into each FSM state, giving the design a DP taste.

Evaluation: The authors study three basic benchmarks (π , Mandelbrot, and prime numbers). For the first two compute-bound benchmarks, the FPGA version outperforms both CPU-only versions (57-core Intel Xeon PHI and 48-core AMD Opteron 6172) by a factor of 2 to 3. However, for the memory-bound third benchmark, the CPU versions are about 100 times faster.

Filgueras et al. add support for the Xilinx Zynq FPGA [41] to the OmpSs framework [6] that provides `task` offloading to any kind of supported accelerator. Although their prototype exclusively uses the FPGA’s ASIC CPUs for the sequential portion of the source code (fpga-centric, $F_{hw,hc}$). The authors claim any work distribution to be possible (e.g., CF_{hw}). The system is the first that combines this flexibility with the task based paradigm (including task dependencies).

LLP: The authors do not describe in detail how the compiler builds the LLP. **Pragmas:** The `task` pragma is extended so that it can be used to annotate functions and to specify dependencies between tasks (clauses `in`, `out`, or `inout`). **Compiler Toolkit:** A custom pass implemented in the Mercurium framework outlines and injects calls for data shipment. The Nanos++ OpenMP runtime provides task parallelism and dependency-based task scheduling.

Evaluation: On four numeric benchmarks (two matrix multiplications with different matrix sizes, complex covariance, and Cholesky decomposition) the FPGA version achieves speed-ups between $6\times$ to $30\times$ compared to a single ARM A9 core.

The system by *Choi et al.* is fpga-centric. Its main objective is to exploit the information on parallelism that the programmers provide in (six supported) pragmas, to generate better, more parallel hardware. The compiler synthesizes one kernel IP per thread in the source program (for example a code block annotated with `parallel num_threads(4)` specifies 4 hardware threads). The support for the `reduction` clause of `parallel` or `parallel for` is unique, although the authors do not elaborate on how they achieve reduction on variables in hardware.

Pragmas: The system is limited to OpenMP constructs for which the compiler can statically determine the number of threads to use. Nested parallelism is possible, although limited to two levels. **HLS:** The extended LegUp generates parallel hardware for `parallel` and `parallel for` and utilizes the other pragmas (`atomic`, etc.) to synchronize between the threads. For `atomic` and `critical`, a hardware mutex core is synthesized.

Evaluation: With the best compiler configuration for the FPGA versions, 7 benchmarks (Black-Scholes option pricing, simulated light propagation, Mandelbrot, line of sight, integer set division, hash algorithms, double-precision floating point sine function) show a geomean speed-up of $7.6\times$ and a geomean area-delay product of 63% compared to generated sequential hardware.

Cilardo et al. think of OpenMP as a system-level design language (e.g., for heterogeneous targets like the Xilinx Zynq) and present a compiler that uniquely supports the complete OpenMP 3.1 specification. They map the whole OpenMP program to the FPGA (where they use softcore processors to run threads with many branches). Note, that the authors even map nested parallelism (i.e., nested `omp` work sharing constructs) to hardware (by exploiting the tree-like structure to minimize path lengths for common control signals).

LLP: The Xilinx Embedded Development Kit (EDK) [39] was used to build the LLP, but the authors only reveal that they use the MicroBlaze [37] softcore for the sequential parts. **Pragmas:** As their custom front-end only supports OpenMP-parsing, it is unlikely that HLS pragmas are passed through.

Evaluation: When comparing their sieve of Eratosthenes to the results from Leow et al. the authors see twice the speed-up. Furthermore, a runtime overhead inspection of the implemented OpenMP directives (`private`, `firstprivate`, `dynamic`, `static`, and `critical`) shows significantly less overhead than the SMP versions on an Intel i7 ($6\times$, $1.2\times$, $3.1\times$, $10.5\times$, and $2.64\times$, respectively).

Cabrera et al. extend OpenMP 3.0 with new semantics for `task` and `target` to ease the offloading to pre-synthesized functional units, i.e., hand-built kernel IPs. There is no outlining of code blocks. Their main contribution is that they provide support for SGI's RASC platform [33] and a multi-threaded runtime library layer with a bitstream cache that enables parallel computation on both the host and the FPGA even while the bitstream is being uploaded.

