Synthesis of Interleaved Multithreaded Accelerators from OpenMP Loops

Lukas Sommer, Julian Oppermann, Jaco Hofmann, and Andreas Koch

Embedded Systems and Applications Group (ESA)
Technische Universität Darmstadt, Germany
{sommer, oppermann, hofmann, koch}@esa.tu-darmstadt.de

***Abstract*—Similarly to CPUs and GPUs, FPGA-based accelerators can also profit from exploiting thread-level parallelism. Thus, the synthesis tools for generating the circuits from high-level languages need to be extended appropriately.**

We present an extension of the Nymble hardware/software-co-compiler for the automatic synthesis of hardware accelerators from OpenMP worksharing loops, and describe modifications to the datapath- and memory-architecture for multi-threaded execution.

The new execution model employs both spatial as well as thread-level parallelism in the microarchitecture of the generated accelerator, with the aim to efficiently hide memory access latencies.

We are able to gain raw speedups of more than a factor of 3x, and improve the utilization of the computing unit by more than factor 8x, when executing four threads instead of a single one on the computing units.

I. Introduction

Recently, FPGAs are increasingly employed in large datacenters as an alternative to GPUs as dedicated hardware accelerators. An example for the use of FPGAs in high-performance computing (HPC) scenarios is Microsoft's Bing and Azure Cloud [1], where all compute nodes are equipped with FPGAs for compute and high-speed network-processing tasks. FPGA-based acceleration has also been used successfully for high-speed network security monitoring [2].

High-level synthesis (HLS) of accelerator designs from high-level language input programs is a powerful method to automatically implement FPGA-based hardware accelerators. As of today, HLS-tools mainly gain their speedup over a sequential execution on a CPU by exploiting the instruction-level parallelism (ILP) present in the input programs. However, the instruction-level parallelism contained in applications is typically limited, e.g., by data-dependencies between operations or by control-flow.

For the successful employment of HLS-generated FPGA-designs in HPC-scenarios, the limited amount of speedup resulting from the exploitation of instruction-level parallelism is not sufficient. This is especially true in light of today's FPGA sizes, which provide an increasing amount of resources to the user.

To make full use of the available resources and in order to make HLS-based FPGA-designs more suitable for use in high-performance computing, HLS-tools recently started to make use of thread-level parallelism (TLP). Yet, the automatic parallelization and extraction of thread-level parallelism from input programs is difficult at best and impossible for a large range of problems.

User-annotations guiding the compiler in the parallelization and extraction of TLP from the input program are a possible and powerful alternative. A popular standard for such user-annotations is OpenMP [3]. The use of OpenMP directives for HLS allows to use existing programs as input, without the need for the insertion of special hardware annotation by the user. Furthermore, OpenMP's shared memory computation model is perfectly suited for FPGA-based hardware accelerators, because their architecture often combines the FPGA with an external memory, to which all threads running on the FPGA have access.

From the microarchitecture perspective, TLP can be exploited by carrying out each thread's computations on a dedicated hardware computing unit. However, this solution leaves plenty of room for improvement: Due to the latency caused by accesses to main memory, the computing unit will be idle a significant portion of the execution time, leaving the underlying hardware resources unused. By *interleaving* the execution of multiple threads on the computing units, these latencies can efficiently be bridged with other computations, improving the hardware resource utilization. Therefore it is the aim of this work to extend the existing Nymble hardware/software-co-compiler to synthesize hardware accelerators comprising computing units, which are able to interleave the execution of multiple threads in a time-sliced manner on the same FPGA area.

The rest of this work is structured as follows. Section II gives an overview of the existing approaches for OpenMP-based HLS. In Section III we describe the Nymble hardware-software-co-compiler, on which this work is based. Section IV presents our novel hardware execution model and explains all necessary modifications to the hardware datapath- and memory-architecture. In Section V we evaluate our approach, Section VI concludes and gives an outlook to future work.

II. Related work

There have been a number of approaches to integrate OpenMP into High-Level Synthesis, ranging from mere source-to-source transformations to complex hardware execution models featuring direct access to memory from the accel-

978-1-5386-3797-5/17/$31.00 ©2017 IEEE

erator. Similar to our approach, some of these approaches are based on OpenMP worksharing loops, indicated by `#pragma omp parallel for` in the input program.

