Extending High-Level Synthesis for Task-Parallel Programs

Yuze Chi, Licheng Guo, Young-kyu Choi, Jie Wang, Jason Cong
{chiyuze,lcguo,ykchoi,jiewang,cong}@cs.ucla.edu
University of California, Los Angeles

ABSTRACT

C/C++/OpenCL-based high-level synthesis (HLS) becomes more and more popular for field-programmable gate array (FPGA) accelerators in many application domains in recent years, thanks to its competitive quality of result (QoR) and short development cycle compared with the traditional register-transfer level (RTL) design approach. Yet, limited by the sequential C semantics, it remains challenging to adopt the same highly productive high-level programming approach in many other application domains, where coarse-grained tasks run in parallel and communicate with each other at a fine-grained level. While current HLS tools support task-parallel programs, the productivity is greatly limited in the code development, correctness verification, and QoR tuning cycles, due to the poor programmability, restricted software simulation, and slow code generation, respectively. Such limited productivity often defeats the purpose of HLS and hinder programmers from adopting HLS for task-parallel FPGA accelerators.

In this paper, we extend the HLS C++ language and present a fully automated framework with programmer-friendly interfaces, universal software simulation, and fast code generation to overcome these limitations. Experimental results based on a wide range of real-world task-parallel programs show that, on average, the lines of kernel and host code are reduced by 22% and 51%, respectively, which considerably improves the programmability. The correctness verification and the iterative QoR tuning cycles are both greatly accelerated by 3.2× and 6.8×, respectively.

1 INTRODUCTION

C/C++/OpenCL-based high-level synthesis (HLS) [15] has been adopted rapidly by both the academia and the industry for programming field-programmable gate array (FPGA) accelerators in many application domains, e.g., machine learning [16, 62], scientific computing [40, 68], and image processing [10, 52]. Compared with the traditional register-transfer level (RTL) paradigm (Figure 1) where programmers often spend tens of minutes just to verify the correctness of a code modification, with HLS, programmers can follow a rapid development cycle (Figure 2). Programmers can write code in C and leverage fast software simulation to verify the functional correctness. Such a correctness verification cycle can take as few as just 1 second, allowing functionalities to be iterated at a fast pace. Once the HLS code is functionally correct, programmers can then generate RTL code, evaluate the quality of result (QoR) based on the generated performance and resource reports, and modify the HLS code accordingly. Such a QoR tuning cycle typically takes only a few minutes. Thanks to the advances in HLS scheduling algorithms [6, 7, 19, 28, 30] and timing optimizations [5, 27, 35], HLS can not only shorten the development cycle, but also generate programs that are often competitive in cycle count [17], and more recently in clock frequency as well [27]. Moreover, FPGA vendors provide host drivers and communication interfaces for kernels designed in HLS [31, 63], further reducing programmers' burden to integrate and offload workload to FPGA accelerators.

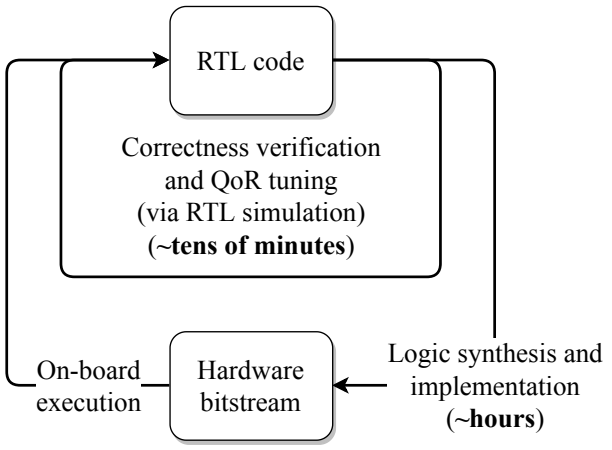

Figure 1: FPGA accelerator development flow without HLS. Programmers often spend tens of minutes after code modification to evaluate the correctness and quality of result.

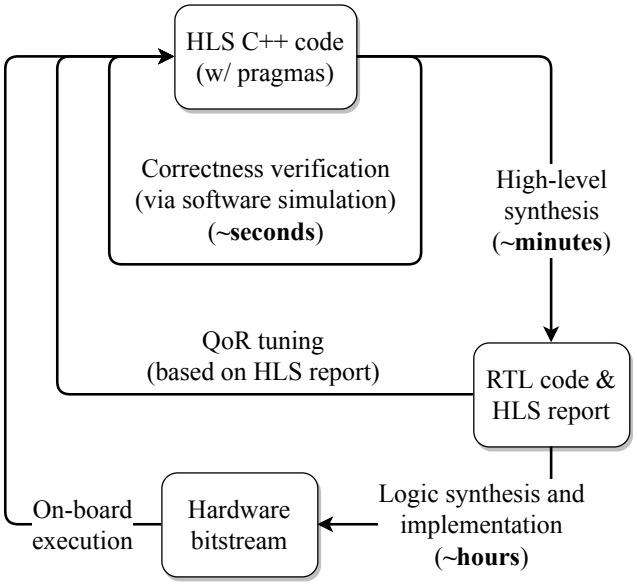

Figure 2: FPGA accelerator development flow with HLS. Programmers spend seconds after code modification to verify the correctness. Quality of result can usually be obtained in less than 10 minutes from the HLS report.

However, not all programs are created equal for HLS. Data-parallel programs can be easily programmed following the sequential C semantics with HLS-specific compiler directives (i.e.,

"pragma"). The HLS compiler can then leverage the directives to extract the parallelism automatically via static dependency analysis. This enables such applications to be quickly designed and iterated in the fast correctness verification cycle and QoR tuning cycle, as shown in Figure 2. However, task-parallel programs are not supported by the native C semantics, and the productivity provided by current HLS tools are greatly limited for the following reasons:

- *Poor programmability.* Due to the lack of convenient application programming interfaces (API), programmers are often forced to write more code than they have to. For example, a network switch needs to forward packets based on their content and the availability of output ports. Without an API to read packets without consuming them (a.k.a., "peek") from the ports, programmers have to manually and carefully create a buffer and maintain a small state machine to keep track of incoming packets. This not only elongates the development cycle, but also makes the code error-prone.
- *Restricted software simulation.* As the key to fast correctness verification, software simulation is not always available to task-parallel programs. For example, Vivado HLS does not support debugging Cannon's algorithm [44] via software simulation because of the existence of feedback loops in data paths, while Intel OpenCL does not support more than 256 concurrent kernels [31] in software simulation. Lack of fast software simulation forces programmers to resort to RTL simulation for correctness verification, significantly elongating the development cycle.
- *Slow code generation.* We found that current HLS compilers view task-parallel code as a monolithic design and processes each instance of the same task as if they are different. For designs that instantiate the same task multiple times (e.g., in a systolic array), this leads to repetitive compilation on each task and unnecessarily slows down code generation. One may argue that programmers can manually synthesize tasks separately and instantiate them in RTL, but doing so requires debugging RTL code, which is time-consuming and error-prone. We think such processes should be automated by the compiler.

Limited productivity for task-parallel programs significantly elongates the development cycles and undermines the benefits brought by HLS. One may argue that programmers should always go for data-parallel implementations when designing FPGA accelerators using HLS, but data-parallelism may be inherently limited, for example, in applications involving graphs. Moreover, researches show that even for data-parallel applications like neural networks [16] and stencil computation [10], task-parallel implementations show better scalability and higher frequency than their data-parallel counterparts due to the localized communication pattern [18]. In fact, at least 6 papers [24, 34, 46, 55, 59, 64] among the 28 research papers published in the ACM FPGA 2020 conference use task-parallel implementation with HLS, and another 3 papers [4, 50, 65] use RTL implementation that would have required task-parallel implementation if written in HLS.