LLP: The target system is embedded into an SGI Altix 4700 server and a proprietary generic-static LLP provided by SGI is used. **Pragmas:** The work introduces the `pragma block` that helps to guide loop restructuring and data partitioning of arrays. **HLS:** Xilinx ISE 9.1 (now part of the Vivado Design Suite) is used to generate bitstreams. **Compiler Toolkit:** Offloading is implemented as a plugin for the Mercurium compiler. The host-side code compiles with GCC 4.1.2 and links against a custom runtime library.

Evaluation: The paper only shows runtimes of a matrix multiplication (32^2 , 64^2 , and 128^2) without any comparisons with CPU codes.

Leow et al. view OpenMP programs as a hardware description language that programmers use to explicitly control the parallelism of the resulting hardware. In contrast to other systems, the result is a *single* hardware entity (F_{hw}) without any outlining and work distribution at all.

HLS, Compiler Toolkit: The translation is integrated into the C-Breeze compiler framework as a custom high-level synthesis pass. It can generate both Handel-C [29] and VHDL code, but different restrictions apply. For example, the VHDL back-end cannot deal with global variables in the input program.

Evaluation: For the first two of the benchmarks (matrix multiplication, sieve of Eratosthenes, Mandelbrot), the FPGA versions achieves speed-ups of $25\times$ and $7\times$ over a symmetrical SMP (UltraSPARC III with 8GiB). For Mandelbrot, the FPGA version is slower than the SMP, even though all SMP codes were compiled with `-O0`.

3.2 Discussion

The surveyed projects are prototypes focusing only on a small subset of OpenMP pragmas and require users with compiler- and/or FPGA-expertise. Almost half of the tools still require manual outlining and invoking of HLS tools, and for only three systems the source code is available.

About half the systems are *host-centric*. The general idea is to achieve performance and efficiency by standing on the shoulders of giants. Research falls into two groups. Systems in one group (Ceissler and Cabrera) assume pre-synthesized, highly optimized and efficient kernel IPs that need to be interconnected. The underlying hope is that the generated glue hardware is not that crucial for performance. Because of the pre-built kernels those systems are tied tightly to specific FPGA platforms, e.g., Intel HARP2, Amazon AWSF1, or SGI RASC.

The other group outlines code blocks and feeds them into an HLS tool chain for building the kernel IP. The hope is that vendors invest enough money and man power into these tools so that they synthesize efficient FPGA hardware. As shown in Fig. 3, outlining can either be done on the level of ASTs or at the IR-level. The latter approach (taken by Knaust) not only suffers from not being future-proof as current HLS tools only provide undocumented IR-level interfaces. The other disadvantage is that it is complicated to pass HLS pragmas to the HLS tool. The problem is that such pragmas need to be transformed into unofficial IR annotations that are even more likely to change or to become unavailable in the future. AST-based outlining does not have these disadvantages because passing HLS pragmas is easy as ASTs can be trivially converted to C code and because using the HLS on AST-level can be expected to work for the foreseeable future. The main problem of using an HLS from a certain vendor is that only this vendor's FPGAs can be used.

The *fpga-centric* approaches understand a whole OpenMP program as a high-level description of the FPGA hardware that has to be built, i.e., the FPGA is no longer used as an accelerator but it is the only device. This group of researchers usually builds specific compilers that focus on optimizing transformations for pragmas that are directly relevant for the hardware synthesis. Depending on the size and the importance of the sequential code blocks, systems either use a softcore processor on the fabric for it, or they include the sequential code into the kernel IP. On the one hand, FPGAs programmed with compilers that use the pragma information are claimed to perform better because the programmer can specify application-specific parallelism. The main drawback, on the other hand, is that host CPUs (optimized for memory-intensive sequential workloads) stay unused. The general problem of the *fpga-centric* approaches is that in general they only work for a specific FPGA and/or tool chain.

4 Conclusion and Future Work

The basic technical issues of host-centric target offloading with a CF_{hw} work distribution have been covered extensively, both with outlining on the AST- or IR-level. Similarly, the fpga-centric compilers that treat OpenMP as some sort of hardware/system-level description languages use basic mapping regimes to assemble FPGA bitstreams for targets and to distribute the work in various ways.

