Extending High-Level Synthesis for Task-Parallel Programs

Yuze Chi*, Licheng Guo*, Jason Lau*, Young-kyu Choi*†, Jie Wang*, Jason Cong*
*University of California, *Los Angeles*, †Inha University
{chiyuze,cong}@cs.ucla.edu

***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 results (QoR) and short development cycles compared with the traditional register-transfer level 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 do support task-parallel programs, the productivity is greatly limited ① in the code development cycle due to the poor programmability, ② in the correctness verification cycle due to restricted software simulation, and ③ in the QoR tuning cycle due to slow code generation. 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, unconstrained software simulation, and fast hierarchical code generation to overcome these limitations and demonstrate how task-parallel programs can be productively supported in HLS. 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 shortened by 3.2× and 6.8×, respectively. Our work is open-source at `https://github.com/UCLA-VAST/tapa/`.

I. Introduction

C/C++/OpenCL-based high-level synthesis (HLS) [1] has been adopted rapidly by both the academia and the industry for programming field-programmable gate array (FPGA) accelerator design in many application domains, e.g., machine learning [2–4], scientific computing [5–8], and image processing [9–11]. Compared with the traditional register-transfer level (RTL) paradigm where the debug turnaround time of even simple applications [12] can take tens of minutes, with HLS, programmers can follow a rapid development cycle. Programmers can write code in C and leverage fast software simulation to verify the functional correctness. The debug turnaround time for such a correctness verification cycle can take as few as just one second instead of tens of minutes, 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 results (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 for a simple design or a component in a modular design. Thanks to the advances in HLS scheduling algorithms [13–17] and timing optimizations [18–21], HLS can not only shorten the development cycle, but also generate programs that are often competitive in cycle count [22], and more recently in clock frequency as well [19, 21]. Moreover, FPGA vendors provide host drivers and communication interfaces for kernels designed in HLS [23, 24], further reducing programmers' burden to integrate and offload workload to FPGA accelerators.

However, not all programs are created equal for HLS. Data-parallel programs can be easily programmed following the sequential C semantics, which enables such applications to be quickly designed and iterated in the fast correctness verification cycle and QoR tuning cycle. In contrast, task-parallel programs are not supported by the native C semantics, and the productivity provided by current HLS tools is 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 necessary. For example, for an accelerator with PEs connected through a simple on-chip network, a network node needs to forward packets based on their content (header) and the availability of output ports. Without an API to read packets without consuming them (i.e., "*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 is 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 software simulation does not support Cannon's algorithm [25] because its sequential execution of tasks cannot correctly simulate feedback loops in data paths, while Intel OpenCL simulator does not support more than 256 concurrent kernels [24]. Unavailability of 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 do not support hierarchical code generation for task-parallel programs. Instead, they treat all tasks as a monolithic design and process each instance of the same task as if they were different. For designs that instantiate the same task many times (e.g., in a systolic array), this leads to repetitive compilation on each task and unnecessarily slows down code generation. Programmers can manually synthesize tasks

2576-2621/21/$31.00 ©2021 IEEE
DOI 10.1109/FCCM51124.2021.00032

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.

Limited productivity support 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 [3] and stencil computation [9], task-parallel implementations show better scalability and higher frequency than their data-parallel counterparts due to the localized communication pattern [26]. In fact, at least 6 papers [11, 27–31] among the 28 research papers published in the ACM FPGA 2020 conference use task-parallel implementation with HLS, and another 3 papers [32–34] 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)[1], 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.
- **Unconstrained software simulation**: We demonstrate that our proposed simulator can correctly simulate task-parallel programs that existing software simulators fail to simulate. Moreover, the correctness verification cycle can be shortened 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.
- **Fully automated open-source framework**: TAPA is open-source at `https://github.com/UCLA-VAST/tapa/`.

Table I summarizes the related work. Among all general HLS tools (Section VI-A) and streaming frameworks (Section VI-B): ① None of them supports peeking in their kernel APIs; only Intel HLS `stream` and Vivado HLS `axis` support transactions; only Merlin allows the accelerator kernel to be called from the host as if it is a C/C++ function. ② Vivado HLS, Merlin, and both streaming frameworks (ST-Accel [36] and Fleet [37]) execute tasks sequentially for simulation, which works on limited applications, while others launch one thread per task instance, which does not scale well. ③ All general 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. To the best of our knowledge, TAPA is the only work that provides convenient programming interfaces, unconstrained software simulation, and hierarchical code generation for general task-parallel programs on FPGAs using HLS.

[1]While a prior work TAPAS [35] and our work TAPA share similarity in name, our work focuses on statically mapping tasks to hardware, yet TAPAS specializes in dynamically scheduling tasks.

TABLE I: Summary of related work.

Related Work	Programmability: Peeking	Programmability: Transaction	Programmability: Host Iface.	Software Simulation	RTL Code Generation
Fleet [37]	No	No	N/A	Sequential	N/A
Intel HLS (`pipe`)	No	No	N/A	Multi-thread	Monolithic
Intel HLS (`stream`)	No	Yes	N/A	Multi-thread	Monolithic
Intel OpenCL	No	No	OpenCL	Multi-thread	Monolithic
LegUp [38, 39]	No	No	N/A	Multi-thread	Monolithic
Merlin [40]	No	No	C++	Sequential	Monolithic
ST-Accel [36]	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

II. Background

A. *Task-Parallel Program*

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 [41]), 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 [9–11], graph processing [42–45], and network switching [33]. Task-parallel programs are often described using dataflow models [46–50], where tasks are called *processes*. Processes communicate only through unidirectional *channels*. Data exchanged between channels are called *tokens*. In this paper, we borrow the terms *channel* and *token*, and 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 [35, 39, 51] in our future work.