Leow et al. [4] as well as Dziurzanski et al. [5] pursue a pure hardware approach and need to emulate shared memory by registers (Leow) and global signals (Dziurzanski), respectively, because their hardware implementation does not include a memory system with access to shared memory.

In their work, Cilardo et al. [6], [7] use an OpenMP-annotated input program as starting point for the synthesis of specialized hardware accelerators in a System-on-chip, combined with general-purpose processors. All elements of the resulting MPSoC have full access to the memory via the communication network of the SoC.

The integration of OpenMP support into the LegUp high-level synthesis system [8] by Choi et al. [9] features multiple levels of parallelism in hardware. The resulting system is composed of a MIPS processor and multiple hardware accelerators, which have direct access to memory via the interconnect.

All of the approaches presented above exploit thread-level parallelism with the concurrent execution on multiple hardware units. The set of iterations is partitioned onto multiple identical hardware kernels, which each conduct the computations of a single thread. However, in none of the these approaches multiple threads are executed on the same hardware datapath in an interleaved or concurrent fashion. Each distinct datapath is only running a single thread at a time and this 1:1-relationship between computing units and threads is maintained throughout the execution time. Only Choi et al. are able to pipeline execution of multiple threads in a limited number of cases. Our approach, in contrast, is intended to exploit thread-level parallelism in multiple ways, by distributing the computation across multiple computing units and, at the same time, running multiple threads on each computing unit in an interleaved manner.

Concurrent execution of multiple threads has been used in existing approaches [10], [11], [12]. In contrast to our work, these approaches do not use OpenMP as a starting point for high-level synthesis. The thread-level parallelism originates from threads explicitly managed by the user or from multiple instances of the same program running in distinct processes and requires significant effort from the user. Contrary to these approaches, our work allows users to incrementally modify the program by inserting OpenMP-pragmas, which manage thread-level parallelism. Moreover, our support for OpenMP provides the user with a clearly defined shared memory model and synchronization mechanisms.

Both approaches also integrate thread pipelining, which we do not use in our current implementation.

III. Nymble HW/SW-Co-Compiler

Our work is an extension of the Nymble hardware/software-co-compiler [13], which aims for the automatic synthesis of a large subset of `C` (including, e.g., pointer operations and irregular control flow in loops) for so-called *Adaptive Computer Systems (ACS)*, comprising an FPGA as *reconfigurable computing unit (RCU)* and a general-purpose processor.

Nymble allows the extraction of arbitrary sections of code, usually delimited by `pragmas` for synthesis, to a dedicated hardware accelerator. Furthermore, it also automatically provides all means necessary to interface between the software running on the GPP and the hardware accelerator.

For the high-level synthesis of the extracted hardware part in the Nymble compiler, the `pragma`-denoted section of the input program is represented as a hierarchical tree of so-called *control-memory-data-flow-graphs (CMDFG)*, with one CMDFG per loop in the hardware code region. A CMDFG does not only model a program's data flow between operations, but also incorporates the control flow of the input code. To this end, the control flow in the CMDFG is represented by conditional data flow and predicates, which are added to operations with side-effects (e.g., memory accesses). The tree of graphs contains one CMDFG per natural loop in the input program and reflects the nesting hierarchy of the loops in the program. During the execution of an inner loop, the graph of the surrounding loop does not continue its execution.

The work presented in [10], [11], called *Nymble-SMT*, is also based on the Nymble compilation framework. This approach uses loop-pipelining with a dynamically varying initiation interval. To ensure the minimal latency required to maintain loop-carried dependencies between iterations from the same threads, this implementation uses complex backpressure logic with tokens. In addition, dynamic operator multiplexing is required to resolve concurrent use of shared operators from different iterations of the same thread. As a consequence, the relative costs for the implementation of the Nymble-SMT multithreading model can be very high, especially for integer benchmarks which use only relatively simple operators. In this work we extend Nymble with an *alternative* multithreaded execution model (Section IV-A) and the corresponding controller implementation. To support thread-switching in the computing units, we augment Nymble's CMDFG-model with thread-context stores and implement an algorithm that inserts context stores only where necessary (Section IV-B). More details on the basic mapping from CMDFG to hardware datapath and the hardware/software-interface can be found in [13].