In this paper, we extend the HLS C++ language and present our framework, TAPA (**ta**sk-**pa**rallel), as a solution to the aforementioned limitations of HLS productivity. Our contributions include:

- **Convenient programming interfaces**: We show that, with peeking and transactions added to the programming interfaces, TAPA can be used to program task-parallel kernels with 22% reduction in lines of code (LoC) on average. By unifying the interface used for the kernel and host, TAPA further reduces the LoC on the host side by 51% on average.
- **Universal software simulation**: We demonstrate that our proposed simulator can correctly simulate task-parallel programs that existing simulators fail to simulate. Moreover, the correctness verification cycle can be accelerated by a factor of 3.2× on average.
- **Hierarchical code generation**: We show that by modularizing a task-parallel program and using a hierarchical approach, RTL code generation can be accelerated by a factor of 6.8× on our server with 32 hyper-threads.

Other related HLS tools [3, 14, 32, 63], streaming works [54, 60], and alternative APIs will be discussed in Section 5.1. Table 1 shows the summary of related work. To the best of our knowledge, TAPA is the only work that provides convenient programming interfaces, universal software simulation, and hierarchical code generation for general task-parallel programs on FPGAs using HLS. TAPA is open-source at `https://github.com/ucla-vast/tapa`.

Table 1: Summary of related work.

Related Work	Programmability			Software Simulation	RTL Code Generation
	Peeking	Transaction	Host Iface.		
Fleet [60]	No	No	N/A	Sequential	N/A
Intel HLS (`ihc::pipe`)	No	No	N/A	Multi-thread	Monolithic
Intel HLS (`ihc::stream`)	No	Yes	N/A	Multi-thread	Monolithic
Intel OpenCL	No	No	OpenCL	Multi-thread	Monolithic
LegUp [3]	No	No	N/A	Multi-thread	Monolithic
Merlin [14]	No	No	C++	Sequential	Monolithic
ST-Accel [54]	No	No	VFS	Sequential	Hierarchical
Vivado HLS (`ap_fifo`)	No	No	OpenCL	Sequential	Monolithic
Vivado HLS (`axis`)	No	Yes	OpenCL	Multi-thread	Manual
Xilinx OpenCL	No	No	OpenCL	Multi-thread	Monolithic
TAPA	Yes	Yes	C++	Coroutine	Hierarchical

2 BACKGROUND

2.1 Task-Level Parallelism

Task-level parallelism is a form of parallelization of computer programs across multiple processors. In contrast to data parallelism where the workload is partitioned on data and each processor executes the same program (e.g., OpenMP [21]), different processors in a task-parallel program often behave differently, while data are passed between processors. Examples of task-parallel programs include image processing pipelines [10, 52], graph processing [61, 67], and network switching [50]. Software programs usually implement tasks as threads and/or processes and rely on the operating system to schedule execution and handle communication. This often leads to poor performance caused by inefficient inter-task communication and frequent context switch [2]. Hardware programs, on the other hand, can be much more efficient due to the massive amount of inherently parallel logic units. In this paper, we focus on the problem of statically mapping tasks to hardware. That is, instances of tasks are synthesized to different areas in an FPGA accelerator. We plan to address dynamic scheduling in our future work.

2.2 Task-Parallel Programming Models

Task-parallel programs are often described as communicating sequential processes [29] or using dataflow models [36, 43, 51]. Kahn process network (KPN) [36] is one of the most popular models used. Under the KPN model, tasks are called *processes*. Processes communicate only through unidirectional *channels*. Data exchanged between channels are called *tokens*. KPN requires that ① each process is deterministic, i.e., the same input sequence must produce the same output sequence; ② channels are unbound, read blocks if and only if the channel is empty, and write always succeed immediately; ③ a process cannot test an input channel for existence of tokens without consuming them. While KPN and models derived from KPN (e.g., synchronous dataflow [43]) have been successful in scheduling tasks on parallel processors, we show in the next section that, when applied to model task-parallel HLS programs, such models lack good programmability support. In this paper, we borrow the terms *process*, *channel*, and *token* used in the KPN formulation, but are not limited to KPN or any dataflow model. In fact, we will describe our programming model as a hierarchical finite state machine in Section 3.1.1.

2.3 Motivating Example

Graph is an important data structure that is critical in many data mining and machine-learning algorithms [26, 38, 45, 47, 49]. While there are many existing FPGA accelerators designed for graph algorithms [22, 23, 37, 48, 65–67], none of them are programmed in HLS. HLS's lack of good productivity for task-parallel programs is one of the reasons it is not adopted for graph algorithms. In this section, we use a real-world design to illustrate the productivity issues for implementing graph accelerators in HLS, which serve as a motivating example for our work.

Our example accelerator implements PageRank [49] on the Alveo U280 board and leverages the high-bandwidth memories (HBM). The input graph is pre-processed and loaded into the HBM on the FPGA. The accelerator adopts an edge-centric graph programming model [53] and decouples the computation into two phases, i.e., the scatter phase and the gather phase [11, 67]. In the scatter phase, edges are streamed from the HBM to the processing elements (PE) on FPGA. For each edge, an update message is generated to propagate the weighted ranking of the source vertex to the destination vertex. The updates are collected and stored off-chip in the HBM. In the gather phase, the updates are loaded from the HBM and the rankings are accumulated over each vertex. Our PageRank accelerator instantiates multiple PEs. The PEs are connected to a vertex handler and a control module. The control module coordinates accesses to the vertex attributes and iterative execution between the two phases. Figure 3 shows the block diagram of the example accelerator.

We measured 4.4 GTEPS[1] on-board execution throughput using the accelerator with 19 HBM channels in use. As a comparison, multi-thread CPU performance is around 0.7 GTEPS with 4 DDR4 memory channels [11]. Even if we assume a similar memory bandwidth as the FPGA accelerator and project the CPU performance to 3.5 GTEPS, it would still be more than 20% slower, due to the lack of fine-grained control over communication. While developing this

[1]Giga traversed edges per second.

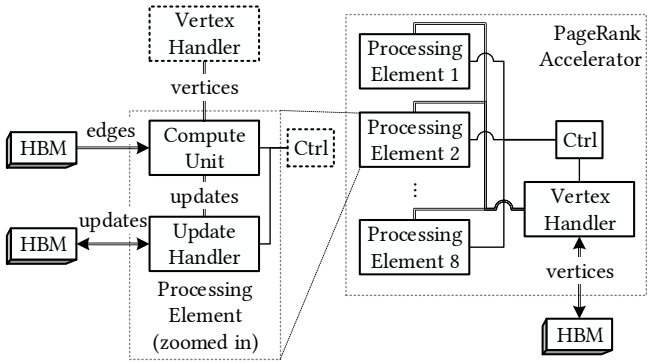

Figure 3: Example PageRank accelerator design.

accelerator, we found that the following are missing or hard-to-use in the HLS tools and significantly impact the productivity.

1) *Peeking.* Peeking is defined as reading a token from a channel without consuming it. As mentioned in Section 2.2, KPN explicitly prohibits such behavior. Yet, such a pattern is common in many applications. For example, in the PageRank accelerator, the `UpdateHandler` module needs to keep track of the number of updates destinated to each vertex partition. Due to the large number of partitions, block RAMs (BRAM) are used for storing the update counts. However, incrementing a value in BRAM cannot be done in a single clock cycle on FPGAs due to the addressing latency, which prevents the loop from being fully pipelined. A workaround is to accumulate the update count in a register for updates with the same partition id (`pid`) and only write changes to BRAM when the `pid` changes. This requires us to detect conflicts on the addresses and stop reading the input channel when conflict occurs, as shown in the green lines marked with "+" in Listing 1. Without a `peek` API, one has to write it as the red lines marked with "−" in Listing 1 to manually maintain a buffer for the incoming values. This not only increases the programming burden, but also makes the design prone to errors in state transitions of the buffer.

2) *Transactions.* A sequence of tokens may constitute a single logical communication transaction. Using the same PageRank accelerator example, in the gather phase when the updates are read from HBM, the updates transmitted from `UpdateHandler` to `ComputeUnit` for each vertex partition can be considered a single transaction. Since only `UpdateHandler` knows the number of updates transmitted in each transaction, `ComputeUnit` needs to test for a special token to detect the *end of transaction* (green lines marked with "+" in Listing 2). Without an `eot` API, one has to manually add a special bit to the data structure representing the tokens (red lines marked with "−" in Listing 2). Note that the `Update` struct is used elsewhere and it is infeasible to add the `eot` bit directly to the `Update` struct. An alternative solution, i.e., sending the length of transaction to the token consumer beforehand, is not only more complicated, but also impractical in cases where the tokens are generated dynamically and the length of transaction cannot be determined beforehand.