B. *A Motivating Example*

An on-chip ring network is a commonly used topology to provide all-to-all interconnection among many task-parallel processing elements (PE) in a single FPGA accelerator, which is particularly useful in graph processing [52–58] where each vertex may be connected to any other vertices. A ring network has the advantages of simplicity and high routability, but implementing a customized ring network in HLS faces several issues that make such designs verbose to write, hard to read, and error-prone. In this section, we use a simplified real-world design to illustrate the productivity issues for implementing such a ring network in HLS, which serves as a motivating example for our work.

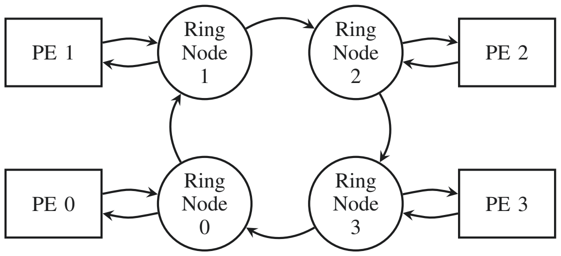

Fig. 1: An accelerator with 4 PEs connected via a ring network.

Fig. 1 shows an example where PEs in an accelerator are interconnected via a ring network. In this example, network nodes form a cyclic ring, and each ring node is connected to a PE via a bidirectional link. Each PE can send packets to other PEs through its associated node, specifying its destination PE in the packet header. Each node forwards packets either to its next node or to its associated PE, based on the packet header. We assume packets are sent infrequently and channels between nodes are provisioned so that they will never be full. Furthermore, we would like to insert packets from PEs to the network ASAP so that PEs will not stall due to back pressure from the ring nodes. While such a ring node can be written using Vivado HLS (Listing 1), we found that the followings are missing or hard-to-use in the HLS tools and significantly degrade the productivity.

1) Peeking: Peeking is defined as reading a token from a channel without consuming it. Compared with the normal destructive read, peeking is non-destructive because the token may be read many times. For example, in our ring network, when Node 1 receives incoming packets from both PE 1 (via `pe_in`) and Node 0 (via `node_in`), it will forward the packet from PE 1 to Node 2 (via `node_out`) to prevent PE 1 from being stalled due to back pressure. In the same clock cycle, the packet from Node 0 cannot be forwarded unless the destination of that packet is PE 1 (via `pe_out`), because we cannot write two tokens to the same output channel (`node_out`) in the same clock cycle. This requires us to conditionally read tokens based on the content of tokens. Without a `peek` API, one has to manually maintain a buffer for the incoming values, as shown in Line 7–15 of Listing 1. 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 ring network example, we consider the whole accelerator execution as a logical communication transaction, and let each PE control the termination of each `RingNode`, as shown in Line 11 of Listing 1. Without an `eot` API, one has to manually add a special bit to the data structure to indicate the *end-of-transaction* (Line 1–4 of Listing 1). Note that the `Pkt` struct may be used elsewhere, thus it may be infeasible to add the `eot` bit directly to the `Pkt` struct. Moreover, determining the end of transaction must be a peek operation; otherwise, the HLS compiler will be unable to schedule the exit condition in the first stage of pipeline, leading to II greater than 1. This further complicates the HLS implementation (Listing 1).

3) System integration: To offload computation kernel from the host CPU to PCIe-based 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 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. This includes manually setting up environmental variables for simulation, creating, and maintaining OpenCL `Context`, `CommandQueue`, `Program`, `Kernel`, etc. data structures [59]. Task-parallel accelerators often make the situation worse because the parallel tasks are often described as distinct OpenCL kernels [24], which significantly increases the programmers' burden on managing multiple kernels in the host-kernel interface. In our experiments, 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 want 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 allows programmers to instantiate tasks by invoking each of them sequentially [23]. While this is very concise to write (Listing 2), it leads to incorrect simulation results because the communication between a ring node and its corresponding PE is bidirectional, yet sequential execution can only send tokens from nodes to PEs because of their invocation order. This problem was also pointed out in [60]. In order to run software simulation correctly, the programmer can change the source code to run tasks in multiple threads, but doing so requires the same piece of task instantiation code to be written twice for synthesis and simulation, reducing productivity. While there exist other tools (e.g. [24]) that can run tasks in parallel threads and do not have the same correctness problem, we will show in Section V-D that such simulators do not scale well when the number of task instances increases.

5) RTL code generation: In our ring network example, the same ring node is instantiated many times. While state-of-the-art HLS compilers can recognize multiple instances of the same function and reuse HLS results for regular non-task-parallel programs, task-parallel programs are always treated as a monolithic one. This means instances of the same task in a task-parallel program are treated as if they were different, possibly in order to explore different communication interfaces of each instance. This significantly elongates the code generation time when the number of instances is large (Section V-E). We can manually do hierarchical code generation, i.e., synthesize each task separately and connect 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 and address these challenges by providing convenient programming interfaces, unconstrained software simulation, and hierarchical

```
struct PktEoT {
  Pkt pkt;                Auxiliary struct for termination control;
  bool eot;               eot stands for "end of transaction".
};
void RingNode(stream<Pkt>& node_in,  stream<PktEoT>& pe_in,
              stream<Pkt>& node_out, stream<Pkt>& pe_out) {
  Pkt node_pkt;                    Manually maintained input
  bool node_pkt_valid = false;     buffers to implement non-
  PktEoT pe_pkt;                   destructive read (i.e., peek).
  bool pe_pkt_valid = false;
  while (!(pe_pkt_valid && pe_pkt.eot)) {
    if (!pe_pkt_valid)                                 Manually
      pe_pkt_valid = pe_in.read_nb(pe_pkt);            update
    if (!node_pkt_valid)                               buffers.
      node_pkt_valid = node_in.read_nb(node_pkt);
    if (pe_pkt_valid) {
      node_out.write(pe_pkt.pkt);
      pe_pkt_valid = false;
      if (node_pkt_valid && IsForThisNode(node_pkt)) {
        pe_out.write(node_pkt);
        node_pkt_valid = false;
      }
    } else if (node_pkt_valid) {
      Pkt pkt = node_pkt;
      node_pkt_valid = false;
      (IsForThisNode(pkt) ? pe_out : node_out).write(pkt);
    }
  }  // Highlighted are destructive read operations and
}    // non-destructive read (peek) operations.
```