In contrast to Nymble-SMT, our model does not use loop pipelining to reduce the cost and complexity of the controller implementation. In order to support loop pipelining with a static II in our multithreaded model, we would need to add logic to synchronize stalls across all CDFGs of the graph-hierarchy. Besides that, the size of a thread's context, i.e. the elements that need to be stored and restored upon stall and reactivation, increases when applying loop pipelining.

Being based on OpenMP, our implementation incurs less overhead in the hardware/software-interface, as a single data-transfer and invocation is sufficient, compared to Nymble-SMT where each thread has to go through the HW/SW-interface. Contrary to Nymble-SMT, our approach supports the duplication of datapaths, which allows for the concurrent execution of multiple threads on distinct resources. Our approach

```
float a;
float x [SIZE];
float y [SIZE];

#pragma omp parallel for schedule(static)\
shared(a, x, y) private(i)\
num_threads(NUM_THREADS)
for(i=0; i<SIZE; i++){
    y[i] = a * x[i] + y[i];
}
```

Fig. 1. Example of an OpenMP worksharing loop.

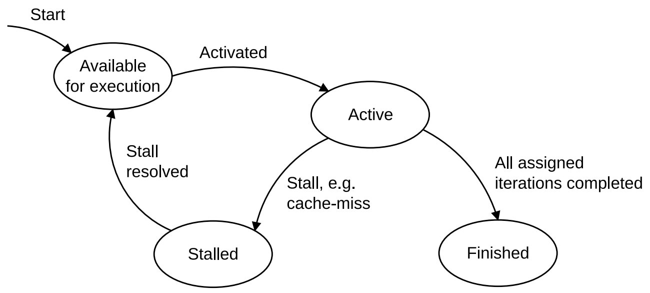

Fig. 2. Thread state diagram.

employs a fair round-robin hardware thread scheduler, whereas Nymble-SMT uses a static prioritization scheme which can lead to the starvation of threads. Furthermore, we implemented a new cache- and memory-infrastructure which allows for shared memory between threads and higher operation frequencies of the kernels compared to Nymble-SMT.

IV. Nymble-OMP

The aim of this work is to extend the existing implementation of Nymble to support the synthesis of *multithreaded* FPGA-accelerators based on OpenMP input programs. We therefore need to add support for processing of loop-nests marked as OpenMP worksharing loops by the `pragma omp parallel for` to Nymble. To this end, we implement the detection and extraction of OpenMP worksharing loops in Nymble's frontend. Nymble-OMP will then construct a hierarchical tree of CMDFGs for all loops in the annotated loop-nest.

The execution on the hardware datapaths synthesized from the resulting set of CMDFGs follows a novel execution model. The new execution model facilitates the interleaved execution of multiple threads on each hardware datapath, and the distribution of the input problem among multiple identical computing units running concurrently.

The following sections describe our novel execution model and necessary modifications to Nymble's datapath architecture in detail.

A. Execution Model

The basic idea of OpenMP worksharing loops, such as the one shown in the code snippet in Fig. 1, is to distribute the work specified by the loop body across a team of threads. From the set of iterations of the annotated loop, each thread is assigned a subset, whose size is dependent on the chosen distribution scheme.

On a typical multi-core CPU, the threads in the team will be distributed across the cores present in the CPU, where the execution of multiple threads will be interleaved in a time-sliced manner in case the number of threads specified by the user exceeds the number of cores.

In contrast, the existing OpenMP-based approaches presented in Section II synthesize a dedicated hardware computing unit per thread and carry out each thread's computation on distinct hardware resources without interleaving. The advantage of this model is that a thread can always be executed if it is ready to do so.

However, not all operations' latencies can be determined statically during the HLS-scheduling process, which is especially true for cached memory accesses that can cause a significant delay in case of a cache-miss. We refer to this kind of operation as *variable-latency operations* (VLO).

If at runtime a variable-latency operation does not finish within its assumed latency, this results in a stall of the hardware, i.e., the computation on the hardware computing unit is stopped until the VLO completes.

As a consequence, the employed hardware resources are not utilized to their full extent as they remain idle for a significant fraction of the execution time, e.g., if a data item must be retrieved from main memory. With a more efficient utilization of the hardware resources one could either achieve the same runtime performance with fewer hardware resources or improve the performance of the system with the same number of hardware resources.

Thus, the central goal of our execution model is to hide statically unpredictable latencies, such as latencies caused by cached memory accesses, with the execution of another thread on the same hardware computing unit.

