Enabling FPGA and AI Engine Tasks in the HPX Programming Framework for Heterogeneous High-Performance Computing

Torben Kalkhof[0000−0002−4159−244X], Carsten Heinz[0000−0001−5927−4426], and Andreas Koch[0000−0002−1164−3082]

Embedded Systems and Applications Group, TU Darmstadt, Darmstadt, Germany
{kalkhof,heinz,koch}@esa.tu-darmstadt.de

Abstract. The increasing complexity of modern exascale computers, with a growing number of cores per node, poses a challenge to traditional programming models. To address this challenge, Asynchronous Many-Task (AMT) runtimes such as the C++-based HPX, divide computational problems into smaller tasks that are executed asynchronously by the runtime. By unifying the syntax and semantics of local and remote task execution, the scalability for distributed execution is enhanced. The asynchronous execution model conceals communication latency in distributed systems and eliminates global synchronization barriers, which improves the overall utilization of computation resources.
While HPX and other AMT runtimes often support GPUs, there is still a lack of support for other accelerators, such as FPGAs, or more coarse-grained AI processing elements such as AMD's AI Engines (AIE).
In this work, we extend the TaPaSCo framework so that TaPaSCo FPGA and AIE tasks can be transparently integrated into HPX applications. We show results for both microbenchmarks as well as the complete LULESH proxy HPC application to demonstrate this concept and evaluate the overheads. Both applications show that the combination of TaPaSCo and HPX can be efficiently used for cooperative computing between CPU software and FPGA / AIE hardware. Compared to CPU-only execution, we achieve a speedup of up to 2.4x in our stencil microbenchmark and a wall-clock speedup of 1.37x for the entire LULESH application, with 2.12x in the accelerated kernels itself. Our TaPaSCo / HPX integration is released as open-source.

Keywords: FPGA · task-based programming · HPC · AI engines

1 Introduction

The demand for computation power of modern applications is higher than ever, and we have arrived in the era of exascale computing [15]. On the one hand, such huge computation power is achieved by a steadily growing number of compute nodes. On the other hand, there is also a trend to increase the number of *CPU cores per node*. This results in a significant challenge for traditional

parallel programming models in High-Performance Computing (HPC), such as OpenMP and MPI, due to the increase in inter- and intra-node parallelism and concurrency.

In these models, computational problems are divided and statically scheduled to different cores and nodes, respectively. Synchronization barriers are primarily used to synchronize threads within a single node, e.g., at the end of parallel loops. Communication and data exchange between nodes is usually done by message passing, which may introduce implicit global synchronization barriers as well.

Load imbalance between threads and processes as well as communication latency may cause node starvation, as threads or processes block while waiting for messages or other threads to reach a barrier. This degrades the overall resource utilization. Hence, new programming models are desirable to use exascale supercomputers efficiently [14].

A promising approach are Asynchronous Many-Task (AMT) runtimes. In this programming model, the programmer divides the computational problem into small tasks and defines dependencies between these tasks, resulting in a dataflow graph.

The AMT runtime then asynchronously executes these tasks. By expressing data dependencies on a per task level, local as well as global synchronization barriers are reduced. Although AMT systems cannot *eliminate* communication latency, they can hide some or all of the latency. If a task is blocked due to waiting for a message, the runtime will suspend the current task and execute other tasks that are ready to compute.

Many AMT frameworks will have abstraction layers which unify local and remote execution of tasks to provide good scalability, improve load-balancing and, in turn, resource utilization on the intra- and inter-node level of distributed systems.

Besides the increase of cores and nodes, heterogeneity is another important trend in HPC. Due to their versatility, GPUs are most commonly applied in supercomputers and used for hardware acceleration in many applications. Hence, AMT runtimes often provide support for GPU kernels. However, other domain-specific accelerators continue gain in popularity, including more energy-efficient FPGAs or specialized AI hardware such as AMD Versal AI Engines (AIE). Unfortunately, AMT runtimes often lack support for these more specialized architectures.

In this work, we combine HPX [11], a C++-based AMT runtime targeting HPC and scientific applications, with TaPaSCo [9], a framework enabling the easy use of FPGA and AIE for task processing. By extending TaPaSCo, we enable a seamless integration of tasks running on FPGA- or AIE-based accelerators into HPX applications.

In our evaluation, we do not aim to show the highest possible accelerations by using FPGAs or AIEs. Instead, we examine whether HPX makes a suitable framework for the low-overhead integration of these computing elements in heterogeneous HPC applications.

To this end, we first use a stencil-based microbenchmark which allows us to easily measure the impact of different numbers of tasks and task sizes offloaded to an FPGA, including a comparison to an implementation solely based on the C++ standard library.