Listing 1: Ring network node written in Vivado HLS.

```
void Kernel(...) {
  stream<Pkt, 2> node_0_1, node_1_2, ...
  stream<Pkt, 2> from_pe_0, to_pe_0, from_pe_1, to_pe_1, ...
  // Instantiates other channels...
#pragma HLS dataflow
  RingNode(node_0_1, node_1_2, from_pe_1, to_pe_1);
  RingNode(node_1_2, node_2_3, from_pe_2, to_pe_2);
  // Instantiates other ring nodes and PEs...
}
```

Listing 2: Accelerator task instantiation in Vivado HLS.

code generation.

III. TAPA Programming Model and Interfaces

A. Hierarchical Programming Model

TAPA uses a hierarchical programming model. 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. Correspondingly, the instantiated tasks are the *children tasks* of their parent, which may be parents of their own children. Each channel must be connected to exactly two tasks. 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 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*, and waits until all its children tasks finish. One of the tasks is designated as the *top-level task*, which defines the communication interfaces external to the FPGA accelerator, i.e., the *system integration interface*.

B. Convenient Programming Interfaces

1) Communication Interface: TAPA provides separate communication APIs for the producer side and the consumer side, which use `ostream` and `istream` as the interfaces, respectively. 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 (destructive `read`), or duplicate the head of token without removing it (non-destructive read, a.k.a., `peek`), if the channel is not empty. Read, peek, and write operations can be blocking or non-blocking.

```
 1  void RingNode(istream<Pkt>& node_in,  istream<Pkt>& pe_in,
 2                ostream<Pkt>& node_out, ostream<Pkt>& pe_out) {
 3    while (!pe_in.eot()) {
 4      if (!pe_in.empty()) {
 5        node_out.write(pe_in.read());
 6        if (!node_in.empty() && IsForThisNode(node_in.peek()))
 7          pe_out.write(node_in.read());
 8      } else if (!node_in.empty()) {
 9        Pkt pkt = node_in.read();
10        (IsForThisNode(pkt) ? pe_out : node_out).write(pkt);
11      }
12    }  // Highlighted are destructive read operations and
13  }    // non-destructive read (peek) operations.
```

Listing 3: Ring network node written in TAPA.

```
void Kernel(...) {
  channel<Pkt, 2> node_0_1, node_1_2, ...
  channel<Pkt, 2> from_pe_0, to_pe_0, from_pe_1, to_pe_1, ...
  // Instantiates other channels...
  task()
    .invoke(RingNode, node_0_1, node_1_2, from_pe_1, to_pe_1)
    .invoke(RingNode, node_1_2, node_2_3, from_pe_2, to_pe_2)
    // Instantiates other ring nodes and PEs...
}
```

Listing 4: Accelerator task instantiation in TAPA.

A special token denoting end-of-transaction (EoT) is available to all channels. A process can "close" a channel by writing an EoT token to it, and a process can "open" a channel by reading an EoT token from it. A process can also test if a channel is closed, which is a non-destructive read operation to the channel (`eot`). 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, for example, in Line 3 of Listing 3. Listing 3 shows an example of how the communication interfaces are used in TAPA, which implements the same functionality as Listing 1, but with 55% fewer lines due to the absence of the auxiliary `struct` for end-of-transaction token and the manually maintained input buffer that implements peek operations.

2) Instantiation Interface: A parent task can instantiate channels and tasks using the instantiation interface. Channels are instantiated using `channel<type,capacity>`. For example, `channel<Pkt,2>` instantiates a channel with capacity 2, and data tokens transmitted using this channel have type `Pkt`. Tasks are instantiated using `task::invoke`, with the first argument being the task function and the rest of arguments being the arguments to the task instance. This is consistent with `std::invoke` in the C++ standard library. Listing 4 shows how channels and tasks are instantiated in TAPA.

3) System Integration Interface: 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 kernel binaries.

IV. TAPA Framework Implementation

A. Software Simulation

State-of-the-Art Approach: There are two state-of-the-art approaches to run software simulation for task-parallel applications: the sequential approach and the multi-thread approach. A sequential simulator invokes tasks sequentially in the invocation order [23]. Sequential simulators are fast, but cannot correctly simulate the capacity of channels and applications with tasks communicating bidirectionally, as discussed in Section II-B. 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 inefficient context switch handled by the operating system. The FLASH simulator [60, 61] proposed an alternative to the above, which uses HLS scheduling information to create an interleaving execution of all tasks. Note that although FLASH is also single-threaded, it is different from a sequential simulator because it interleaves tasks via source-to-source transformation while a sequential simulator does not. Compared with a sequential simulator, FLASH is on average 1.7× slower [61], due to additional scheduling information being taking into consideration for cycle-accurate modeling. Besides, 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, we propose an approach that uses collaborative coroutines [62, 63] instead of preemptive threads for each task. Note that fast and/or cycle-accurate debugging in general [64] 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/C++ [65]. Coroutines [66] are routines that execute collaboratively; more specifically, coroutines can be explicitly suspended and resumed. A coroutine can invoke subroutines and suspend from and resume to any subroutine [63]. A context switch between coroutines takes only 26ns on modern CPUs [63], while a preemptive thread context switch takes 1.2~2.2µs [67], which is two orders of magnitude slower.