3) *System integration.* To offload computation kernel from the host CPU to FPGA accelerators, programmers need to write host-side code to interface the accelerator kernel with the host. FPGA vendors adopt the OpenCL standard to provide such a functionality. While the standard OpenCL host-kernel interface infrastructure

```
1   1  int counts[kPartitionCount] = {};
2   2  int count, last_pid = -1, last_last_pid = -1;
3     -UpdateWithPid buf;
4     -bool buf_valid = false;
5   3  while (...) {
6     -  if (buf_valid) {
7     -    Pid pid = buf.pid;
      4+   Pid pid = input_chan.peek().pid;
8   5    if (pid != last_pid && pid == last_last_pid) {
9   6      last_last_pid = -1;  // BRAM conflict on counts
10  7    } else {
11    -      Update update = buf.update;
12    -      // read for next iteration
13    -      buf_valid = input_chan.try_read(buf);
      8+     Update update = input_chan.read().update;
14  9      if (last_pid != pid) {
15  10       // commit change to BRAM
16  11       if (last_pid != -1) counts[last_pid] = count;
17  12       count = counts[pid] + 1;
18  13     } else {
19  14       ++count;  // accumulate over register
20  15     }
21  16     last_last_pid = last_pid;
22  17     last_pid = pid;
23  18     // write update to HBM
24    -      updates[...] = update;
25    -    }
26    -  } else {
27    -    // non-blocking read
28    -    buf_valid = input_chan.try_read(buf);
      19+    updates[...] = input_chan.read().update;
29  20   }
30  21 }
```

Listing 1: Code snippets with (green lines marked with "+") and without (red lines marked with "–") a peek API. Without the peek API, the code snippet is 33% longer and error-prone.

```
1     -struct UpdateWithEot {
2     -  Update update;
3     -  bool eot;
4     -};
5     -
6   1  // test for end-of-transaction token
7     -while (!input_chan.peek().eot) {
8     -  Update update = input_chan.read().update;
      2+ while (!input_chan.eot()) {
      3+   Update update = input_chan.read();
9   4    ...  // accumulate the updates
10  5  }
```

Listing 2: Code snippets with (green lines marked with "+") and without (red lines marked with "–") an end-of-transaction (eot) API. Without the eot API, the code snippet is 2× longer.

relieves programmers from writing their own operating system drivers and low-level libraries, it is still inconvenient and hard-to-use. Programmers often have to write and debug tens of lines of code just to set up the host-kernel interface. Task-parallel accelerators often make the situation worse because the parallel tasks are often described as distinct OpenCL kernels [31], which significantly increases the programmers' burden on managing these kernels in the host-kernel interface. For our PageRank accelerator, more than 60 lines of host code are created just for the host-kernel integration, which constitute more than 20 percent of the whole source code. Yet, what we actually need is just a single function invocation of the synthesized FPGA bitstream given proper arguments.

4) *Software simulation.* C does not have explicit parallel semantics by itself. Vivado HLS uses the dataflow model and allow programmers to instantiate tasks by invoking each of them sequentially [63]. While this is very concise to write (red lines marked with "–" in Listing 3), it will lead to incorrect simulation results because the communication between `ComputeUnit` and `UpdateHandler` are bidirectional, yet sequential execution can only send tokens from `ComputeUnit` to `UpdateHandler` because of their invocation order. This problem was also pointed out in [8]. In order to run software simulation correctly, the programmer can change the source code to run tasks in multiple threads for software simulation, but doing so requires the same piece of task instantiation code to be written twice for synthesis and simulation, reducing productivity. While other tools that run tasks in parallel threads do not have the same correctness problem, we will show in Section 4.4 that such simulators do not scale well when the number of tasks increase.

```
1   1  void PageRank(...) {
2   2    ...  // declare channels
3     -#pragma HLS dataflow
4     -  Ctrl(...);
5     -  VertexHandler(...);
6     -  ComputeUnit(...);
7     -  ComputeUnit(...);
      3+   tapa::task()
      4+     .invoke(Ctrl, ...)
      5+     .invoke(VertexHandler, ...)
      6+     .invoke(ComputeUnit, ...)
      7+     .invoke(ComputeUnit, ...)
8   8      ...
9     -  UpdateHandler(...);
10    -  UpdateHandler(...);
      9+     .invoke(UpdateHandler, ...)
      10+    .invoke(UpdateHandler, ...)
11  11     ...
      12+    ;
12  13 }
```

Listing 3: Code snippets that instantiate tasks in Vivado HLS (red lines marked with "–") and TAPA (green lines marked with "+"). The instantiation interface in Vivado HLS is not verbose, but software simulation does not work correctly.

5) *RTL code generation.* In our PageRank design, the same processing element is instantiated 8 times. This makes the HLS compiler synthesize the same PE module 8 times, taking 7 minutes per compilation. We can reduce the code generation time to less than 1 minute by manually synthesizing each module separately and connecting the generated RTL code, but doing so forces us to debug RTL code and spend tens of minutes to verify the correctness for each code modification, thus defeats the purpose for adopting HLS.

In this paper, we present the TAPA framework that addresses these challenges by providing convenient programming interfaces, universal software simulation, and hierarchical code generation.

3 TAPA FRAMEWORK

3.1 Programming Model and Interface

3.1.1 Hierarchical Finite-State Machine Model. Similar to KPN described in Section 2.2, tasks in TAPA communicate via channels. Unlike KPN, tasks are modeled as hierarchical finite-state machines (FSM). Each task is either a leaf that does not instantiate any channels or tasks, or a collection of tasks and channels with which the tasks communicate. A task that instantiates a set of tasks and channels is called the *parent task* for that set. Each channel must be connected to exactly two tasks that are instantiated in the same parent task. One of the tasks must act as a *producer* and the other must act as a *consumer*. The producer *streams* tokens to the consumer via the channel in the first-in-first-out (FIFO) order. Each task is an FSM, where the tokens streamed to and from the task are inputs and outputs to the FSM. In case of a parent task, the state of all instantiated channels and tasks constitute its state. The producer of a channel can test the fullness of the channel and append tokens to the channel (*write*) if the channel is not full. The consumer of a channel can test the emptiness of the channel and remove tokens from the channel (*read*), or duplicate the head of token without removing it (*peek*), if the channel is not empty. Read, peek, and write operations can be blocking or non-blocking. A blocking operation on an input (output) channel keeps the task FSM in its current state until the channel becomes non-empty (non-full). A non-blocking operation *tries* to perform the operation and returns whether it is successful as one of the inputs to the task FSM. Each task is implemented as a C++ function, which can communicate with each other via the *communication interface*. A parent task instantiates channels and tasks using the *instantiation interface*. One of the tasks is designated as the *top-level task*, which defines the communication interfaces external to the FPGA accelerator.

3.1.2 Communication Interface. Tasks communicate with each other through the communication interface. TAPA provides separated communication APIs for the producer side and the consumer side. The producer and consumer tasks of a channel use `ostream` and `istream` as the interfaces, respectively. The interfaces are templated and can be used for any copyable class. On the consumer side, `istream` provides peek that allows the programmer to read a token without removing it from the channel, i.e., the state of the channel is not changed. A special token denoting end-of-transaction (EoT) is available to all channels. A process can "close" a channel by writing an EoT to it, and a process can "open" a channel by reading an EoT from it. An EoT token does not contain any useful data. This is designed deliberately to make it possible to break from a pipelined loop when an EoT is present (Listing 2). Table 2 summarizes the communication interfaces provided by TAPA. Listing 4 shows an example of how the communication interfaces are used in TAPA.