Then, we move on to a full application, namely the LULESH proxy HPC application [1], which we port to HPX and offload parts of the computation to an AIE-based accelerator. The performance results show that our TaPaSCo / HPX integration can achieve speedups in real-world applications, and that HPX enables this at *lower overheads* than the traditional approaches of the C++ library.

The paper is structured as follows. In Section 2, we provide an overview of HPX and TaPaSCo before discussing related work in Section 3. Afterward, we describe our implementation of the TaPaSCo integration with HPX in Section 4. Our benchmarks are introduced in Section 5 followed by presenting the evaluation results in Section 6. Section 7 then concludes the paper.

2 Background

In the following sections, we introduce both of the frameworks this work is based on, namely HPX and TaPaSCo.

2.1 HPX

HPX [11] is an AMT C++ library, and provides an API conforming to the C++ standard API for asynchronous execution.

HPX utilizes the concept of *futures* to execute asynchronous tasks. When we create an asynchronous task, e.g. using `hpx::async()`, the task is not executed immediately, and hence we cannot obtain the task's return value instantly. Instead, a future object is generated as a wrapper around the actual return value, and the current thread continues to execute. We can create an entire task graph using so-called *continuations* on these futures. Continuations define tasks that are executed after the preceding future becomes ready.

Finally, when we need to access the value of a future, the current thread is suspended until the corresponding task is executed and the value is computed.

HPX launches tasks as *lightweight* threads that are executed on a fixed number of operating system (OS) threads. Generally, one worker is assigned per CPU core. The HPX runtime completely manages the lightweight threads, leading to faster context switches when compared to multi-threading on the OS level. The task scheduler ensures good load balancing between the worker threads by applying techniques such as work stealing and sharing.

Further features of HPX include templates for parallel algorithms, remote task execution with a uniform API, and already support for GPUs using CUDA or HIP.

2.2 TaPaSCo

TaPaSCo [9] is a task-based framework providing simple and platform-independent integration of FPGAs into heterogeneous systems. It is made up of two parts: a toolflow that generates bitstreams for the FPGA and a runtime that interfaces with the accelerators. The framework supports various platforms ranging from Zynq SoCs (e.g. Ultra96) to PCIe-based data center cards (e.g. Alveo U280).

Hardware designs in TaPaSCo are built using Processing Elements (PEs), which are supplied by the user as HLS kernel or HDL-written IP with standard AXI4 interfaces. The infrastructure components, such as interrupt controllers and DMA engines, are automatically generated. Additionally, the TaPaSCo toolflow offers a design space exploration to optimize a hardware design for frequency or utilized area.

The TaPaSCo runtime consists of a device driver and a Rust user-space runtime. Also, a clear and platform-independent C++ API is provided for easy integration of the hardware accelerator into a host application. Users can pass their PE arguments to the `launch` call, and TaPaSCo handles requested data transfers to device-local memories automatically, as well as forwarding the arguments and launching the PE. By calling the returned `job_future`, the PE is released after it has finished and TaPaSCo performs data transfers back to host memory if necessary. Users can also manage device memory explicitly by using manual allocations and data transfers.

Furthermore, TaPaSCo provides a plugin system offering additional extensions such as 100G networking or shared virtual memory. Recently, the support for AMD Versal boards has been added, providing access to the AI engines. Moreover, a DMA streaming mode has been introduced, which avoids copies to device-local memory by streaming data directly into PEs.

3 Related Work

There are several other frameworks for AMT computation apart from HPX. The most common approach is to provide a library. While HPX uses the syntax of asynchronous function calls from the C++ standard API, other frameworks use different methods to define tasks. For instance, Charm++ [12] has special objects and interfaces, or StarPU [2] uses custom data structures called *codelets*.

In contrast, Chapel [4] and X10 [5] are domain specific task-based programming languages. Legion [3] provides both, a C++ API as well as the custom language Regent [16], following a more data-centric approach than HPX. In Regent, tasks operate on defined data regions, resulting in a well-defined data flow and execution graph.

OmpSs uses OpenMP-like pragmas and also supports FPGA tasks [6]. In contrast to this work, FPGA task compilation using HLS is completely integrated into the custom compiler, so non-HLS accelerators are not supported.

The stencil microbenchmark we employ below is also used by Grubel et al. [8] to evaluate the impact of different task sizes on the performance of HPX. Karlin

et al. [13] use LULESH to explore and compare different programming models excluding HPX, while Jin et al. [10] implement LULESH kernels using OpenCL HLS on an FPGA. However, Jin et al. only present the execution times of single kernels, without taking into account data movements between host and device. Additionally, it is unclear to us which program arguments they used in particular, preventing a direct comparison.

4 Implementation

In this section, we describe the required changes in TaPaSCo to enable an efficient integration into HPX.

TaPaSCo uses blocking calls to implement launching and releasing of PEs. This means that if currently no PE of the requested type is available to service a task, the calling thread is halted until a PE has finished the previous task and can be re-launched. The executing OS thread is suspended, and in a multi-threaded application, other threads can take over and be executed. The same happens when a PE should be released but has not finished executing yet.