TAPA leverages coroutines to perform software simulation as follows. When a task is instantiated, a coroutine is launched but suspended immediately. 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. Moreover, the coroutines can be distributed in a thread pool. The thread pool launches one thread per CPU core and can bind the thread to the corresponding core, which prevents the threads from preemption against each other. This improves simulation parallelism without introducing high context switch overhead as in the multi-thread simulators. We will show in Section V-D that the coroutine-based simulator outperforms the existing simulators by 3.2× on average. TAPA software simulator is implemented as a C++ library, which can be compiled by any compatible C++ compiler.

B. 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. This can lead to a significant amount of repeated work. For example, the dataflow architecture generated by a stencil accelerator compiler, SODA [7, 9], is highly modularized, and has many functionally identical modules. However, both the Vivado HLS and Intel FPGA OpenCL backends generate RTL code for each 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, doing so defeats the purpose of using HLS for high productivity. 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 invoking the vendor tools in parallel for each task. On average, TAPA reduces HLS compilation time by 4.9× (Section V-E).

Fig. 2 shows how RTL code is generated by TAPA, which is composed of four steps. First, 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 the capacity of each channel. Source-to-source transformation is applied in this step to insert HLS pragmas where necessary (e.g., to generate proper RTL interfaces). Then, the vendor HLS tool is used to

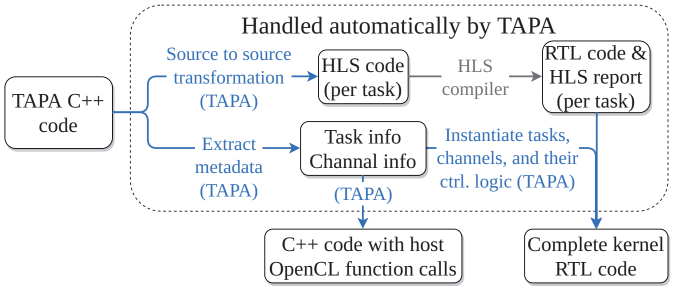
Handled automatically by TAPA
TAPA C++ code
Source to source transformation (TAPA)
HLS code (per task)
HLS compiler
RTL code & HLS report (per task)
Extract metadata (TAPA)
Task info Channal info
Instantiate tasks, channels, and their ctrl. logic (TAPA)
(TAPA)
C++ code with host OpenCL function calls
Complete kernel RTL code

Fig. 2: TAPA code generation. The host-kernel interface code is generated together with the kernel RTL code using metadata of the top-level task.

generate RTL code and HLS report for each task. While TAPA uses libraries to implement kernel APIs extensively, e.g., for `read`, `write`, and the end-of-transaction bit, not all APIs, e.g., peeking, can be implemented as libraries, due to the lack of support from the HLS scheduler. To support peeking, TAPA adds a scalar argument to each `istream`, and connect this port to the output of first-word-fall-through FIFO when the RTL code is assembled in the next step.

Using the metadata extracted in the first step, TAPA assembles the per-task RTL code to create the complete kernel. In this step, for each parent task, TAPA instantiates the children tasks and channels, and generates a small state machine that controls start of the children tasks and termination of the parent task. Finally, TAPA packages the assembled RTL code to a format that the vendor implementation tool can recognize (`xo` file for Vitis).

V. Evaluation

We prototype TAPA on Xilinx devices using Vivado HLS as the backend; support for Intel devices will be added later. 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 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.

A. Benchmarks

Table II summarizes the benchmarks used in this paper. 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 not only guarantees source codes to all tools are functionally equivalent, but also makes all tools generate consistent quality of results (QoR), which enables fair comparison of tool run time. Note that we aim to compare the productivity of the HLS tools, not QoR (although we want to make sure there is no QoR degradation). In particular, we were unable to guarantee that the generated RTL codes have exactly the same cycle-accurate behavior without having access to the HLS compiler's scheduling algorithm. For example, the bucket sort network 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, the manually maintained buffer forces an additional latency of 1 cycle at each network stage. The shallower pipeline makes TAPA use 40% fewer LUTs and 39% fewer FFs for `network`. For other benchmarks, TAPA uses 0.4% fewer LUTs and 1% fewer FFs on average. This shows that the additional APIs provided by TAPA does not add resource overhead.

TABLE II: Benchmarks used in this paper. Each task may be instantiated multiple times, so task instance count (#Inst.) and channel count (#Chan.) are greater than task count (#Task).