In case the executing thread hits a stall, i.e., if the execution of a datapath operation takes longer than statically assumed for HLS-scheduling, we perform a thread-switch, replacing the currently active thread by some other available, execution-ready thread.

In order to be able to perform a thread-switch, we need to remove the 1:1-relationship between threads and computing units, as used by existing approaches. Instead of mapping each thread in the execution context to a dedicated computing unit, we assign a *set of threads* to each computing unit. The execution of all threads assigned to each computing unit is then interleaved in a time-sliced manner. In doing so, a thread-switch is only performed in case the currently executing thread encounters a stall.

At runtime, a thread can be in one of four different states, shown in Fig. 2, in our execution model. The single thread in state *Active* executes until it hits a stall, causing the state transition to state *Stalled*. As a consequence, one of the available threads is chosen as next thread to be executed and activated. We currently use a round-robin scheme to select the next thread from the set of threads in state *Available for execution*, but many other selection schemes, such as techniques used for hardware-only schedulers in GPUs [14], are conceivable and could be implemented in our controller architecture.

At the same time, the execution of the variable-latency operation continues in the background, until it is completed and the previously stalled thread becomes available again.

The cycle of execution, stall, and reactivation continues for each thread until the thread eventually completes all iterations it was assigned at the beginning of the hardware execution and changes to state *Finished*.

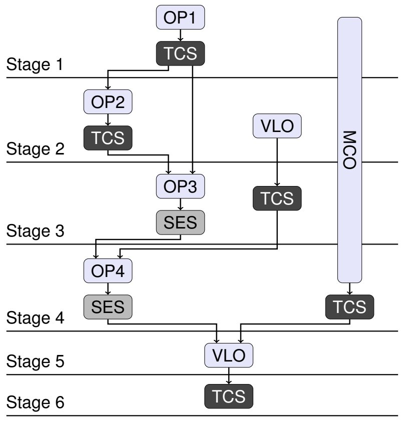

Fig. 3. Example for the two different kinds of data storage in the datapath (SES = single element storage, TCS = thread context storage, VLO = variable-latency operation, MCO = multi-cycle operation).

B. Datapath architecture and composition

Our multithreaded execution model requires the hardware datapaths in the computing unit to be able to store the thread contexts of all non-active threads, i.e. all threads stalled or available for execution, which have been assigned to that computing unit.

On a CPU, a thread context is typically defined by the values held in registers, which are stored and restored in case of a thread context switch. In Nymble's CMDFG model (cf. Section III), all data- and control-flow values are represented by data flow edges in the associated CMDFGs, so a thread's context is defined by the values that flow between the operators comprised in the hardware computing unit.

In order to carry values across clock cycle boundaries and to store thread context in the datapath, we insert two different types of intermediate storage elements into the hardware datapaths:

- Single element storage (SES) only holds a single data item, but require significantly fewer hardware resources (registers in particular) for their hardware realization.
- Thread context storage (TCS) is able to store a single data item *per thread* and can be indexed by a thread's unique ID to retrieve the value for a particular thread.

We use registers for the implementation of both kinds of storage elements, because TCS typically store 4-8 elements (max. 512 bit) and an implementation in block RAM would be a waste of memory resources.

Our datapaths are statically scheduled, i.e. each operation is assigned a time-step, called *stage* in the Nymble context. It would be a waste of resources to insert thread context storages in more locations than absolutely necessary. In general the insertion of a thread context storage to hold a value is required only if a thread switch can occur between the time the value is produced by an operation, and the time it is consumed by its latest (in the schedule) user. Expressed as a rule, an operation **OP** that produces a data value must be provided with a thread context storage if any of the stages in the interval $[\textbf{Start}(\textbf{OP}), \textbf{Last Use}(\textbf{OP})[$ contains a variable-latency operation.

The CMDFG-excerpt in Fig. 3 depicts some typical scenarios showcasing examples for applications of the rules described above. VLOs are generally provided with a thread context storage in the following stage. The multi-cycle operation (MCO, multiple clock cycles fixed latency) must be provided with a thread context storage as it spans across Stage 3, in which a thread switch can happen due to the VLO in Stage 2. Operation 1 (**OP1**) has been provided with a thread context storage because its last use (**OP3**), which is decisive for the rule, is again in Stage 3, where a thread switch can happen. In contrast to VLOs, thread-context storage linked to simple operations must be added to the same stage, as simple operators cannot hold the value. Operation 2 is also provided with a thread context storage due to the fact that the stage it has been scheduled in contains a VLO. For Operation 3, a simple single element storage is sufficient, as no thread switching can happen between its start in Stage 3 and its last use in Stage 4. The same holds true for Operation 4, because a potential thread switch caused by the variable-latency operation in Stage 5 will not occur in Stage 5, but in Stage 6.