In HPX, this would mean that the current worker thread is suspended and cannot execute other tasks, which goes against the underlying concept of HPX. Thus, non-blocking calls need to be added to TaPaSCo for launching and releasing PEs, that *always* return immediately, whether the PE could be launched or released or not.

These non-blocking calls are then used inside a wrapper that just suspends the current HPX lightweight thread, instead of the underlying OS thread, allowing the HPX scheduler to assign other tasks to this worker thread. The TaPaSCo task will be rescheduled until the PE is ready.

All the user has to do is call this wrapper function inside the HPX task, and pass the PE and required arguments to the wrapper function call.

In the launch call, TaPaSCo first allocates space in off-chip memory for input and output data, and then tries to acquire a PE for this job. During testing, we realized that this can lead to oversubscription of memory, since many tasks are created long in advance of their actual launches when using HPX. Hence, we change the order and first acquire a PE, *before* moving data to off-chip memory. In certain applications, it can be beneficial to coordinate the transfer of data in separate tasks. This allows for increased parallelism by overlapping data transfers and PE jobs. Additionally, HPX offers an `io_pool_executor` specially designed for I/O intensive tasks that may be utilized for these transfers.

5 Benchmarks

In this section, we introduce our stencil microbenchmark and LULESH case study which we use to evaluate our work. We describe the design of our accelerators, and discuss challenges we encountered while using FPGA and AIE tasks in HPX.

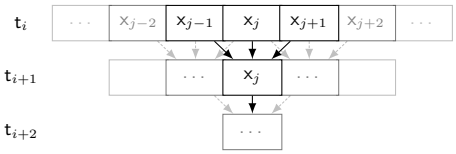

Fig. 1. Computation flow of the 1D-stencil microbenchmark.

5.1 Microbenchmark: 1D-Stencil

The stencil microbenchmark iteratively simulates heat diffusion in a ring over time given by the following formula

$$x_i(t_{j+1}) = x_i(t_j) + k\frac{dt}{dx^2}(x_{i-1}(t_j) - 2x_i(t_j) + x_{i+1}(t_j)) \tag{1}$$

where x_i denotes the i-th element in the array representing the ring, and t_j the current iteration. k, dt and dx are constants denoting the heat transfer coefficient, time step, and grid spacing. Figure 1 visualizes the computation flow. The array is then divided into equally sized partitions and one task is created to calculate one timestep for one partition each.

We implement an HLS kernel using Vitis HLS to offload tasks to the FPGA. It is fully pipelined and computes eight values in parallel. In total, we put 16 of these PEs onto the Alveo U280 and connect them to the 16 ports of the left HBM stack available on this UltraScale+ device. The PEs run at 450 MHz so that frequency and datawidth match with the HBM ports. At the beginning of the benchmark, we offload a given number of partitions to the on-device HBM where they stay for the entire benchmark. All tasks working on the offloaded partitions are then executed by the PEs, avoiding costly data transfers in between iterations. Only values on the partition boundaries must be exchanged as the neighboring partitions require them in the next iteration.

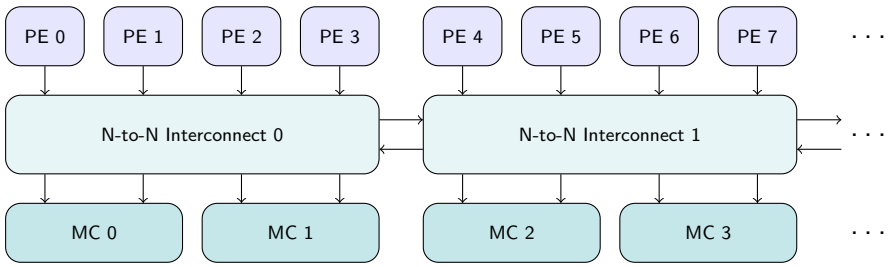

Fig. 2. Structure of the built-in crossbar between user PEs and HBM memory controllers (MC).

To exploit the high bandwidth of HBM, it is crucial to allow as many parallel memory accesses as possible. The left stack of the HBM on the Alveo U280

has eight memory controllers with 2 pseudo-channels each, with one AXI port per pseudo-channel. Although we do not use the entire 4 GB of the HBM stack, we distribute the partitions equally over the stack to improve parallelism. Located between the AXI ports and pseudo-channels of the memory controller, there is a built-in crossbar to allow access from each AXI port to the entire stack, as depicted in Figure 2. However, the crossbar only provides direct n-to-n-interconnects for *four* ports and pseudo-channels each, which we call a *group*. Inter-group accesses become expensive, since they may congest the single connection crossing group boundaries. Thus, we schedule tasks on the PEs in a way that avoids interference on the inter-group interconnects.