Benchmark	Application	#Task	#Inst.	#Chan.
`cannon`	Cannon's algorithm [25]	5	91	344
`cnn`	VGG [68] convolutional network [3]	14	209	366
`gaussian`	Gaussian stencil filter [9]	15	564	1602
`gcn`	Graph convolutional network [52]	5	12	25
`gemm`	General matrix multiplication [3]	14	207	364
`network`	Bucket sort w/ Omega network [69]	3	14	32
`page_rank`	PageRank citation ranking [54]	4	18	89

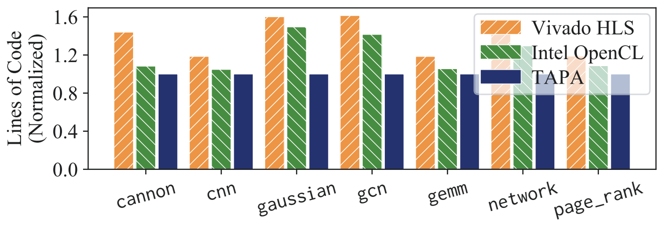

Fig. 3: LoC comparison for kernel code. Lower is better.

B. 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 concisely. Fig. 3 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.

C. Lines of Host Code

The host code used in the benchmarks contains a minimal test bench 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 4 shows the lines of host code comparison. On average, the length of host code is reduced by 51%.

D. Software Simulation Time

Fig. 5 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

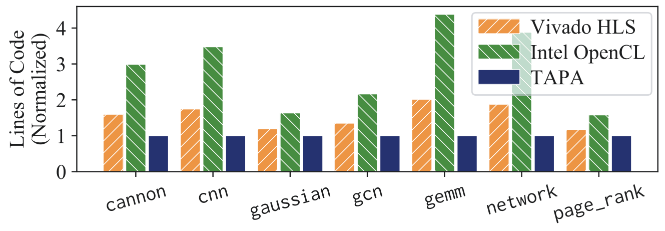

Fig. 4: LoC comparison for host code. Lower is better.

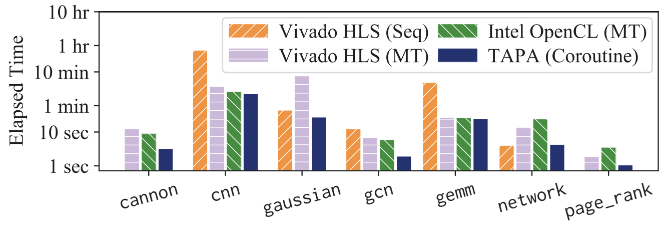

Fig. 5: Simulation time in log scale. Lower is better. Sequential simulator fails to simulate cannon and pagerank correctly. Intel OpenCL multi-thread simulator cannot simulate gaussian due to its large number of task instances.

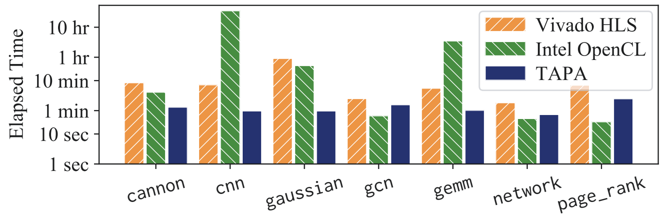

Fig. 6: RTL code generation time in log scale. Lower is better.

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 by the simulator (256). However, the multi-thread 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). Although the coroutine-based TAPA simulator is not always the fastest simulator for all benchmarks, the worst-case slowdown is only 6%, which is not significant in comparison with the multi-thread simulator, which can be 11× slower. On average, TAPA is 3.2× faster than other simulators.

E. RTL Code Generation Time

Fig. 6 shows the RTL code generation time comparison. Thanks to the hierarchical programming model and modularized code generator, TAPA shortens 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 run HLS for each task instance; ② TAPA runs HLS in parallel on multi-core machines.

VI. Related Work

Table I on Page 2 shows a brief summary of the related HLS tools. Section VI-A presents more details about these tools. Two domain-specific streaming frameworks are discussed in Section VI-B. *SystemC* and *pthread* are two well-known alternative API paradigms that support task-parallel programs. We will discuss and compare them with TAPA in Section VI-C.

A. HLS Support for Task-Parallel Programs

Intel HLS supports two different inter-task communication interfaces: pipe and stream. pipe implements a simple FIFO interface with data, valid, and ready signals, while stream implements an Avalon-ST interface that supports transactions. Tasks are instantiated using launch and collect.

Intel FPGA OpenCL supports the simple FIFO interface via two sets of APIs, i.e., standard OpenCL pipe and Intel-specific channel. Tasks are instantiated by defining OpenCL __kernels, which forces instances of the same task to be synthesized separately as different OpenCL kernels.

Vivado (Vitis) HLS provides two different streaming interfaces: ap_fifo and axis. ap_fifo generates the simple FIFO interface. Tasks are instantiated by invoking the corresponding functions in a dataflow region (Listing 2). axis 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.

Xilinx OpenCL supports standard OpenCL pipe, which generates AXI-Stream interfaces similar to Vivado HLS axis, but pipe does not provide APIs to support transactions.

LegUp supports the simple FIFO interface via FIFO. Tasks are instantiated using pthread API (Section VI-C).

Merlin [40] 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 HLS tools' programming interfaces.

Their limitations are summarized in Table I on Page 2. Note that a common limitation of HLS tools (including TAPA) is that they can not *guarantee* the software description produces deterministic output sequences for task-parallel programs. For instance, the emptiness test to an input channel is prone to breaking determinism, yet it is available to all HLS tools for performance and expressiveness reasons: merging two input channels round-robin using non-blocking reads would produce an output sequence determined by the relative arrival order of the input tokens. An implication of non-determinism is we cannot assert that a program is deadlock-free just because its simulation succeeds. This is different from deterministic programs, e.g., Kahn process networks [47], whose successful simulation generally implies deadlock-free on-board execution. For applications that can be efficiently written without breaking determinism, e.g., streaming applications, there are dedicated frameworks developed specifically for them, which are discussed in the next section.