The field is in a proof-of-concept state. We think that what is needed now is a focus on performance and efficiency. There is not yet a benchmark to quantitatively compare systems. Little work has been done so far on optimization. For example, self-adapting, dynamic LLPs may be the better infrastructure and may free FPGA resources for functional entities/kernel IPs. Instead of treating each `omp target` region in isolation, it may be promising to explore how to automatically connect kernel IPs in a streaming fashion (as human FPGA engineers usually do). Currently, FPGA-expertise is required to achieve better performance than leaving the FPGA unused. This burden needs to be taken from the OpenMP programmer, i.e., they should no longer need to be experts in HLS pragmas and in the tools of FPGA vendors.

From our perspective, the key to all of this is a better code analysis that not only spans across all the OpenMP pragmas used in a given code, but that also spans from IR-level to low-level HLS transformations. We feel that at least there should be a (to be designed) interface between the various tools along the tool chain to convey optimization-related analysis data.

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 01IH17003D).

References

1. Adept: Adept Benchmark Suite. <http://www.adept-project.eu/benchmarks.html>
2. Barcelona Supercomputing Center: Mercurium C/C++/Fortran source-to-source compiler. <https://www.bsc.es/research-and-development/software-and-apps/software-list/mercurium-ccfortran-source-source-compiler>
3. Barcelona Supercomputing Center: Nanos++. <https://pm.bsc.es/nanos>
4. Barcelona Supercomputing Center: OmpSs@FPGA. <https://pm.bsc.es/ompss-at-fpga>
5. Barcelona Supercomputing Center: Repository of the Mercurium C/C++/Fortran source-to-source compiler. <https://www.github.com/bsc-pm/mcxx>
6. Barcelona Supercomputing Center: The OmpSs Programming Model. <https://pm.bsc.es/ompss>
7. Bosch, J., et al.: Application acceleration on FPGAs with OmpSs@FPGA. In: Proceedings of the International Conference on Field-Programmable Technology (FPT 2018), Naha, Japan, December 2018

8. Cabrera, D., Martorell, X., Gaydadjiev, G., Ayguade, E., Jiménez-González, D.: OpenMP extensions for FPGA accelerators. In: Proceedings of the International Conference on Systems, Architectures, Modeling and Simulation (SAMOS 2009), Samos, Greece, pp. 17–24, July 2009
9. Canis, A., et al.: LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In: Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA 2011), Monterey, CA, pp. 33–36, February 2011
10. Canis, A., Choi, J., Chen, Y.T., Hsiao, H.: LegUp High-Level Synthesis. <http://legup.eecg.utoronto.ca/>
11. Ceissler, C.: HardCloud Github Wiki. <https://github.com/omphardcloud/hardcloud/wiki>
12. Ceissler, C., Nepomuceno, R., Pereira, M.M., Araujo, G.: Automatic offloading of cluster accelerators. In: Proceedings of the International Symposium on Field-Programmable Custom Computing Machines (FCCM 2018), Boulder, CO, p. 224, April 2018
13. Choi, J., Brown, S., Anderson, J.: From software threads to parallel hardware in high-level synthesis for FPGAs. In: Proceedings of the International Conference on Field-Programmable Technology (FPT 2013), Kyoto, Japan, pp. 270–277, January 2013
14. Cilaro, A., Gallo, L., Mazzeo, A., Mazzocca, N.: Efficient and scalable OpenMP-based system-level design. In: Proceedings of Design, Automation and Test in Europe (DATE 2013), Grenoble, France, pp. 988–991, March 2013
15. Filgueras, A., et al.: OmpSs@Zynq all-programmable SoC ecosystem. In: International Symposium on Field-Programmable Gate Arrays (FPGA 2014), Monterey, CA, pp. 137–146, February 2014
16. Halstead, R.J., Najjar, W.A.: Compiled multithreaded data paths on FPGAs for dynamic workloads. In: Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES 2013), Montreal, QC, pp. 21–30, September 2013
17. Impulse Accelerated Technologies: Impulse CoDeveloper. <http://web.archive.org/web/20180827120514/impulseaccelerated.com/tools.html>
18. Intel Corporation: Intel FPGA SDK for OpenCL. <https://www.intel.de/content/www/de/de/software/programmable/sdk-for-openc/overview.html>
19. Intel Corporation: Intel FPGA SDK for OpenCL Board Support Package Floorplan Optimization Guide. <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/an/an824.pdf>
20. Intel Corporation: Intel Quartus Prime. <https://www.intel.de/content/www/de/de/software/programmable/quartus-prime/overview.html>
21. Knaust, M., Mayer, F., Steinke, T.: OpenMP to FPGA offloading prototype using OpenCL SDK. In: Proceedings of the International Workshop High-Level Parallel Programming Models and Supportive Environment (HIPS 2019), Rio de Janeiro, Brazil, p. to appear, May 2019
22. Korinth, J., Chevallier, D.d.l., Koch, A.: An open-source tool flow for the composition of reconfigurable hardware thread pool architectures. In: Proceedings of the International Symposium on Field-Programmable Custom Computing Machines (FCCM 2015), Vancouver, BC, pp. 195–198, May 2015
23. 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: Proceedings of the International Symposium on Applied Reconfigurable Computing, (ARC 2019), Darmstadt, Germany, pp. 214–229, April 2019