3.1.3 Instantiation Interface. A parent task can instantiate channels and tasks using the instantiation interface. Channels are instantiated using `tapa::channel<type, capacity>`. For example, `tapa::channel<VertexReq, 2>` instantiates a channel with capacity 2, meaning up to 2 tokens can be written to this channel without reading them out or blocking the producer. Data tokens transmitted using this channel have type `VertexReq`. Tasks are instantiated using `tapa::task::invoke`. By default, a parent task does not finish until all its children tasks finish. A child task can optionally be invoked with `tapa::detach`, meaning the child task is launched and detached immediately, and the parent does not wait for it to finish. The `tapa::detach` invocation type is particularly useful when a task never terminates, e.g., `VertexHandler` with an infinite loop (Listing 4). Listing 5 shows an example of how channels and tasks are instantiated in TAPA.

Table 2: TAPA communication interface.

`tapa::ostream<T>&` API	Producer-side functionality
`bool full();`	fullness test
`void write(T);`	blocking write a data token
`bool try_write(T);`	non-blocking write a data token
`void close();`	blocking write an EoT token
`bool try_close();`	non-blocking write an EoT token
`tapa::istream<T>&` API	**Consumer-side functionality**
`bool empty();`	emptiness test
`T peek();`	blocking peek a data token
`bool try_peek(T&);`	non-blocking peek a data token
`T read();`	blocking read a data token
`bool try_read(T&);`	non-blocking read a data token
`bool eot();`	return if next token is EoT
`bool try_eot(bool&);`	return if next token exists and if it is EoT
`void open();`	blocking read an EoT token
`bool try_open();`	non-blocking read an EoT token

```
void VertexHandler(tapa::istream<VertexReq>& req_s, ...) {
  for (;;) {
    VertexReq req;
    if (req_s.try_read(req)) {
      ...  // handle requests
    }
  }
}

void Ctrl(tapa::ostream<VertexReq>& vertex_req, ...) {
  ...                       // initial setup
  while (...) {             // iterative execution
    VertexReq req(...);     // request vertices
    vertex_req.write(req);
    ...                     // finish scatter & do gather
  }
}
```

Listing 4: TAPA communication interface example.

```
void PageRank(...) {
  tapa::channel<VertexReq, 2> vertex_req;
  ...
  tapa::task()
    .invoke<tapa::detach>(VertexHandler, vertex_req, ...)
    .invoke(Ctrl, vertex_req, ...)
    ...
    ;
}
```

Listing 5: TAPA instantiation interface example.

3.1.4 System Integration Interface. To offload a kernel to an FPGA accelerator, programmers will need to integrate the FPGA into the host CPU system. Thanks to the vendor-provided system drivers and the standard OpenCL accelerator APIs, most programmers only need to follow the OpenCL host-kernel communication specification and invoke proper APIs. However, those OpenCL APIs are still verbose and take a long time to learn and develop. For example, programmers need to learn the concepts of "platform", "context", "queue", and "kernel" in OpenCL and manage them for each accelerator, yet the only thing necessary is usually just find a proper FPGA accelerator or simulation environment and use it to run the program. This overhead for programmers is exacerbated by task-parallel accelerators, where parallel tasks are often synthesized as concurrent OpenCL kernels that need to be managed separately by the host.

TAPA uses a unified system integration interface to further reduce programmers' burden. To offload a kernel to an FPGA accelerator, programmers only need to call the top-level task as a C++ function in the host code. Since TAPA can extract metadata information, e.g., argument type, from the kernel code, TAPA will automatically synthesize proper OpenCL host API calls and emit an implementation of the top-level task C++ function that can set up the runtime environment properly. As a user of TAPA, the programmer can use a single function invocation in the same source code to run software simulation, hardware simulation, and on-board execution, with the only difference of specifying proper bitstreams.

3.2 Software Simulation

State-of-the-Art Approach. There are mainly two state-of-the-art approaches that run fast software simulation for task-parallel applications: the sequential approach and the multi-thread approach. A sequential simulator invokes tasks sequentially in the invocation order [63]. Sequential simulators are fast, but cannot correctly simulate the capacity of channels and applications with tasks communicating bidirectionally, as discussed in Section 2.3. A multi-thread simulator invokes tasks in parallel by launching a thread for each task. This enables the capacity of channels and bidirectional communication to be simulated correctly. However, they may perform poorly due to the inefficiency of inter-thread communication and context switch handled by the operating system. The FLASH simulator [8, 12] proposed an alternative to the above, which relies on the HLS scheduling information to mimic the RTL FSM. While this simulation approach itself is faster than multi-thread simulators, generating simulation executable becomes slower due to the need of the HLS scheduler output for cycle-accuracy, which is not needed for correctness verification.

In this section, we present an alternative approach to run software simulation on task-parallel applications. Given that the inefficiency of multi-thread execution is mainly caused by the preemptive nature of operating system threads and inspired by the widespread adoption of coroutines in modern software languages [25, 41], we propose an approach that uses collaborative coroutines instead of preemptive threads. Note that fast and/or cycle-accurate debugging in general [33] is out of the scope of this paper; we focus on the correctness and scalability issues for task-parallel programs.

Coroutine-Based Approach. Routines in programming languages are the units of execution contexts, e.g., functions in C [39]. Coroutines [20] are routines that execute collaboratively; more specifically, coroutines can be explicitly suspended and resumed. Coroutines can even maintain their own stacks. As a result, each coroutine can invoke subroutines themselves and suspend from and resume to any subroutine [41]. Coroutines that have their own stacks are called *stackful* coroutines. A context switch between coroutines takes only 26ns on modern CPUs [41]. As a comparison, an operating system thread context switch takes 1.2 ˜2.2µs [2], which is two orders of magnitude slower.

TAPA leverages stackful coroutines to perform software simulation. When channels are instantiated in the simulator, enough memory space is reserved to ensure the channel capacity can be simulated correctly. When tasks are instantiated, a coroutine is launched but suspended immediately for each task. Once all tasks are instantiated, the simulator starts to resume the suspended coroutines. A resumed task will be suspended again if any input channel is accessed when empty or any output channel is accessed when full, which means that no progress can be made from this task. A different task will then be selected and resumed by the simulator.

For example, in the task instantiation code shown in Listing 5, both `VertexHandler` and `Ctrl` are launched as coroutines and suspended immediately by the `invoke` function calls. Once all tasks are instantiated, the simulator starts to pick tasks for execution. `Ctrl` is picked first, which will write vertex requests to `vertex_req`. Once `vertex_req` becomes full, the simulator determines that no progress can be made from `Ctrl`, thus will suspend it and pick another task for execution. `VertexHandler` is then resumed and tokens will be read from `vertex_req`. Once `vertex_req` becomes empty, the simulator determines that no progress can be made from `VertexHandler`, thus will suspend it and pick the next task for execution.

To better utilize the available CPU cores, we use a thread pool to execute the coroutines. We will show in Section 4.4 that the coroutine-based simulator outperforms the existing simulators by 3.2× on average (Section 4.4).

3.3 RTL Code Generation

State-of-the-art Approach. Current HLS tools treat the whole task-parallel program as a monolithic design, treat channels as global variables, and compile different instances of tasks as if they are completely unrelated. While this enables instance-specific optimizations, e.g., different constant arguments can be propagated to different instances, it can also lead to a significant amount of repeated work. For example, the dataflow architecture generated by the SODA compiler [9, 10] is highly modularized and many modules are functionally identical. However, both the Vivado HLS backend and Intel FPGA OpenCL backend of SODA generate RTL code for each SODA module separately. When the design scales out to hundreds of modules, RTL code generation can easily run for hours, taking even longer time than logic synthesis and implementation. While we recognize that a programmer can manually generate RTL code for each task and glue them at RTL level to speed up RTL code generation, doing so defeats the purpose of using HLS for high productivity, because the glued RTL code can be

error-prone yet cannot be verified using fast software simulation. We also recognize that fast RTL code generation in general is an interesting problem, but we focus on the inefficiency exacerbated by task-parallel programs in this paper.

Modularized Approach. Thanks to the hierarchical programming model, TAPA can keep the program hierarchy, recognize different instances of the same task, and compile each task only once. As such, the total amount of time spent on RTL code generation is reduced. Moreover, modularized compilation makes it possible to compile tasks in parallel, further reducing RTL code generation time on multi-core machines. TAPA implements this by doing a source-to-source transformation to generate the vendor HLS code for each task and invoking the vendor tools in parallel for each task. On average, TAPA reduces HLS compilation time by 4.9× (Section 4.5).