B. Streaming Framework

ST-Accel [36] is a high-level programming platform 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.

Fleet [37] 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 [70].

TAPA aims to support more general task-parallel applications beyond streaming.

C. 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 [71, 72]. The official open-source SystemC simulator implementation uses coroutines without thread pooling. Some HLS tools support a subset of untimed SystemC as the input [23]. SystemC supports task-parallel programs natively via the `SC_MODULE` constructs and `tlm_fifo` interfaces, which supports peeking. While SystemC supports peeking FIFOs and coroutine-based simulation for task-parallel programs, it is limited by its special and verbose coding style. Listing 5 shows the example discussed in Section II-B written in SystemC. Compared with other C-like HLS languages, SystemC is more verbose and less productive due to its special language constructs: for TAPA code snippets shown in Listing 3 and Listing 4, the equivalent SystemC kernel code would be 86% longer. On the host side, SystemC generates the main function in `sc_main` by itself for simulation, and programmers need to spend time incorporating the SystemC test bench with other parts of their program. This is not a problem if the whole system is defined by the kernel in SystemC, e.g., as in embedded systems, but in data center applications where the FPGA accelerator is only part of the system, this introduces non-trivial complication.

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 each argument needs to be manually packed and passed. Listing 6 shows an example using the accelerator discussed in Section II-B. Compared with the `invoke` API used by TAPA, the pthread APIs require more effort to program: for TAPA code snippets shown in Listing 3 and Listing 4, equivalent pthread-based code would be 2.4× long.

In summary, while the API alternatives do exist in their own domains, they are more verbose and thus less productive compared with TAPA for task-parallel FPGA acceleration.

```
SC_MODULE(RingNode) {
  sc_port<tlm_fifo_get_if<Pkt>> node_in;
  sc_port<tlm_fifo_get_if<PktEoT>> pe_in;
  sc_port<tlm_fifo_put_if<Pkt>> node_out, pe_out;
  SC_CTOR(RingNode) { SC_THREAD(thread); }
  void thread() { while (...) {...} }
};
SC_MODULE(Kernel) {
  tlm_fifo<Pkt> node_0_1{/*depth=*/2}, node_1_2{2}, ...
  // Other channels...
  RingNode node1, node2, ...
  // Other tasks...
  SC_CTOR(Kernel) {
    node1.node_in(node_0_1);
    node1.node_out(node_1_2);
    // Other argument bindings...
  }
};
```

Listing 5: SystemC TLM API example.

```
struct RingNode_Arg {
  FIFO<Pkt>* node_in, node_out, pe_out;
  FIFO<PktEoT>* pe_in;
};
void RingNode(void* arg) {
  FIFO<Pkt>* node_in = ((RingNode_arg*)arg)->node_in;
  // Unpack other arguments...
  while (...) {...}
  pthread_exit(NULL);
}
void Kernel(...) {
  FIFO<Pkt> node_0_1, node_1_2, ...
  // Instantiate other channels...
  RingNode_Arg node1_arg, node2_arg, ...
  node1_arg.node_in = &node_0_1;
  // Pack other arguments...
  pthread_t node1_pid, node2_pid, ...;
  pthread_create(&node1_pid, NULL, RingNode, &node1_arg);
  // Create other threads...
  pthread_join(&node1_pid, NULL);
  // Join other threads...
}
```

Listing 6: Pthread API example.

VII. 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: on average, ① its enhanced programming interface helps to reduce the lines of kernel code by 22%, ② its unified system integration interface reduces the lines of host code by 51%, ③ its coroutine-based software simulator shortens the correctness verification development cycle by 3.2×, ④ its modularized code generation approach shortens the QoR tuning development cycle by 6.8×. 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 dynamic tasks on FPGAs.

Acknowledgment

The authors would like to thank the anonymous reviewers and our labmate, Linghao Song, for their valuable comments and helpful suggestions. This work is partially supported by a Google Faculty Award, the NSF RTML program (CCF-1937599), NIH Brain Initiative (U01MH117079), the Xilinx Adaptive Compute Clusters (XACC) program, and CRISP, one of six JUMP centers.

REFERENCES

[1] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *TCAD*, 2011.

[2] X. Wei, Y. Liang, and J. Cong, "Overcoming Data Transfer Bottlenecks in FPGA-based DNN Accelerators via Layer Conscious Memory Management," in *DAC*, 2019.

[3] J. Cong and J. Wang, "PolySA: Polyhedral-Based Systolic Array Auto-Compilation," in *ICCAD*, 2018.

[4] Y.-H. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, and Z. Zhang, "HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing," in *FPGA*, 2019.

[5] H. R. Zohouri, A. Podobas, and S. Matsuoka, "Combined Spatial and Temporal Blocking for High-Performance Stencil Computation on FPGAs Using OpenCL," in *FPGA*, 2018.

[6] M. Koraei, O. Fatemi, and M. Jahre, "DCMI: A Scalable Strategy for Accelerating Iterative Stencil Loops on FPGAs," *TACO*, vol. 16, no. 4, 2019.

[7] Y. Chi and J. Cong, "Exploiting Computation Reuse for Stencil Accelerators," in *DAC*, 2020.

[8] J. de Fine Licht, A. Kuster, T. De Matteis, T. Ben-Nun, D. Hofer, and T. Hoefler, "StencilFlow: Mapping Large Stencil Programs to Distributed Spatial Computing Systems," in *CGO*, 2021.