Since launching a PE has potentially much higher overhead, due to the PCIe latency, than scheduling a lightweight thread on the CPU, we implement two strategies for partitioning tasks on the FPGA. In the first variant, we have *equally sized* partitions on CPU and FPGA and vary the number of partitions that are offloaded to the PEs. Secondly, we also vary the total amount of data offloaded to the FPGA, but always split it into 16 partitions *independent of the partition size* on the host. In this approach, there is always exactly one task per PE in every iteration.

To evaluate the benefits of using TaPaSCo with light-weight threads in HPX compared to OS multi-threading, we implement a non-HPX baseline, only using the asynchronous functions provided by the C++ standard library and original blocking TaPaSCo calls. Here, an OS thread is created for each task.

5.2 Case Study: LULESH

The Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) proxy application models hydrodynamics codes by solving the Sedov blast wave problem. As a proxy application, LULESH is highly simplified while maintaining the typical structure of similar scientific applications allowing benchmarking of a real-world problem without physical background.

In LULESH, the physical problem space is divided into a hexahedral mesh on which the computation is performed. The underlying algorithm is a Lagrange leapfrog algorithm, consisting of three phases. During the first phase, the force on each node of the mesh at the current timestep is calculated. Based on this, acceleration, velocity, and position of the nodes are updated.

In contrast to the nodal-based first phase, the second part of the algorithm updates the properties of each mesh element, i.e. each hexahedron in the mesh. Mainly, pressure, internal energy, and relative volume of the elements are calculated as well as the sound speed in each element. To simulate different materials, the mesh is split into multiple regions. Although the material properties are the same in LULESH, each region must be handled in a separate loop. Additionally, the computational intensity of the regions is varied, which is simulated by simply repeating the computation of the element quantities for some of the regions.

The last phase calculates the timestep of the next iteration. However, its runtime is negligible compared to the other two phases.

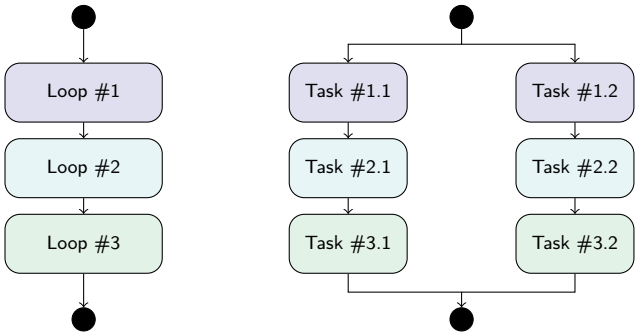

Fig. 3. Comparison of loop structure (left) to our task structure (right). Loops impose a synchronization barrier, while tasks only depend on their respective predecessors and require a single synchronization barrier at the end.

The OpenMP reference implementation consists of 45 parallel-for loops performing the different computation steps after each other. An existing HPX implementation of LULESH takes the approach of simply converting all OpenMP loops to HPX-provided parallel for-loops [17].

However, the performance of that implementation is shown to be *worse* than the OpenMP reference implementation in our measurements. This non-optimal HPX implementation does not remove any synchronization barriers, and due to its "too static" scheduling, there is no load imbalance to be expected between OpenMP threads, which in turn could be exploited using the AMT mechanisms of code better matching the HPX paradigm.

To obtain a higher-quality baseline, we thus carefully created our own, better optimized software-only LULESH implementation using HPX. To this end, we decided to manually decompose the computation into tasks and try to remove barriers as far as possible.

We maintain the partitioning in kernels from the reference implementation in large parts, but heavily use continuations to chain the tasks performing computations on the same subset of data in consecutive kernels, as illustrated in Figure 3.

Furthermore, we launch tasks of completely independent kernels in parallel. This way we can reduce the number of synchronization barriers to six per iteration. After all optimizations, our CPU-based application is even slightly faster than the original OpenMP reference implementation of LULESH.

Due to expensive data movements to and from FPGA off-chip memory in every iteration of the leapfrog algorithm, we need to offload kernels in which we do multiple calculations on the same input data. At the same time, the offloaded part should be *independent* of other tasks, so that we can still run calculations on the CPU in parallel.

Based on this, we choose the update of the element properties in the most expensive region during phase two. Since this computation is floating point intensive, we use the AMD Versal AI Engines (AIE) on the VCK5000.

The AIE consists of 400 tiles with VLIW architecture and SIMD vector processors including single-precision floating point pipelines. All tiles are connected via a streaming interconnect, and neighboring tiles can also use buffers in local memory to exchange data.

We design specialized hardware on the Programmable Logic (PL) part of the Versal device to efficiently feed the AIE with data. To this end, we implement a custom PE streaming input data from off-chip memory into the AIE and output data vice versa. The entire computation is repeated as many times as it would been executed on the host.