3.4 TAPA Automation Overview

The TAPA automation flow is shown in Figure 4. The TAPA C++ source code can be compiled directly for software simulation and correctness verification. Starting from the same TAPA C++ source code, TAPA extracts the HLS code for each task and the metadata information of the whole design, including the communication topology among tasks, token types exchanged between tasks, and channels' capacity. The vendor HLS tool is then leveraged to generate RTL code and performance/resource report for each task. The extracted metadata is used to instantiate the task instances and connect them together systematically, producing the overall HLS report and kernel RTL code, which can be used for QoR tuning and logic synthesis and implementation, respectively. The same metadata information is also used to create the host-kernel communication interface, which can be used for on-board execution or optionally RTL simulation.

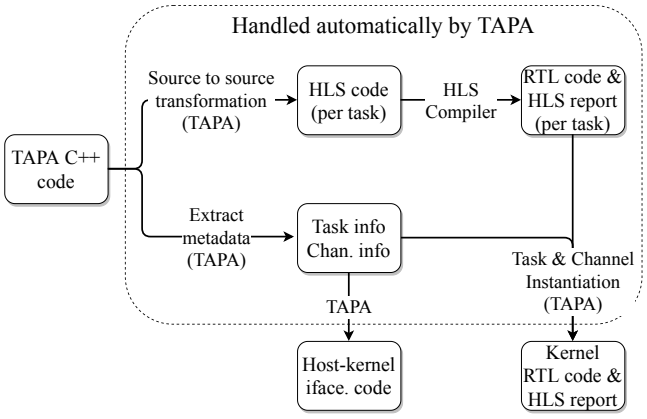

Figure 4: TAPA automation flow overview.

4 EVALUATION

We prototype TAPA on Xilinx devices using Vivado HLS as the backend; support for Intel devices will be added later. Clang compiler infrastructure is modified to extract information about tasks and perform source-to-source transformation to generate Vivado HLS kernel code and OpenCL host code. GCC is used to compile the host executables and the software simulators. We compare the productivity of TAPA with two vendor tools that provide end-to-end high-level programming experience (including host-kernel communication): Xilinx Vitis/Vivado HLS 2019.2 suite and Intel FPGA SDK for OpenCL Pro Edition 19.4. The experimental results are obtained on an Ubuntu 18.04 server with 2 Xeon Gold 6244 processors.

4.1 Benchmarks

We used the following benchmarks for comparison. All implementations (Vivado HLS, Intel OpenCL, and TAPA) of each benchmark are written in such a way that tasks in each implementation have one-to-one correspondence, corresponding loops are scheduled with the same initiation interval (II), and each task performs the same computation. This guarantees all tools generate consistent quality of results. Note that we aim to compare the productivity of each of the HLS tools, not the quality of result. In particular, we were unable to guarantee that the generated RTL codes have exactly the same behavior without having access to the HLS compiler's scheduling algorithm. For example, the network switch implemented in TAPA has a total latency of 3 cycles while the Vivado HLS implementation has a total latency of 6. This is inevitable because, using Vivado HLS, one has to manually buffer the incoming packets, forcing an additional latency of 1 cycle at each network stage. Table 3 summarizes the number of tasks and channels used in each benchmark.

Cannon's Algorithm. Cannon's algorithm [44] is a distributed algorithm for matrix multiplication that runs on 2D mesh of processing elements (PE). This benchmark contains 8×8 PEs. Each PE is internally vectorized to perform 8 multiply-accumulate operations per cycle for two 128×128 matrices. Besides the 64 PEs, the accelerator also contains 9 data distributor/collector for each matrix. The inputs to the whole accelerator are 1024×1024×1024.

Convolutional Neural Network. Convolutional neural networks are very popular for many machine learning applications, e.g., image classification [58]. This benchmark implements the third layer of VGG [58] [2] based on a systolic array implementation generated from PolySA [16]. PolySA is a polyhedral-based systolic array auto-compilation framework that can generate optimal designs within one hour with performance comparable to state-of-the-art manual designs.

Gaussian Filter. The Gaussian filter is often employed for low-pass filtering on input signals or images, or used iteratively for solving linear system of equations. This benchmark is based on a dataflow microarchitecture generated from SODA [10]. SODA is a stencil compiler that can generate optimal communication-reuse buffers with temporal and spatial parallelism. This benchmark performs 8 iterations of Gaussian filtering, each of which is capable of processing 16 input elements in parallel. The input size is 32768×32768.

Graph Convolutional Network. Graph convolutional network [38] is an emerging type of neural network that processes sparse and irregular data as opposed to dense and regular ones like images. This benchmark implements a forward layer of GCN for the Cora

[2] Parameters $\{i, o, h, w, p, q\} = \{512, 512, 56, 56, 3, 3\}$.

Table 3: Benchmarks used in this paper. Each task may be instantiated multiple times, so the number of task instances is greater than the number of tasks.

Benchmark	#Tasks	#Task Instances	#Channels
cannon	5	91	344
cnn	14	209	366
gaussian	15	564	1602
gcn	5	12	25
gemm	14	207	364
network	3	14	32
page_rank	4	18	89

dataset [57], which contains 2708 vertices and 10556 edges. The input and output features have 1433 and 16 dimensions, respectively.

General Matrix Multiplication. This benchmark is based on a systolic array implementation generated from PolySA [16]. Compared with Cannon's algorithm, PolySA avoids feedback data paths in the systolic array, and can support non-square matrices. The inputs to the accelerator are 1040×1024×1024.

Network Switching. This benchmark implements an 8×8 Omega network switch [42] that can route packets from any input port to any output port. The packets are 64-bit wide with the first 3 bits being the header and are generated randomly with an even distribution among the 8 destination ports.

PageRank. This benchmark implements the PageRank [49] citation ranking algorithm for general large graphs as described in Section 2.3. We use the Slashdot community graph [45] as the dataset for debugging, which contains 77360 vertices and 905468 edges. The accelerator design itself can scale up to 2^{26} vertices and 2^{28} edges.

4.2 Lines of Kernel Code

TAPA simplifies the kernel code in two aspects. First, the TAPA communication interfaces simplify the code with the built-in support for peeking and transactions. This not only simplifies the body of each task definition, but also removes the necessity for many `struct` definitions. Second, the TAPA instantiation interfaces simplify the code by allowing tasks to be launched and detached concisely. Without this functionality, each task in Vivado HLS must be carefully given a termination condition, whereas Intel OpenCL requires verbose kernel instantiation attributes for each instance of task. Figure 5 shows the lines of kernel code comparison of each benchmark. On average, TAPA reduces the lines of kernel code by 22%. Note that only synthesizable kernel code is counted; code added for multi-thread software simulation is not counted for Vivado HLS.

4.3 Lines of Host Code

The host code used in the benchmarks contains a minimal testbench to verify the correctness of the kernel code. TAPA system-integration API automatically interfaces with the OpenCL host APIs and relieves the programmer from writing repetitive code just to connect the kernel to a host program. Table 6 shows the lines

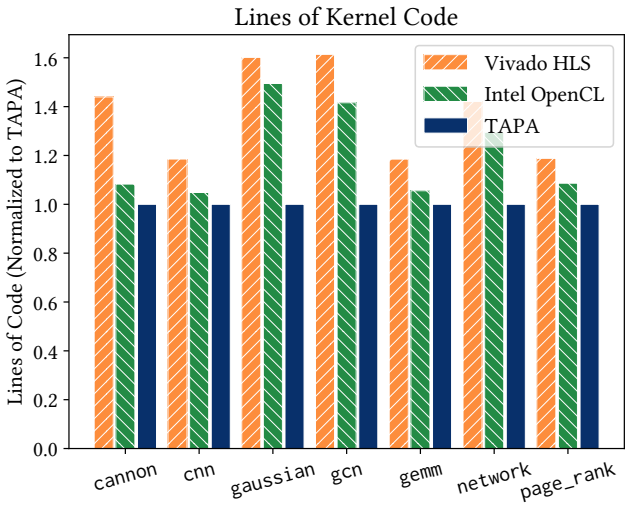

Figure 5: LoC comparison for kernel code. Lower is better.