With the thread-context storage included in the datapaths, the context of all currently non-active threads can be stored within the datapath, and restored when a thread's execution is resumed.

However, the interleaved execution of multiple threads on a computing unit is not the only way we can make use of the thread-level parallelism in OpenMP worksharing loops. Just as common in today's multicore CPUs, and similar to the previous approaches presented in Section II, we further exploit thread-level parallelism by including *multiple* identical computing units in the hardware accelerator and distributing the computations of the worksharing loop across these computing units. The computing units work in isolation from each other. Thread context is not shared and threads do not migrate between units.

C. Memory architecture

In order to make use of potential spatial and/or temporal locality exhibited by memory accesses, we insert a cache-infrastructure in front of the interface providing access to the external RAM on the device. We use a direct-mapped cache

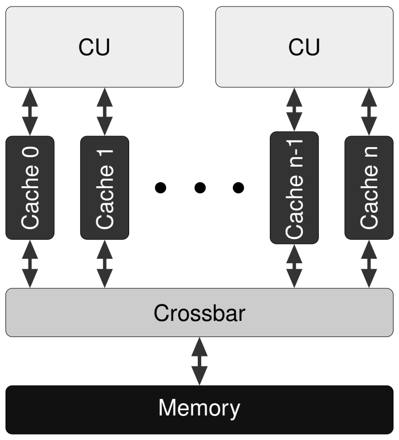

Fig. 4. Cache- and memory-architecture, n is the total number of threads across all computing units.

with a *write-through* strategy for writes to memory. Each cache uses 512 cache-lines with 16 32-bit words in each cache-line. A dedicated AXI4 master interface is used in the kernel interface for memory accesses. The cache IP we use in our implementation provides a single AXI4 slave interface, but does not yet support reordering of accesses, i.e., a cache miss will delay all subsequent accesses, even if they want to access data present in the cache.

If such a cache would be shared by multiple threads, accesses from different threads can delay each other, with a potentially negative impact on performance. In order to achieve maximum performance, we instantiate a dedicated cache *per thread*, allowing each thread to access memory independently.

This model, depicted in Fig. 4, is compliant to the OpenMP shared memory model, which allows for threads to have their own view of memory between synchronization points [3]. In addition to the shared memory, the OpenMP standard allows for each thread to have *thread-private* memory, e.g., for individual loop counters. However, in our CMDFG-model such thread-local values are only present as intermediate values in the CMDFGs and are thread-private by design. Therefore we do not need to make any arrangements for explicit thread-private memory in our memory architecture.

V. Evaluation

In this section we evaluate how our new execution model affects performance and the effects of the necessary changes to the datapath- and memory-architecture. We compare our new execution model to single-threaded execution and consider speedups as well as the costs of the hardware implementation of our execution model.

Similar to related work [9], [12], we use benchmarks with integer arithmetic. Our testcases are taken from the Adept benchmark suite [15]. Additionally we added an implementation of the sparse matrix-vector multiplication using the compressed row storage format. Furthermore, we also use floating-point implementations of each testcase for our evaluation. Compared to the integer versions, these testcases exhibit a different memory access behavior, i.e., memory accesses happen less frequently as the computation for each element takes longer. By including floating-point versions in our evaluation, we can study the effects of the memory access behavior in more detail. For our evaluation, vector-based benchmarks are set up to process 2000 element, matrix-based benchmarks work on matrices of 50×50 elements.

We compare our new multi-threaded execution model to an execution model where only a single thread is running on each computing unit. To this end, we examine four different configurations: A single-threaded configuration with two threads running on two computing units and three multi-threaded configurations with two computing units running two, three and four threads each (four, six, and eight threads total).

First, we evaluate the costs regarding hardware resources for the implementation of our execution model. This includes the necessary modifications to the datapath architecture (cf. Section IV-B) and the implementation of the controller and thread scheduler.