24. Lattner, C., The Clang Team: Clang: a C language family frontend for LLVM. <https://clang.llvm.org/>
25. LegUp Computing: LegUp. <http://www.legupcomputing.com/>
26. Leow, Y., Ng, C., Wong, W.: Generating hardware from OpenMP programs. In: Proceedings of the International Conference on Field Programmable Technology (FPT 2006), Bangkok, Thailand, pp. 73–80, December 2006
27. Lin, C., Guyer, S., Jimenez, D.: The C-Breeze Compiler Infrastructure. <https://www.cs.utexas.edu/users/c-breeze/>
28. LLVM Team: llvm-gcc - LLVM C front-end. <https://releases.llvm.org/2.9/docs/CommandGuide/html/llvmgcc.html>
29. Mentor: Handel-C. <https://www.mentor.com/products/fpga/handel-c/>
30. Podobas, A.: Accelerating parallel computations with OpenMP-driven system-on-chip generation for FPGAs. In: Proceedings of the International Symposium on Embedded Multicore/Manycore SoCs (MCSoc 2014), Aizu-Wakamatsu, Japan, pp. 149–156, September 2014
31. Podobas, A., Brorsson, M.: Empowering OpenMP with automatically generated hardware. In: Proceedings of the International Conference on Systems, Architectures, Modeling and Simulation (SAMOS 2016), Agios Konstantinos, Greece, pp. 245–252, January 2016
32. Schulte, C., Lagerkvist, M., Tack, G.: Gecode. <https://www.gecode.org>
33. Silicon Graphics: Reconfigurable Application-Specific Computing User’s Guide, March 2006. <https://irix7.com/techpubs/007-4718-004.pdf>
34. Sommer, L., Korinth, J., Koch, A.: OpenMP device offloading to FPGA accelerators. In: Proceedings of the International Conference on Application-specific Systems, Architectures and Processors (ASAP 2017), Seattle, WA, pp. 201–205, July 2017
35. The GCC Team: GCC, the GNU Compiler Collection. <https://gcc.gnu.org/>
36. The LLVM Team: The LLVM Compiler Infrastructure. <https://llvm.org/>
37. Xilinx: MicroBlaze Soft Processor Core. <https://www.xilinx.com/products/design-tools/microblaze.html>
38. Xilinx: Partial Reconfiguration in the Vivado Design Suite. <https://www.xilinx.com/products/design-tools/vivado/implementation/partial-reconfiguration.html>
39. Xilinx: Platform Studio and the Embedded Development Kit (EDK). <https://www.xilinx.com/products/design-tools/platform.html>
40. Xilinx: Vivado Design Suite. <https://www.xilinx.com/products/design-tools/vivado.html#documentation>
41. Xilinx: Zynq-7000 SoC Data Sheet. https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf
42. Yviquel, H., Hahnfeld, J.: libomptarget - OpenMP offloading runtime libraries for Clang. <https://github.com/clang-omp/libomptarget>