One challenge in implementing the computation graph for the AIE is that the structure of the LULESH computation does not perfectly match the streaming-based architecture of the AIE, e.g., many input and intermediate variables are reused in multiple steps of the calculation. This means they have to be buffered in between.

Input variables are buffered in FIFOs in the PL and streamed into the AIE *again* at a later stage using an additional port. However, intermediate variables are buffered in the AIE by dummy kernels, which just forward the data from one buffer to another.

6 Evaluation

As described in the Introduction, we begin our evaluation with the more focused 1D-stencil microbenchmark, before moving on to a full HPC code, namely the LULESH hydrodynamics proxy application.

All tests were run on an AMD Epyc 7443 24-core processor equipped with 256 GB RAM. The stencil benchmark is run on an Alveo U280 clocked at 450 MHz and connected with PCIe 3.0 x16 to the host. Our LULESH accelerator runs on a VCK5000 connected via PCIe 4.0 x8. While the AIE run at 1.25 GHz, the custom data streaming engine we implemented in the Versal PL part is clocked with 200 MHz, which is sufficient to completely saturate the AIE bandwidth by using multiple streaming ports in parallel.

We use HPX v1.9.0 with jemalloc. All benchmarks are compiled with GCC 11.2.1 and `-DCMAKE_BUILD_TYPE=Release`.

6.1 Microbenchmark: 1D-Stencil

The presented numbers of our stencil microbenchmark are averaged over 100 runs. For the two smallest partition sizes of the non-HPX version, we reduced the number of runs to 50 due to long execution times. Each run calculates 300 timesteps of the heat distribution on an array with $256 \cdot 2^{20}$ single-precision floating point numbers (1 GB). The absolute execution times are shown in Figure 4, depending on the partition size and ratio of offloaded partitions to the FPGA.

We achieve faster computation times through collaborative computing using CPU and FPGA in all our measurement series. The shortest execution time of 3 s is achieved using HPX with a partition size of 4 MB on the host and 16 hardware

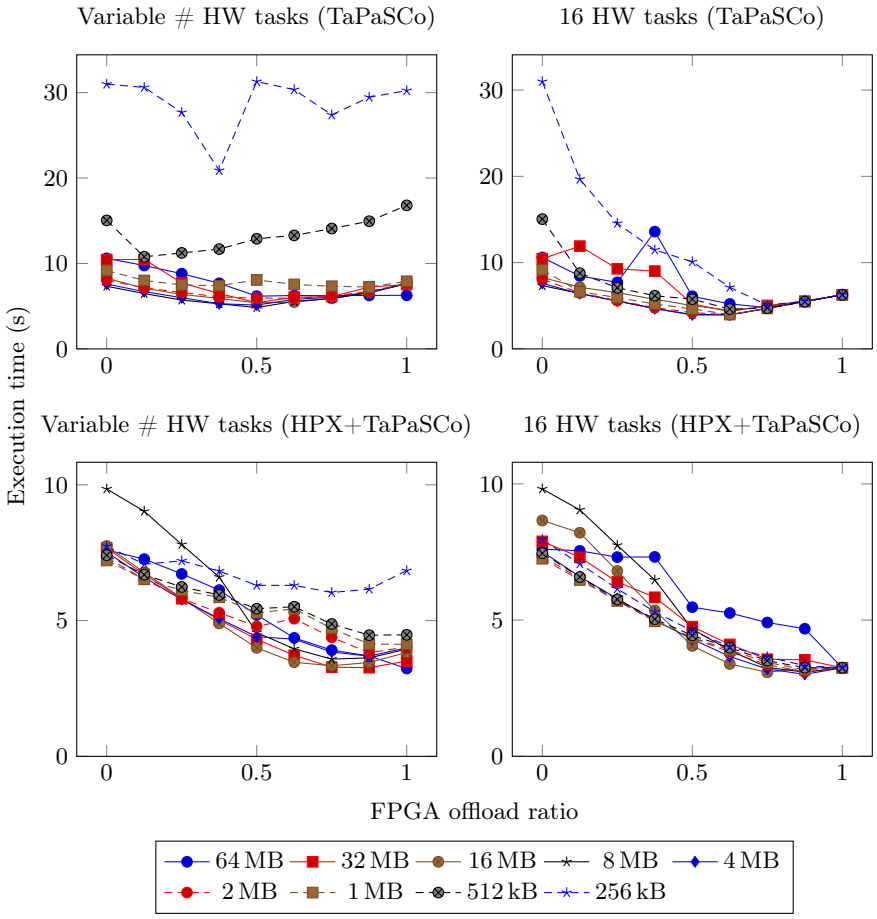

Fig. 4. Absolute execution times of the stencil microbenchmark dependent on partition size and offload ratio of partitions to the FPGA. The upper two graphs show results for the non-HPX variant using the standard C++ library, the lower graphs contain the results of the TaPaSCo / HPX integration. In the left two graphs, partition sizes on FPGA and host are equal, thus the number of offloaded tasks is variable. On the right, the offloaded calculations are always split into 16 partitions.

tasks that account for 87.5 % of the calculations. This results in a speedup of 2.4x compared to the fastest software-only computation using HPX with 1024 partitions of 1 MB each, which takes 7.22 s.