We use Vivado 2016.4 for the FPGA implementation, targeting a Virtex 7 (XC7VX690T) device on a VC709 board. The memory and host-connectivity-infrastructure is configured to run at 200 MHz, independent from the actual operation frequency of the hardware kernel.

The results for all four configurations are given in Table I. The number of DSP blocks used is unaffected by the choice of the execution model. As expected the number of occupied BRAM-slices increases with an increasing number of threads, as we need to spend more resources on the extra caches in the design used for the additional threads (cf. Section IV-C). The logic resources required for the implementation of the additional caches, our thread-switching logic and the thread-context storage also cause an increase in resource usage.

However, the biggest effect of our execution model with regard to the synthesis results is the limitation of the operation frequencies of the hardware kernels. The complexity of the combinatorial computations and indexing of thread-context storage causes the thread-switching logic to become critical for the achievable frequency and causes the frequency to decrease with an increasing number of threads interleaved on the computing unit. This effect has also been observed by Choi et al. in their work [9].

Despite the negative impact of the lower operation frequencies on performance, we are still able to gain a significant speedup over the single-threaded execution. The bar-plot in Fig. 5 shows the relative speedup over execution with a single thread per computing unit for each of our three multithreaded configurations. The execution times were obtained by executing the resulting FPGA accelerator designs on the VC709-board. The accelerated program is running on the host computer and data is transferred to the external RAM on the FPGA board using PCI Express. We use performance counters inside the kernels and combine them with the clock period achieved during FPGA implementation to calculate the runtime. The data transfer time from host to FPGA and back are not included in the runtime, as they are independent of the

TABLE I

FPGA IMPLEMENTATION RESULTS (POST-PLACE&ROUTE): VARYING NUMBER OF THREADS PER COMPUTING UNIT

Testcase	1 thread per computing unit						2 threads per computing unit					
	Exec.-time (μs)	Freq. (MHz.)	LUT(%)	FF(%)	BRAM(%)	DSP(%)	Exec.-time (μs)	Freq. (MHz.)	LUT(%)	FF(%)	BRAM(%)	DSP(%)
AXPY Int.	3815	175	17.86	12.86	31.12	0.17	2013	165	20.63	15.23	34.46	0.17
AXPY Float	3997	175	18.34	13.06	31.12	0.11	2854	157	21.62	16.17	34.66	0.11
Dense MV Int.	4785	162	19.17	13.09	31.12	0.50	3175	150	23.42	17.00	34.66	0.50
Dense MV Float	5714	169	19.47	13.28	31.12	0.44	4873	135	23.68	17.18	34.66	0.44
SPMV Int.	1803	168	19.58	13.10	31.12	0.17	1209	162	23.98	17.11	34.66	0.17
SPMV Float	2086	180	19.23	12.54	30.92	0.11	1733	140	23.47	16.53	34.46	0.11
Vector Scaling Int.	3434	175	17.62	12.81	31.12	0.17	1877	155	20.80	15.74	34.66	0.17
Vector Scaling Float	3302	188	17.79	12.88	31.12	0.11	1873	157	20.93	15.81	34.66	0.11

Testcase	3 threads per computing unit						4 threads per computing unit					
	Exec.-time (μs)	Freq. (MHz.)	LUT(%)	FF(%)	BRAM(%)	DSP(%)	Exec.-time (μs)	Freq. (MHz.)	LUT(%)	FF(%)	BRAM(%)	DSP(%)
AXPY Int.	1473	150	23.57	17.89	37.93	0.17	1372	126	27.16	21.29	41.67	0.17
AXPY Float	2617	133	24.56	18.84	38.13	0.11	2418	130	27.55	21.48	41.67	0.11
Dense MV Int.	2396	150	26.71	20.09	38.13	0.50	2312	132	30.60	23.17	41.67	0.50
Dense MV Float	3648	146	27.07	20.27	38.13	0.44	3635	134	30.76	23.35	41.67	0.44
SPMV Int.	1040	141	27.26	20.24	38.13	0.17	922	133	30.91	23.34	41.67	0.17
SPMV Float	1416	139	27.55	20.42	38.13	0.11	1227	139	31.33	23.51	41.67	0.11
Vector Scaling Int.	1291	150	22.86	17.54	37.93	0.17	1134	122	25.56	20.09	41.46	0.17
Vector Scaling Float	1443	150	23.66	18.38	38.13	0.11	1463	127	26.47	20.94	41.67	0.11

Number of LUTs, FFs, BRAM slices and DSP blocks are given as percentage for brevity. The total number of available resources are 433200, 866400, 1470 and 3600 respectively.