[9] Y. Chi, J. Cong, P. Wei, and P. Zhou, "SODA : Stencil with Optimized Dataflow Architecture," in *ICCAD*, 2018.

[10] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz, "Programming Heterogeneous Systems from an Image Processing DSL," *TACO*, vol. 14, no. 3, 2017.

[11] J. Li, Y. Chi, and J. Cong, "HeteroHalide: From Image Processing DSL to Efficient FPGA Acceleration," in *FPGA*, 2020.

[12] UCLA-VAST, "TAPA Sample Applications." [Online]. Available: https://github.com/UCLA-VAST/tapa/tree/master/apps

[13] J. Cong and Z. Zhang, "An Efficient and Versatile Scheduling Algorithm Based On SDC Formulation," in *DAC*, 2006.

[14] J. Cheng, S. T. Fleming, Y. T. Chen, J. H. Anderson, and G. A. Constantinides, "EASY: Efficient Arbiter SYnthesis from Multi-threaded Code," in *FPGA*, 2019.

[15] J. Cheng, L. Josipović, G. A. Constantinides, P. Ienne, and J. Wickerson, "Combining Dynamic & Static Scheduling in High-level Synthesis," in *FPGA*, 2020.

[16] H. Hsiao and J. Anderson, "Thread Weaving: Static Resource Scheduling for Multithreaded High-Level Synthesis," in *DAC*, 2019.

[17] A. Haj-Ali, Q. Huang, W. Moses, J. Xiang, K. Asanovic, J. Wawrzynek, and I. Stoica, "AutoPhase: Juggling HLS Phase Orderings in Random Forests with Deep Reinforcement Learning," in *MLSys*, 2020.

[18] Y. T. Chen, J. H. Kim, K. Li, G. Hoyes, and J. H. Anderson, "High-Level Synthesis Techniques to Generate Deeply Pipelined Circuits for FPGAs with Registered Routing," in *FPT*, 2019.

[19] L. Guo, J. Lau, Y. Chi, J. Wang, C. H. Yu, Z. Chen, Z. Zhang, and J. Cong, "Analysis and Optimization of the Implicit Broadcasts in FPGA HLS to Improve Maximum Frequency," in *DAC*, 2020.

[20] L. Josipović, S. Sheikhha, A. Guerrieri, P. Ienne, and J. Cortadella, "Buffer Placement and Sizing for High-Performance Dataflow Circuits," in *FPGA*, 2020.

[21] L. Guo, Y. Chi, J. Wang, J. Lau, W. Qiao, E. Ustun, Z. Zhang, and J. Cong, "AutoBridge: Coupling Coarse-Grained Floorplanning and Pipelining for High-Frequency HLS Design on Multi-Die FPGAs," in *FPGA*, 2021.

[22] J. Cong, P. Wei, C. H. Yu, and P. Zhang, "Automated Accelerator Generation and Optimization with Composable, Parallel and Pipeline Architecture," in *DAC*, 2018.

[23] Xilinx, "Vivado Design Suite User Guide: High-Level Synthesis (UG902)," 2020.

[24] Intel, "Intel FPGA SDK for OpenCL Pro Edition: Programming Guide," 2020.

[25] H.-J. Lee, J. P. Robertson, and J. A. Fortes, "Generalized Cannon's Algorithm for Parallel Matrix Multiplication," in *ICS*, 1997.

[26] J. Cong, P. Wei, C. H. Yu, and P. Zhou, "Latte: Locality Aware Transformation for High-Level Synthesis," in *FCCM*, 2018.

[27] T. Young-Schultz, L. Lilge, S. Brown, and V. Betz, "Using OpenCL to Enable Software-like Development of an FPGA-Accelerated Biophotonic Cancer Treatment Simulator," in *FPGA*, 2020.

[28] V. Rybalkin and N. Wehn, "When Massive GPU Parallelism Ain't Enough: A Novel Hardware Architecture of 2D-LSTM Neural Network," in *FPGA*, 2020.

[29] A. Sohrabizadeh, J. Wang, and J. Cong, "End-to-End Optimization of Deep Learning Applications," in *FPGA*, 2020.

[30] J. De Fine Licht, G. Kwasniewski, and T. Hoefler, "Flexible Communication Avoiding Matrix Multiplication on FPGA with High-Level Synthesis," in *FPGA*, 2020.

[31] J. Jiang, Z. Wang, X. Liu, J. Gómez-Luna, N. Guan, Q. Deng, W. Zhang, and O. Mutlu, "Boyi: A Systematic Framework for Automatically Deciding the Right Execution Model of OpenCL Applications on FPGAs," in *FPGA*, 2020.

[32] H. Zeng and V. Prasanna, "GraphACT: Accelerating GCN training on CPU-FPGA heterogeneous platforms," in *FPGA*, 2020.

[33] P. Papaphilippou, J. Meng, and W. Luk, "High-Performance FPGA Network Switch Architecture," in *FPGA*, 2020.

[34] H. Chen, S. Madaminov, M. Ferdman, and P. Milder, "FPGA-Accelerated Samplesort for Large Data Sets," in *FPGA*, 2020.

[35] S. Margerm, A. Sharifian, A. Guha, A. Shriraman, and G. Pokam, "TAPAS: Generating Parallel Accelerators from Parallel Programs," in *MICRO*, 2018.

[36] Z. Ruan, T. He, B. Li, P. Zhou, and J. Cong, "ST-Accel: A High-Level Programming Platform for Streaming Applications on FPGA," in *FCCM*, 2018.

[37] J. Thomas, P. Hanrahan, and M. Zaharia, "Fleet: A Framework for Massively Parallel Streaming on FPGAs," in *ASPLOS*, 2020.