When comparing the upper and lower plots in Figure 4 to each other, you can see that the implementation using HPX, shown in the lower plots, is faster for all run configurations. Scheduling, suspending, and switching threads is much more efficient in HPX's lightweight threading model compared to OS threads. This is especially true when dealing with hundreds or thousands of concurrent threads, as in this benchmark. Due to this increased management overhead, the implementation without HPX cannot benefit from more offloaded tasks and reaches minimal runtimes around an offload ratio of 0.5. In contrast, the HPX implementation performs best if between 75 % to 87.5 % of the calculation is offloaded to the FPGA.

The performance of parallel computing on CPUs benefits from fine-grained parallelism due to smaller partitions to a certain degree. However, for FPGA tasks, it is better to keep the number of tasks low to improve execution times. The right plots in Figure 4 show that the overall execution time is shortened by setting the number of tasks running on hardware to 16 for each iteration and resizing the partition size accordingly to the offload ratio. This is particularly true for the measurement series with partition sizes smaller than 16 MB.

Each PE launch involves PCIe latency during writing control registers and handling the PE interrupt. Additionally, the threads managing the TaPaSCo tasks on the host side must be scheduled, which introduces overhead, although no actual computation is performed on the CPU.

6.2 Case Study: LULESH

We evaluate our LULESH implementation with three different problem sizes and three different numbers of regions each. The problem size specifies the number of elements in each mesh dimension. In addition to the default size of 45, we evaluate with 60 and 75 elements. We use the standard configuration for the region costs: half of the regions are computed once or twice respectively, but on one region the calculation is repeated 20 times. The presented numbers are the mean over 100 runs with 200 iterations of the Lagrange leapfrog algorithm each.

Note that we have very carefully optimized our HPX software-only baseline, making it even faster than the OpenMP reference code. The respective speedup factors are shown in Figure 5 while Table 1 lists the corresponding absolute numbers. Compared to that reference code, the wall-clock speedups of our AIE-accelerated version would be even higher than reported below. However, as our focus was on examining the overheads of the HPX / TaPaSCo integration, instead of examining absolute performance gains, we use our optimized HPX software code as baseline for our measurements.

Figure 6 illustrates how the cooperative execution of TaPaSCo and HPX has improved the total execution time and the execution time of the AIE-accelerated part compared to our software-only implementation. The corresponding absolute numbers can be found in Table 2.

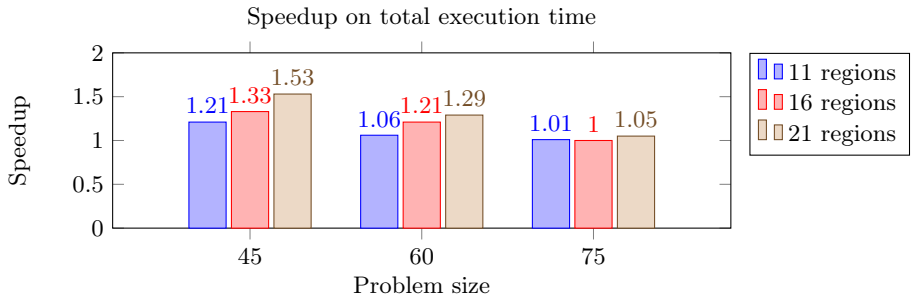

Fig. 5. Speedup of our software-only LULESH implementation using HPX compared to the OpenMP reference implementation.

Table 1. Absolute runtimes of our software-only LULESH implementation using HPX and the reference OpenMP implementation.

Problem size	45		60		75	
Runtime	HPX	OpenMP	HPX	OpenMP	HPX	OpenMP
11 regions	1.663 s	2.018 s	3.074 s	3.246 s	4.748 s	4.785 s
16 regions	1.574 s	2.092 s	2.765 s	3.341 s	4.882 s	4.904 s
21 regions	1.422 s	2.180 s	2.697 s	3.476 s	4.816 s	5.062 s

For some instances, we use different task sizes for the software-only and AIE-accelerated versions and report the optimal runtimes for each. The chosen task size can affect other parts of the application, not just the part accelerated by AIE, as evidenced by the numbers for software-only execution and a problem size of 75. Despite the differences in the lower table, the total runtimes are very similar. To indicate the limited comparability in the lower table, we have underlined the respective numbers.