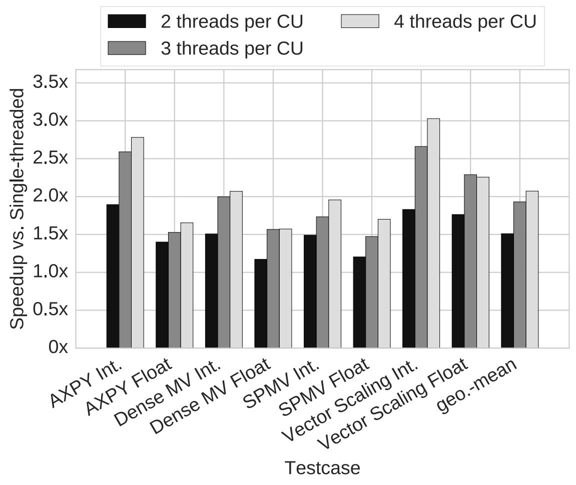

Fig. 5. Speedup compared to single threaded execution.

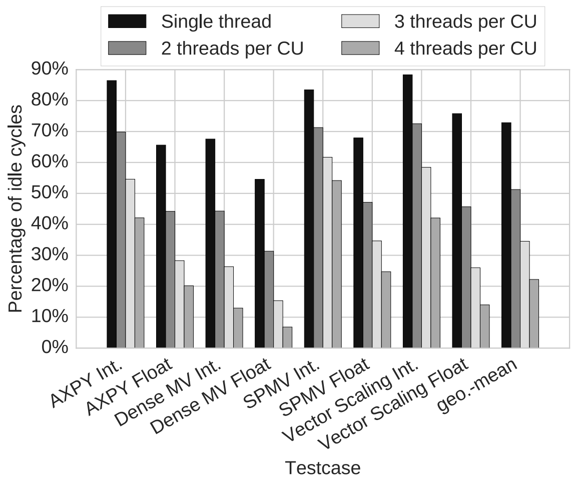

Fig. 6. Percentage of idle cycles encountered during execution.

execution model used.

We are able to achieve a significant speedup in all testcases, in some cases of more than a factor of 3x by interleaving the computation of multiple threads. The geo.-mean speedups are 1.51x, 1.93x and 2.07x, respectively.

The impact of our execution model is also visible in Fig. 6, which relates the number of idle cycles to the number of overall cycles. While the single-threaded computing unit lies idle for more than half of the execution time in almost all cases, our execution model improves the utilization of the computing units significantly. With an increasing number of threads assigned to each computing unit, stalls can be hidden more effectively with computations from other threads, resulting in a reduction of the idle-cycles by more than factor 8x (testcase *Dense MV Float*).

While there is a significant increase in speedup when going from two threads per CU to three threads per CU for most of the cases, this trend cannot always be sustained when increasing the number of threads per CU to four. In those cases the interleaving of three threads on each CU already leads to a high utilization (cf. Fig. 6), leaving less headroom for an extra thread. In combination with the reduced clock frequency, the addition of another thread is not always beneficial, as can be seen in testcase *Vector Scaling Float*.

As described earlier, the performance is also affected by the frequency of memory accesses. As visible in Fig. 5, we generally achieve a higher speedup for integer benchmarks, where memory access happen more frequently, and the time spent during stalls caused by memory accesses makes up for a greater portion of the execution time. Especially in these cases the strength of our execution model, the effective hiding of memory access latencies, comes into play.

In summary, the synthesis of hardware accelerators from OpenMP programs clearly benefits from the interleaving of threads on the computing units in our execution model. The number of idle cycles decreases significantly (up to factor 8x) and speedups of more than a factor of 3x can be achieved. The implementation of our execution model requires some additional resources, especially for the extra caches. The optimal number of threads is dependent on the input program and the memory access behavior exhibited by the program.

VI. Conclusion and Outlook

In this work, we presented an extension of the Nymble hardware/software-co-compiler to automatically generate multithreaded hardware accelerators from OpenMP worksharing loops. The novel execution model presented here as well as the modifications made to datapath- and memory-architecture allow to interleave the execution of multiple threads on one or more hardware computing units in a time-sliced manner.