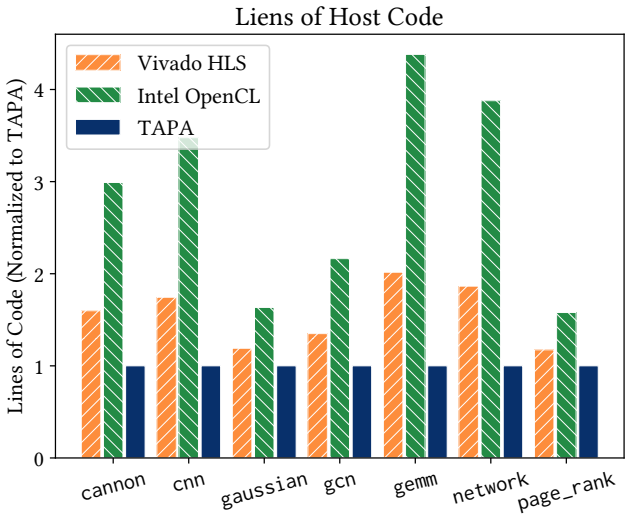

Figure 6: LoC comparison for host code. Lower is better.

of host code comparison. On average, the length of host code is reduced by 51%.

4.4 Software Simulation Time

Figure 7 shows four simulators, that is, the sequential Vivado HLS simulator, the multi-thread Vivado HLS simulator, the multi-thread Intel OpenCL simulator, and the coroutine-based TAPA simulator. Among the three simulators, the sequential simulator fails to correctly simulate benchmarks that require feedback data paths (`cannon` and `page_rank`). Due to the larger memory footprint required for storing the tokens transmitted between tasks and lack of parallelism, the sequential simulator is outperformed by the coroutine-based simulator in all but one of the benchmarks (`network`). The two multi-thread simulators correctly simulate all benchmarks, except that Intel OpenCL cannot handle `gaussian` because its large number of task instances (564) exceeds the maximum allowed (256) by the simulator. However, the multi-thread

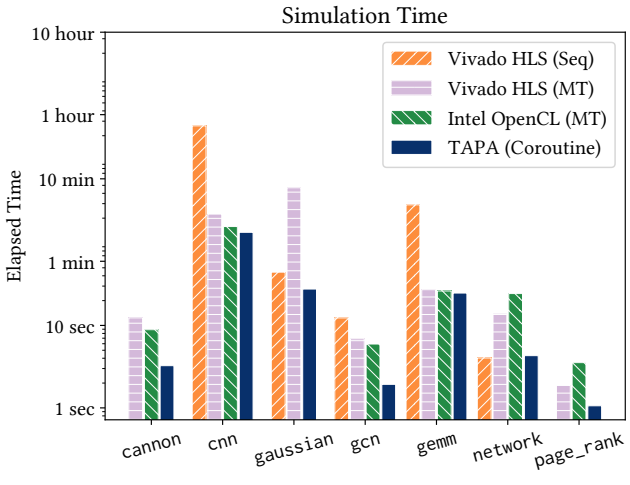

Figure 7: Simulation time comparison. Lower is better. The sequential simulator fails to simulate `cannon` and `pagerank` correctly. The Intel OpenCL multi-thread simulator cannot simulate `gaussian` due to its large number of task instances.

simulators perform poorly on benchmarks that are communication-intensive (e.g., `network`) or have more tasks than the number of available threads (e.g., `gaussian`). The coroutine-based TAPA simulator can correctly simulate all benchmarks without significant performance loss for both communication-intensive and computation-intensive tasks with 3.2× average speedup.

4.5 RTL Code Generation Time

Figure 8 shows the RTL code generation time comparison. Thanks to the hierarchical programming model and modularized code generator, TAPA accelerates the HLS compilation time by 6.8× on average. This is because ① TAPA runs HLS for each task only once even if it is instantiated many times, while Vivado HLS and Intel OpenCL runs HLS for each task instance; ② TAPA runs HLS in parallel on multi-core machines.

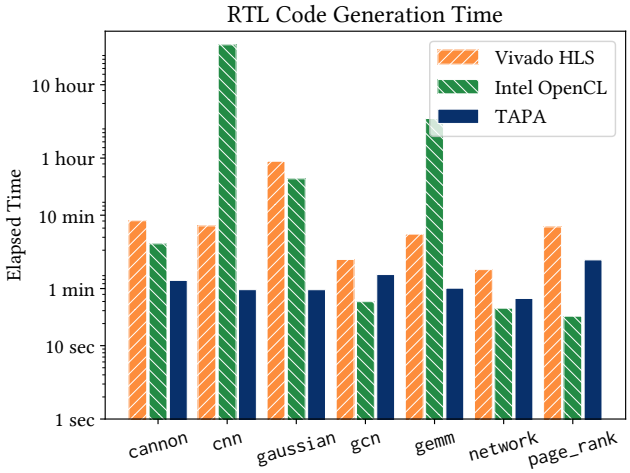

Figure 8: RTL code generation time. Lower is better.

5 RELATED WORK

5.1 HLS Support for Task-Parallel Programs

Intel HLS compiler supports two different inter-task communication interfaces, `ihc::pipe` and `ihc::stream`. `ihc::pipe` implements a light-weight hardware FIFO with `data`, `valid`, and `ready` signals, while `ihc::stream` implements an Avalon-ST interface that supports transactions. Tasks are instantiated using `ihc::launch` and `ihc::collect`. Software simulation is done via launching multiple threads. Instances of the same task are synthesized separately.

Intel OpenCL compiler supports light-weight FIFO via two sets of APIs, i.e., standard OpenCL `pipe` and Intel-specific `channel`. Tasks are instantiated by defining OpenCL `__kernel`s, which forces instances of the same task to be synthesized separately as different OpenCL kernels. OpenCL runtime handles the software simulation by launching multiple threads.

Vivado HLS provides two different streaming interfaces: `ap_fifo` and `axis`. The `ap_fifo` interface generates light-weight FIFO interface. Tasks are instantiated by invoking the corresponding functions in a `dataflow` region, and instances of the same task are synthesized separately. Software simulation is done by sequentially executing the tasks. The `axis` interface generates AXI-Stream interface with transaction support. It requires the programmers to instantiate channels and tasks in a separate configuration file when running logic synthesis and implementation. This allows different instances of the same task to be synthesized only once, but takes longer time to learn and implement compared with `ap_fifo`. OpenCL runtime handles the software simulation for tasks instantiated with the `axis` interface by launching multiple threads.

Xilinx OpenCL compiler supports standard OpenCL `pipe`, which generates AXI-Stream interfaces similar to Vivado HLS `axis`, but `pipe` does not provide APIs to support transactions. Like Vivado HLS `axis`, software simulation of `pipe` is handled by the OpenCL runtime by launching multiple threads.

LegUp compiler provides `legup::FIFO`, which implements light-weight FIFOs. Tasks are instantiated using `pthread` API (Section 5.3). Software simulation is accomplished by launching multiple threads. Instances of the same task are synthesized separately.

Merlin compiler [14] allows programmers to call the FPGA kernel as a C/C++ function and provides OpenMP-like simple pragmas with automated design-space exploration based on machine learning. To support task-parallel programs, Merlin leverages its backend vendor tools' programming interfaces. Software simulation is done by sequentially executing the tasks.

In summary, as pointed out in Table 1 (on page 2), none of the state-of-the-art HLS tools provide peeking support. Only Intel HLS `ihc::stream` and Vivado HLS `axis` support transactions. Only Merlin allows the accelerator kernel to be called as if it is a C/C++ function. Vivado HLS and Merlin execute tasks sequentially for simulation while others launch multiple threads. All HLS tools treat a task-parallel program as a monolithic design and generate RTL code for each instance of task separately, except that Vivado HLS `axis` allows programmers to manually instantiate tasks using a configuration file when running logic synthesis and implementation.

5.2 Streaming Framework

Streaming applications are a special type of task-parallel applications that do not require complex control over inter-task communication and often expose massive data parallelism in addition to task parallelism. There are previous works that focus specifically on such applications.