The smallest problem size of 45 provides the best speedups for all three numbers of regions in our AIE-accelerated HPX implementation when compared to the HPX baseline. The highest speedup achieved is 1.37x on the total execution time and 2.12x on the accelerated part with 16 regions. As the problem size increases, the speedups decrease for 11 and 16 regions. For 21 regions, the speedup for a problem size of 60 is lower than for 45, but rises to 1.25x on the total execution time and 1.9x on the accelerated part for the largest problem size.

To understand the speedup, it is important to examine the absolute numbers of the accelerated part of the application. These numbers are listed in the lower table in Table 2. In the AIE-accelerated implementation, the runtime of this part is mainly influenced by the calculation of the most expensive region, which is repeated 20 times. The computation of this region on the AIE takes longer than processing all other regions on the CPU in parallel. However, as the number of regions increases, the most time-consuming region becomes smaller, resulting in reduced computation time for both the specific region and the total runtime.

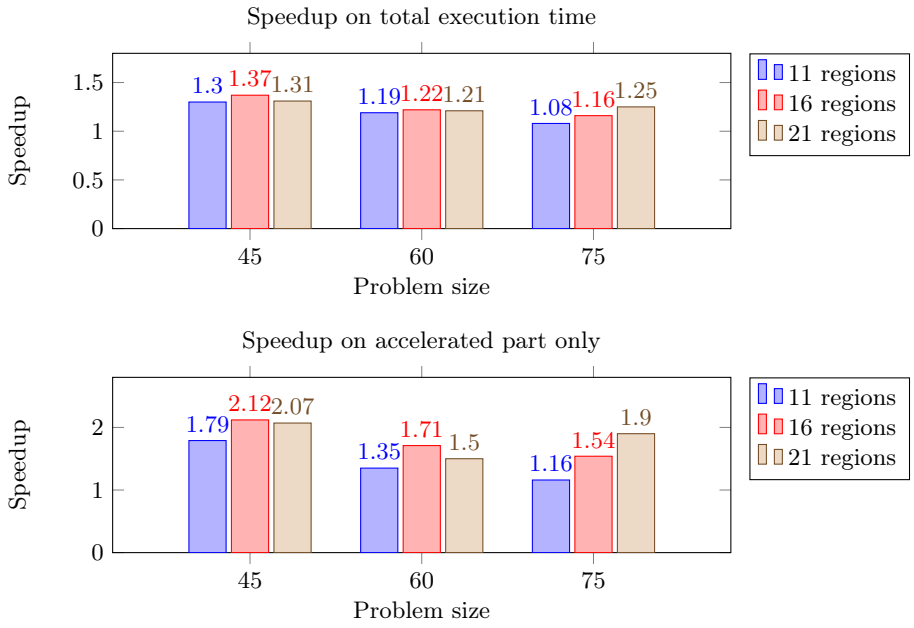

Fig. 6. Speedup of the HPX LULESH implementation running cooperatively on CPU and AIE compared to running in software only. The speedup is given on the total execution time as well as the execution time of the accelerated part only.

Table 2. Absolute execution times of the HPX LULESH implementation running cooperatively on CPU and AIE compared to running in software only. Numbers are given for the total execution time and the time of the accelerated part only. Underlined numbers indicate that different task sizes are used for the software-only and AIE-accelerated variants.

Total execution time

Problem size	45		60		75	
Runtime	HPX	HPX+AIE	HPX	HPX+AIE	HPX	HPX+AIE
11 regions	1.663 s	1.277 s	3.074 s	2.575 s	4.749 s	4.414 s
16 regions	1.574 s	1.149 s	2.765 s	2.262 s	4.882 s	4.196 s
21 regions	1.422 s	1.088 s	2.697 s	2.225 s	4.816 s	3.860 s

Execution time of accelerated part only

Problem size	45		60		75	
Runtime	HPX	HPX+AIE	HPX	HPX+AIE	HPX	HPX+AIE
11 regions	0.910 s	0.510 s	1.393 s	1.028 s	1.887 s	1.623 s
16 regions	0.819 s	0.387 s	1.228 s	0.718 s	2.138 s	1.383 s
21 regions	0.667 s	0.322 s	1.016 s	0.675 s	2.070 s	1.092 s

The heuristic determining the sizes of the regions includes some randomness, though, and it is not directly proportional to either the number of regions or the problem size. This is reflected in the numbers. For instance, when the problem size is 60, the most expensive region is nearly halved during the step from 11 to 16 regions, but then only shrinks slightly for 21 regions.

In the software-only implementation, the runtime depends mainly on how parallelizable the AIE-accelerated part is. When the problem size is 45, the most expensive region dominates the runtime and is split into very few tasks. This limited parallelizability actually allows for the highest speedups by our AIE accelerator. However, if we increase the problem size to 60, all regions become larger, which allows for more tasks per region and improves parallelism. Nonetheless, the most expensive region still dominates the runtime.