We investigated the impact of our new execution model on hardware resource consumption and performance and compared our model to single-threaded execution. Our results show that the HLS of FPGA-based accelerators clearly benefits from our execution model. The interleaving of multiple threads on a computing unit allows to efficiently hide latencies caused by memory accesses. We were able to gain raw speedups of more than a factor of 3x with four-way multithreaded execution and improve the utilization of the computing units by more than a factor of 8x.

Our evaluation also showed that the optimal number of threads per computing unit is dependent on the input program and its memory access behavior. In the future, we want to automatically determine the best number of threads using compiler analyses. Besides that, we plan to add support for the OpenMP offloading constructs [16] to our compiler, allowing the user to clearly denote regions of code that should be extracted to a hardware accelerator and which and how data is mapped to the device memory. We also want to further investigate the impact of the memory- and cache-architecture on performance and how this infrastructure can be adapted to the input problem.

References

[1] A. Caulfield, E. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, October 2016.

[2] S. Muhlbach, M. Brunner, C. Roblee, and A. Koch, "Malcobox: Designing a 10 gb/s malware collection honeypot using reconfigurable technology," in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*. IEEE, 2010, pp. 592–595.

[3] "OpenMP Application Programming Interface - OpenMP Standard 4.5," Nov. 2015.

[4] Y. Leow, C. Ng, and W. Wong, "Generating hardware from OpenMP programs," in *IEEE International Conference on Field Programmable Technology, 2006. FPT 2006*, Dec. 2006, pp. 73–80.

[5] P. Dziurzanski, W. Bielecki, K. Trifunovic, and M. Kleszczonek, "A System for Transforming an ANSI C Code with OpenMP Directives into a SystemC Description." in *DDECS*, vol. 6, 2006, pp. 151–152.

[6] A. Cilardo, L. Gallo, and N. Mazzocca, "Design space exploration for high-level synthesis of multi-threaded applications," *Journal of Systems Architecture*, vol. 59, no. 10, Part D, pp. 1171–1183, Nov. 2013.

[7] A. Cilardo, L. Gallo, A. Mazzeo, and N. Mazzocca, "Efficient and scalable OpenMP-based system-level design," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, Mar. 2013, pp. 988–991.

[8] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 33–36.

[9] J. Choi, S. Brown, and J. Anderson, "From software threads to parallel hardware in high-level synthesis for FPGAs," in *2013 International Conference on Field-Programmable Technology (FPT)*, Dec. 2013, pp. 270–277.

[10] J. Huthmann, J. Oppermann, and A. Koch, "Automatic high-level synthesis of multi-threaded hardware accelerators," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2014, pp. 1–4.

[11] J. Huthmann and A. Koch, "Optimized high-level synthesis of SMT multi-threaded hardware accelerators," in *2015 International Conference on Field Programmable Technology (FPT)*, 2015, pp. 176–183.

[12] M. Tan, B. Liu, S. Dai, and Z. Zhang, "Multithreaded pipeline synthesis for data-parallel kernels," in *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 718–725. [Online]. Available: http://dl.acm.org/citation.cfm?id=2691365.2691510

[13] J. Huthmann, B. Liebig, J. Oppermann, and A. Koch, "Hardware/software co-compilation with the Nymble system," in *2013 8th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, Jul. 2013, pp. 1–8.

[14] S.-Y. Lee and C.-J. Wu, "CAWS: Criticality-aware Warp Scheduling for GPGPU Workloads," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. New York, NY, USA: ACM, 2014, pp. 175–186.

[15] Nick Johnson, Michèle Weiland, Trevor Carlson, and Sudarshan Balaji, "The Adept Benchmark Suite," 2015. [Online]. Available: http://www.adept-project.eu/images/whitepapers/p_590591_1446568406_Adept_Whitepaper_Benchmarks.pdf

[16] S. F. Antao, A. Bataev, A. C. Jacob, G.-T. Bercea, A. E. Eichenberger, G. Rokos, M. Martineau, T. Jin, G. Ozen, Z. Sura, T. Chen, H. Sung, C. Bertolli, and K. O'Brien, "Offloading support for openmp in clang and llvm," in *Proceedings of the Third Workshop on LLVM Compiler Infrastructure in HPC*, ser. LLVM-HPC '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 1–11.