ST-Accel [54] is a high-level programming platform for streaming applications that features highly efficient host-kernel communication interface exposed as a virtual file system (VFS). It uses Vivado HLS as its backend for hardware generation and its software simulation is done by sequential execution.

Fleet [60] is a massively parallel streaming framework for FPGAs that features highly efficient memory interfaces for massive instances of parallel processing elements. Programmers write Fleet programs in a domain-specific RTL language based on Chisel [1]. The programs can be simulated in Scala[3].

In summary, while these frameworks are specialized for streaming patterns, neither of them provide peeking and transaction interface in the kernel. Both run software simulation sequentially, which does not have correctness problem for streaming applications but will be restrictive for general task-parallel programs.

5.3 Alternative APIs

SystemC is a set of C++ classes and macros that provide detailed hardware modeling and event-driven simulation. It supports both cycle-accurate and untimed simulation and many simulator implementations are available [13, 56]. Some HLS tools support a subset of untimed SystemC as the input [63]. SystemC supports task-parallel programs natively via the `sc_module` constructs and `tlm_fifo` interfaces. Listing 6 shows an example using the accelerator discussed in Section 2.3. Compared with other C-like HLS languages, SystemC can model more hardware details but is more verbose and less productive due to its special language constructs: for the code snippets shown in Listing 4 and Listing 5, equivalent SystemC code would be 37% longer.

Pthread API is a set of widely used standard APIs that can be used to implement task-parallel programs using threads. Pthread requires programmers to explicitly create and join threads, and arguments need to be manually packed and passed. Listing 7 shows an example using the accelerator discussed in Section 2.3. Compared with the `tapa::invoke` API used by TAPA, the pthread APIs require more effort to program: for the code snippets shown in Listing 4 and Listing 5, equivalent pthread-based code would be 78% longer.

In summary, while the existing API alternatives are widely used in some domains, they are more verbose and thus less productive compared with TAPA.

```
SC_MODULE(Ctrl) {
  sc_core::sc_port<tlm::tlm_fifo_put_if<VertexReq>>
    vertex_req;         // declare communication interface
  ...
  SC_CTOR(Ctrl) { SC_THREAD(thread); }
  void thread() { ... }                // task description
};

SC_MODULE(PageRank) {
  // instantiate channels
  tlm::tlm_fifo<VertexReq> vertex_req{/*depth=*/2};
  ...
  Ctrl ctrl;                          // instantiate tasks
  ...
  SC_CTOR(PageRank) {
    // bind channels to communication interfaces
    ctrl.vertex_req(vertex_req);
    ...
  }
};
```

Listing 6: SystemC TLM API example.

```
struct Ctrl_Arg {          // task communication interface
  channel<VertexReq>* vertex_req;
  ...
};

void Ctrl(void* arg) {                 // task description
  Ctrl_Arg* ctrl_arg = (Ctrl_Arg*)arg; // unpack arguments
  channel<VertexReq>* vertex_req = ctrl_arg->vertex_req;
  ...
  pthread_exit(NULL);
}

void PageRank(...)
  channel<VertexReq> vertex_req;   // instantiate channels
  ...
  Ctrl_Arg Ctrl_arg;
  Ctrl_arg.vertex_req = &vertex_req;     // pack arguments
  ...
  pthread_t Ctrl_pid, ...;               // launch threads
  pthread_create(&Ctrl_pid, NULL, Ctrl, (void*)&Ctrl_arg);
  ...
  pthread_join(&Ctrl_pid, NULL);           // join threads
  ...
}
```

Listing 7: Pthread API example.

6 CONCLUSION AND FUTURE WORK

In this paper, we present TAPA as an HLS C++ language extension to enhance the programming productivity of task-parallel programs on FPGAs. TAPA has multiple advantages over state-of-the-art HLS tools: 1) its enhanced programming interface helps to reduce the lines of kernel code by 22% on average, 2) its unified system integration interface reduces the lines of host code by 51% on average, 3) its coroutine-based software simulator reduces the length of correctness verification development cycle by 3.2× on average, 4) its modularized code generation approach accelerates the QoR tuning development cycle by 6.8× on average. As a fully automated and open-source framework, TAPA aims to provide highly productive development experience for task-parallel programs using HLS. For future work, we plan to extend our work to support dynamically generating and executing tasks on FPGAs.

[3]Scala is the language in which Chisel is embedded.

ACKNOWLEDGMENT

This work is partially supported by a Google Faculty Award, the NSF RTML program, Xilinx Adaptive Compute Cluster (XACC) Program, and the CDSC industrial partners.

REFERENCES

[1] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In *DAC*.

[2] Eli Bendersky. 2018. Measuring context switching and memory overheads for Linux threads. (2018). https://eli.thegreenplace.net/2018/measuring-context-switching-and-memory-overheads-for-linux-threads/

[3] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems. In *FPGA*.

[4] Han Chen, Sergey Madaminov, Michael Ferdman, and Peter Milder. 2020. FPGA-Accelerated Samplesort for Large Data Sets. In *FPGA*.

[5] Yu Ting Chen, Jin Hee Kim, Kexin Li, Graham Hoyes, and Jason H. Anderson. 2019. High-Level Synthesis Techniques to Generate Deeply Pipelined Circuits for FPGAs with Registered Routing. In *FPT*.

[6] Jianyi Cheng, Shane T. Fleming, Yu Ting Chen, Jason H. Anderson, and George A. Constantinides. 2019. EASY: Efficient Arbiter SYnthesis from Multi-threaded Code. In *FPGA*.

[7] Jianyi Cheng, Lana Josipović, George A. Constantinides, Paolo Ienne, and John Wickerson. 2020. Combining Dynamic & Static Scheduling in High-level Synthesis. In *FPGA*.

[8] Yuze Chi, Young-kyu Choi, Jason Cong, and Jie Wang. 2019. Rapid Cycle-Accurate Simulator for High-Level Synthesis. In *FPGA*.

[9] Yuze Chi and Jason Cong. 2020. Exploiting Computation Reuse for Stencil Accelerators. In *DAC*.

[10] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. 2018. SODA : Stencil with Optimized Dataflow Architecture. In *ICCAD*.

[11] Yuze Chi, Guohao Dai, Yu Wang, Guangyu Sun, Guoliang Li, and Huazhong Yang. 2016. NXgraph: An Efficient Graph Processing System on a Single Machine. In *ICDE*.

[12] Young-kyu Choi, Yuze Chi, Jie Wang, and Jason Cong. 2020. FLASH: Fast, ParalleL, and Accurate Simulator for HLS. *TCAD* (2020).

[13] Moo Kyoung Chung, Jun Kyoung Kim, and Soojung Ryu. 2014. SimParallel: A High Performance Parallel SystemC Simulator Using Hierarchical Multi-threading. (2014).

[14] Jason Cong, Muhuan Huang, Peichen Pan, Di Wu, and Peng Zhang. 2016. Software Infrastructure for Enabling FPGA-Based Accelerations in Data Centers. In *ISLPED*.

[15] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. 2011. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *TCAD* (2011).

[16] Jason Cong and Jie Wang. 2018. PolySA: Polyhedral-Based Systolic Array Auto-Compilation. In *ICCAD*.

[17] Jason Cong, Peng Wei, Cody Hao Yu, and Peng Zhang. 2018. Automated Accelerator Generation and Optimization with Composable, Parallel and Pipeline Architecture. In *DAC*.

[18] Jason Cong, Peng Wei, Cody Hao Yu, and Peipei Zhou. 2018. Latte: Locality Aware Transformation for High-Level Synthesis. In *FCCM*.

[19] Jason Cong and Zhiru Zhang. 2006. An Efficient and Versatile Scheduling Algorithm Based On SDC Formulation. In *DAC*.

[20] Melvin E. Conway. 1963. Design of a Separable Transition-Diagram Compiler. *Commun. ACM* 6, 7 (1963), 396–408.

[21] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An Industry Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering* 5, 1 (1998), 46–55.

[22] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. 2016. FPGP: Graph Processing Framework on FPGA A Case Study of Breadth-First Search. In *FPGA*.