[38] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. Anderson, S. Brown, and T. Czajkowski, "LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems," in *FPGA*, 2011.

[39] J. Choi, S. D. Brown, and J. H. Anderson, "From Pthreads to Multicore Hardware Systems in LegUp High-Level Synthesis for FPGAs," *TVLSI*, vol. 25, no. 10, 2017.

[40] J. Cong, M. Huang, P. Pan, D. Wu, and P. Zhang, "Software Infrastructure for Enabling FPGA-Based Accelerations in Data Centers," in *ISLPED*, 2016.

[41] L. Dagum and R. Menon, "OpenMP: An Industry Standard API for Shared-Memory Programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, 1998.

[42] G. Dai, Y. Chi, Y. Wang, and H. Yang, "FPGP: Graph Processing Framework on FPGA A Case Study of Breadth-First Search," in *FPGA*, 2016.

[43] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang, "ForeGraph: Exploring Large-scale Graph Processing on Multi-FPGA Architecture," in *FPGA*, 2017.

[44] S. Zhou, R. Kannan, V. K. Prasanna, G. Seetharaman, and Q. Wu, "HitGraph: High-throughput Graph Processing Framework on FPGA," *TPDS*, 2019.

[45] Y. Wang, J. C. Hoe, and E. Nurvitadhi, "Processor Assisted Worklist Scheduling for FPGA Accelerated Graph Processing on a Shared-Memory Platform," in *FCCM*, 2019.

[46] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, vol. 21, no. 8, 1978.

[47] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," in *IFIP*, 1974.

[48] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," *IEEE*, vol. 75, no. 9, 1987.

[49] J. T. Buck, "Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model," Ph.D. dissertation, 1993.

[50] J. L. Peterson, "Petri Nets," *ACM Computing Surveys*, vol. 9, no. 3, 1977.

[51] M. Abeydeera and D. Sanchez, "Chronos: Efficient Speculative Parallelism for Accelerators," in *ASPLOS*, 2020.

[52] T. N. Kipf and M. Welling, "Semi-Supervised Classification with Graph Convolutional Networks," in *ICLR*, 2017.

[53] C. Deng, Z. Zhao, Y. Wang, Z. Zhang, and Z. Feng, "GraphZoom: A Multi-level Spectral Approach for Accurate and Scalable Graph Embedding," in *ICLR*, 2020.

[54] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web," Tech. Rep., 1998.

[55] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters," *Internet Mathematics*, vol. 6, no. 1, 2009.

[56] J. Mcauley, "Learning to Discover Social Circles in Ego Networks," in *NIPS*, 2012.

[57] Y. Chi, G. Dai, Y. Wang, G. Sun, G. Li, and H. Yang, "NXgraph: An Efficient Graph Processing System on a Single Machine," in *ICDE*, 2016.

[58] G. Dai, T. Huang, Y. Chi, J. Zhao, G. Sun, Y. Liu, Y. Wang, Y. Xie, and H. Yang, “GraphH: A Processing-in-Memory Architecture for Large-scale Graph Processing,” *TCAD*, 2018.

[59] Xilinx, “Vitis Accel Hello World Example.” [Online]. Available: https://github.com/Xilinx/Vitis_Accel_Examples/blob/21bb0cf788ace593c6075accff7f7783588ae8b4/hello_world/src/host.cpp#L58-L115

[60] Y. Chi, Y.-k. Choi, J. Cong, and J. Wang, “Rapid Cycle-Accurate Simulator for High-Level Synthesis,” in *FPGA*, 2019.

[61] Y.-k. Choi, Y. Chi, J. Wang, and J. Cong, “FLASH: Fast, ParalleL, and Accurate Simulator for HLS,” *TCAD*, 2020.

[62] A. L. de Moura and R. Ierusalimschy, “Revisiting Coroutines,” *TOPLAS*, vol. 31, no. 2, 2009.

[63] O. Kowalke, “Boost Library Documentation, Coroutine2,” 2014. [Online]. Available: https://boost.org/doc/libs/1_65_0/libs/coroutine2/doc/html/coroutine2/intro.html

[64] A. S. Jamal, E. Cahill, J. Goeders, and S. J. E. Wilton, “Fast Turnaround HLS Debugging using Dependency Analysis and Debug Overlays,” *TRETS*, vol. 13, no. 1, 2020.

[65] D. E. Knuth, *Fundamental Algorithms. The Art of Computer Programming 1*, 3rd ed., 1997.

[66] M. E. Conway, “Design of a Separable Transition-Diagram Compiler,” *Communications of the ACM*, vol. 6, no. 7, 1963.

[67] E. Bendersky, “Measuring context switching and memory overheads for Linux threads,” 2018. [Online]. Available: https://eli.thegreenplace.net/2018/measuring-context-switching-and-memory-overheads-for-linux-threads/

[68] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” in *ICLR*, 2015.

[69] D. H. Lawrie, “Access and Alignment of Data in an Array Processor,” *ToC*, vol. C-24, no. 12, 1975.

[70] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, “Chisel: Constructing Hardware in a Scala Embedded Language,” in *DAC*, 2012.

[71] T. Schmidt, G. Liu, and R. Dömer, “Exploiting Thread and Data Level Parallelism for Ultimate Parallel SystemC Simulation,” in *DAC*, 2017.

[72] M. K. Chung, J. K. Kim, and S. Ryu, “SimParallel: A High Performance Parallel SystemC Simulator Using Hierarchical Multi-threading,” in *ISCAS*, 2014.