Conversely, when we increase the problem size to 75, the total runtime does not change significantly when varying the number of regions. This is because the overall number of tasks for all regions is large enough to hide the higher computation cost of the expensive region. However, the runtime of the AIE-accelerated variant decreases with the increasing number of regions in this scenario as well, enabling higher speedups with a larger number of regions.

7 Conclusion

This work enables the transparent integration of TaPaSCo tasks executed on FPGA and AIE accelerators in HPX. The main benefit of this integration is that applications can leverage the asynchronous and lightweight execution model on the CPU, while being able to include custom hardware accelerators on FPGA or AIE at the same time.

We are able to achieve up to 2.4x speedup in our stencil-based microbenchmark by offloading tasks to an FPGA, as compared to executing all tasks on a CPU. Our comparison to an implementation using the C++ standard library for concurrency, demonstrates that the lightweight HPX tasks, based on lightweight threads in turn, significantly reduce overhead compared to multi-threading using OS-level threads, especially when launching thousands of tasks in parallel.

Furthermore, we implement the LULESH proxy HPC application in HPX and use the AMD Versal AIE to accelerate key parts. Our results showed that we were able to achieve speedups of up to 1.37x on the total wall-clock time, which serves as a proof of the usability of the HPX + TaPaSCo approach for real-world applications.

TaPaSCo is already available as open-source on GitHub, where we will also provide our HPX integration [7].

Acknowledgments. This research was funded by the German Federal Ministry for Education and Research (BMBF) with the funding ID 01 IS 21007 B.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory. Tech. Rep. LLNL-TR-490254
2. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. CCPE - Concurrency and Computation: Practice and Experience, Euro-Par 2009 (Feb 2011)
3. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: Expressing locality and independence with logical regions. In: 2012 International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–11 (Nov 2012)
4. Chamberlain, B., Callahan, D., Zima, H.: Parallel Programmability and the Chapel Language. The International Journal of High Performance Computing Applications **21**(3), 291–312 (Aug 2007)
5. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., Von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. ACM SIGPLAN Notices **40**(10), 519–538 (Oct 2005)
6. Deharo, J.M., Bosch, J., Filgueras, A., Vidal, M., Jimenez-Gonzalez, D., Alvarez, C., Martorell, X., Ayguade, E., Labarta, J.: OmpSs@FPGA framework for high performance FPGA computing. IEEE Transactions on Computers pp. 1–1 (2021)
7. Embedded Systems and Applications Group, TU Darmstadt: Esa group on Github, https://github.com/esa-tu-darmstadt
8. Grubel, P., Kaiser, H., Cook, J., Serio, A.: The Performance Implication of Task Size for Applications on the HPX Runtime System. In: 2015 IEEE International Conference on Cluster Computing (2015)
9. Heinz, C., Hofmann, J., Korinth, J., Sommer, L., Weber, L., Koch, A.: The tapasco open-source toolflow. Journal of Signal Processing Systems (May 2021)
10. Jin, Z., Finkel, H.: Evaluating LULESH Kernels on OpenCL FPGA. In: Applied Reconfigurable Computing. Springer International Publishing, Cham (2019)
11. Kaiser, H., Diehl, P., Lemoine, A.S., Lelbach, B.A., Amini, P., Berge, A., Biddiscombe, J., Brandt, S.R., Gupta, N., Heller, T., Huck, K., Khatami, Z., Kheirkhahan, A., Reverdell, A., Shirzad, S., Simberg, M., Wagle, B., Wei, W., Zhang, T.: Hpx - the c++ standard library for parallelism and concurrency. Journal of Open Source Software **5**(53), 2352 (2020)
12. Kale, L., Krishnan, S.: Charm++: A portable concurrent object oriented system based on c++. pp. 91–108. Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA (Oct 1993)
13. Karlin, I., Bhatele, A., Keasler, J., Chamberlain, B.L., Cohen, J., Devito, Z., Haque, R., Laney, D., Luke, E., Wang, F., Richards, D., Schulz, M., Still, C.H.: Exploring traditional and emerging parallel programming models using a proxy application. In: IEEE 27th Int. Symposium on Parallel and Distributed Processing (2013)
14. Kogge, P., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hiller, J., Keckler, S., Klein, D., Lucas, R.: Exascale computing study: Technology challenges in achieving exascale systems. DARPA IPTO, Technical Representative **15** (01 2008)
15. RIKEN: Japan's fugaku gains title as world's fastest supercomputer, https://www.riken.jp/en/news_pubs/news/2020/20200623_1/
16. Slaughter, E., Lee, W., Treichler, S., Bauer, M., Aiken, A.: Regent: a high-productivity programming language for HPC with logical regions. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–12. SC '15 (Nov 2015)
17. Wei, W.: lulesh-hpx on Github, https://github.com/weilewei/lulesh-hpx