[23] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. 2017. ForeGraph: Exploring Large-scale Graph Processing on Multi-FPGA Architecture. In *FPGA*.

[24] Johannes De Fine Licht, Grzegorz Kwasniewski, and Torsten Hoefler. 2020. Flexible Communication Avoiding Matrix Multiplication on FPGA with High-Level Synthesis. In *FPGA*.

[25] Ana Lúcia de Moura and Roberto Ierusalimschy. 2009. Revisiting Coroutines. *TOPLAS* 31, 2 (2009).

[26] Chenhui Deng, Zhiqiang Zhao, Yongyu Wang, Zhiru Zhang, and Zhuo Feng. 2020. GraphZoom: A Multi-level Spectral Approach for Accurate and Scalable Graph Embedding. In *ICLR*.

[27] Licheng Guo, Jason Lau, Yuze Chi, Jie Wang, Cody Hao Yu, Zhe Chen, Zhiru Zhang, and Jason Cong. 2020. Analysis and Optimization of the Implicit Broadcasts in FPGA HLS to Improve Maximum Frequency. In *DAC*.

[28] Ameer Haj-Ali, Qijing Huang, William Moses, John Xiang, Krste Asanovic, John Wawrzynek, and Ion Stoica. 2020. AutoPhase: Juggling HLS Phase Orderings in Random Forests with Deep Reinforcement Learning. In *MLSys*.

[29] C. A. R. Hoare. 1978. Communicating Sequential Processes. *Commun. ACM* 21, 8 (1978).

[30] Hsuan Hsiao and Jason Anderson. 2019. Thread Weaving: Static Resource Scheduling for Multithreaded High-Level Synthesis. In *DAC*.

[31] Intel. 2020. Intel FPGA SDK for OpenCL Pro Edition: Programming Guide. (2020).

[32] Intel. 2020. Intel High Level Synthesis Compiler Pro Edition: User Guide. (2020).

[33] Al Shahna Jamal, Eli Cahill, Jeffrey Goeders, and Steven J. E. Wilton. 2020. Fast Turnaround HLS Debugging using Dependency Analysis and Debug Overlays. *TRETS* 13, 1 (2020).

[34] Jiantong Jiang, Zeke Wang, Xue Liu, Juan Gómez-Luna, Nan Guan, Qingxu Deng, Wei Zhang, and Onur Mutlu. 2020. Boyi: A Systematic Framework for Automatically Deciding the Right Execution Model of OpenCL Applications on FPGAs. In *FPGA*.

[35] Lana Josipović, Shabnam Sheikhha, Andrea Guerrieri, Paolo Ienne, and Jordi Cortadella. 2020. Buffer Placement and Sizing for High-Performance Dataflow Circuits. In *FPGA*.

[36] Gilles Kahn. 1974. The Semantics of a Simple Language for Parallel Programming. In *IFIP*.

[37] Soroosh Khoram, Jialiang Zhang, Maxwell Strange, and Jing Li. 2018. Accelerating Graph Analytics by Co-Optimizing Storage and Access on an FPGA-HMC Platform. In *FPGA*.

[38] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *ICLR*.

[39] Donald Ervin Knuth. 1997. *Fundamental Algorithms. The Art of Computer Programming 1* (3rd ed.).

[40] Mostafa Koraei, Omid Fatemi, and Magnus Jahre. 2019. DCMI: A Scalable Strategy for Accelerating Iterative Stencil Loops on FPGAs. *TACO* 16, 4 (2019).

[41] Oliver Kowalke. 2014. Boost Library Documentation, Coroutine2. (2014). https://boost.org/doc/libs/1_65_0/libs/coroutine2/doc/html/coroutine2/intro.html

[42] Duncan H. Lawrie. 1975. Access and Alignment of Data in an Array Processor. *ToC* C-24, 12 (1975).

[43] Edward A. Lee and David G. Messerschmitt. 1987. Synchronous Data Flow. *IEEE* 75, 9 (1987).

[44] Hyuk-Jae Lee, James P. Robertson, and José A.B. Fortes. 1997. Generalized Cannon's Algorithm for Parallel Matrix Multiplication. In *ICS*.

[45] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. 2009. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *Internet Mathematics* 6, 1 (2009), 29–123.

[46] Jiajie Li, Yuze Chi, and Jason Cong. 2020. HeteroHalide: From Image Processing DSL to Efficient FPGA Acceleration. In *FPGA*.

[47] Julian Mcauley. 2012. Learning to Discover Social Circles in Ego Networks. In *NIPS*.

[48] Eriko Nurvitadhi, Gabriel Weisz, Yu Wang, Skand Hurkat, Marie Nguyen, James C. Hoe, José F Martínez, and Carlos Guestrin. 2014. GraphGen: An FPGA Framework for Vertex-Centric Graph Computation. In *FCCM*.

[49] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1998. *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report.

[50] Philippos Papaphilippou, Jiuxi Meng, and Wayne Luk. 2020. High-Performance FPGA Network Switch Architecture. In *FPGA*.

[51] James L Peterson. 1977. Petri Nets. *Comput. Surveys* 9, 3 (1977).

[52] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. 2017. Programming Heterogeneous Systems from an Image Processing DSL. *TACO* 14, 3 (2017).

[53] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: Edge-centric Graph Processing using Streaming Partitions. In *SOSP*.

[54] Zhenyuan Ruan, Tong He, Bojie Li, Peipei Zhou, and Jason Cong. 2018. ST-Accel: A High-Level Programming Platform for Streaming Applications on FPGA. In *FCCM*.

[55] Vladimir Rybalkin and Norbert Wehn. 2020. When Massive GPU Parallelism Ain't Enough: A Novel Hardware Architecture of 2D-LSTM Neural Network. In *FPGA*.

[56] Tim Schmidt, Guantao Liu, and Rainer Dömer. 2017. Exploiting Thread and Data Level Parallelism for Ultimate Parallel SystemC Simulation. In *DAC*.

[57] Prithviraj Sen, Galileo Mark Namata, Mustafa Bilgic, Lise Getoor, Brian Gallagher, and Tina Eliassi-Rad. 2008. Collective Classification in Network Data. *AI Magazine* 29, 3 (2008), 93–106.

[58] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *ICLR*.

[59] Atefeh Sohrabizadeh, Jie Wang, and Jason Cong. 2020. End-to-End Optimization of Deep Learning Applications. In *FPGA*.

[60] James Thomas, Pat Hanrahan, and Matei Zaharia. 2020. Fleet: A Framework for Massively Parallel Streaming on FPGAs. In *ASPLOS*.
[61] Yu Wang, James C. Hoe, and Eriko Nurvitadhi. 2019. Processor Assisted Worklist Scheduling for FPGA Accelerated Graph Processing on a Shared-Memory Platform. In *FCCM*.
[62] Xuechao Wei, Yun Liang, and Jason Cong. 2019. Overcoming Data Transfer Bottlenecks in FPGA-based DNN Accelerators via Layer Conscious Memory Management. In *DAC*.
[63] Xilinx. 2020. Vivado Design Suite User Guide: High-Level Synthesis (UG902). (2020).
[64] Tanner Young-Schultz, Lothar Lilge, Stephen Brown, and Vaughn Betz. 2020. Using OpenCL to Enable Software-like Development of an FPGA-Accelerated Biophotonic Cancer Treatment Simulator. In *FPGA*.
[65] Hanqing Zeng and Viktor Prasanna. 2020. GraphACT: Accelerating GCN training on CPU-FPGA heterogeneous platforms. In *FPGA*.
[66] Jialiang Zhang and Jing Li. 2018. Degree-aware Hybrid Graph Traversal on FPGA-HMC Platform. In *FPGA*.
[67] Shijie Zhou, Rajgopal Kannan, Viktor K Prasanna, Guna Seetharaman, and Qing Wu. 2019. HitGraph: High-throughput Graph Processing Framework on FPGA. *TPDS* (2019).
[68] Hamid Reza Zohouri, Artur Podobas, and Satoshi Matsuoka. 2018. Combined Spatial and Temporal Blocking for High-Performance Stencil Computation on FPGAs Using OpenCL. In *FPGA*.