PASTA: Programming and Automation Support for Scalable Task-Parallel HLS Programs on Modern Multi-Die FPGAs

Moazin Khatti, *Simon Fraser University*, moazin_khatti@sfu.ca
Xingyu Tian, *Simon Fraser University*, xingyu_tian@sfu.ca
Yuze Chi, *UCLA*, i@blaok.me
Licheng Guo, *UCLA*, lcguo@ucla.edu
Jason Cong, *UCLA*, cong@cs.ucla.edu
Zhenman Fang, *Simon Fraser University*, zhenman@sfu.ca

***Abstract*—In recent years, there has been increasing adoption of FPGAs in datacenters as hardware accelerators, where a large population of end users are software developers. While high-level synthesis (HLS) facilitates software programming, it is still challenging to scale large accelerator designs on modern datacenter FPGAs that often consist of multiple dies and memory banks. More specifically, routing congestion and extra delays on these multi-die FPGAs often cause timing closure issues and severe frequency degradation at the physical design level, which are difficult to digest and optimize for high-level programmers using HLS. One promising approach to mitigate such issues is to develop a high-level task-parallel programming model with HLS and physical design co-optimization. Unfortunately, existing studies only support a programming model where tasks communicate with each other via FIFOs, while many applications are not streaming friendly and many existing accelerator designs heavily rely on buffer based communication between tasks.**

In this paper, we take a step further to support a task-parallel programming model where tasks can communicate via both FIFOs and buffers. To achieve this goal, we design and implement the PASTA framework, which takes a large task-parallel HLS design as input and automatically generates a high-frequency FPGA accelerator via HLS and physical design co-optimization. First, we design a decoupled latency-insensitive buffer channel that supports memory partitioning and ping-pong buffering, which is compatible with the vendor Vitis HLS compiler. In the frontend, we develop an easy-to-use programming interface to allow end users to use our buffer channel in their applications. In the backend, we provide automatic coarse-grained floorplanning and pipelining for designs that use our proposed buffer channel. We test PASTA on a set of task-parallel HLS designs that use buffers for task communication and show an average of 36% (up to 54%) frequency improvement for large design configurations.

I. Introduction

In the past few years, there is a growing interest in adopting FPGAs in datacenters as hardware accelerators, due to their low power, high flexibility, performance and energy efficiency. Indeed, two major FPGA vendors, Altera and Xilinx, have been recently acquired by two major server CPU vendors, Intel and AMD [1], [2]. Moreover, major cloud service providers, such as AWS [3], Microsoft Azure [4] and Alibaba Cloud [5], have all started providing computing instances equipped with FPGAs. Since a large population of datacenter users are software developers, it is important to make software-oriented programming on datacenter FPGAs easy and efficient. While the continuous development of high-level synthesis (HLS) facilitates FPGA programming by software developers, it is still nontrivial to develop efficient large-scale accelerator designs on modern multi-die FPGAs in datacenters.

Specifically, modern datacenter FPGAs consist of a plethora of resources, including multiple DDR and/or HBM (high-bandwidth memory) banks and multiple dies connected via silicon interposers [6]. Furthermore, they often consist of IP cores with fixed locations, such as the Vitis platform region, PCIe and the HBM system, which consume a lot of resources around them and redirect nearby paths to take much longer routes [7]. When HLS designs are scaled up on these FPGAs to best utilize the available resources, they often encounter timing closure issues and experience severe clock frequency degradation, due to routing congestion and extra delays. Even worse, these issues happen at the physical design layer and are thus challenging for HLS programmers to digest and optimize.

One of the promising directions to address such problems is via HLS and physical design co-optimization: a user still programs FPGA accelerators in a high-level language and (implicitly or explicitly) passes metadata to the backend tool, while the backend tool automatically applies floorplanning and timing optimizations based on the metadata. For example, a recent study, TAPA/AutoBridge [7] [8], made an initial attempt to successfully mitigate these timing closure issues in a task-parallel HLS programming model where parallel tasks only communicate via latency-insensitive FIFO channels. The latency-insensitive nature of the communication channels allows the tool to automatically spread up the tasks across multiple small FPGA regions to alleviate local routing congestion and pipeline global communication wires between tasks to avoid long critical paths. Unfortunately, existing studies [7] [8] only support a task-parallel programming model where tasks can only communicate via latency-insensitive FIFO channels. However, a lot of applications are not friendly for streaming and a lot of existing FPGA accelerator designs heavily rely on on-chip buffer based communication (often via ping-pong buffering) [9] [10] [11] [12] [13].

In this paper, we take a step further and develop the PASTA framework to support a more general task-parallel HLS programming model where tasks can communicate with each other via both FIFO and buffer based latency-insensitive channels. PASTA takes a large-scale task-parallel HLS program as input and automatically generates a high-frequency and high-performance accelerator design on modern multi-die FPGAs via HLS and physical design co-optimization.

To support buffer based communication between tasks, we have to address the following new challenges. 1) We need to design a *latency-insensitive* buffer communication channel that is *decoupled* from tasks (i.e., producer and consumer hardware

modules), so that we can easily extract metadata about the tasks and (buffer) communication channels from the input HLS program and then apply coarse-grained floorplanning and pipelining optimizations in the physical design. 2) This buffer channel needs to support the *ping-pong buffering* feature to allow its producer and consumer tasks with separate finite state machines (FSMs) to concurrently write and read the buffer sections in a dataflow fashion. 3) It also needs to support *multidimensional arrays with memory partitioning* schemes such as *cyclic*, *block* and *complete* that are required by loop pipelining and unrolling optimizations in its producer and consumer tasks. 4) The I/O protocol of this buffer channel needs to be *compatible* with vendor HLS tools so that the HLS-based producer and consumer tasks can seamlessly interface with it. 5) All the above requirements would make the buffer channel design cumbersome to use. Therefore, we need to develop an easy-to-use programming interface for end users and automate the program transformations in the frontend tool. 6) We need to properly apply coarse-grained floorplanning and pipelining in the backend tool for task-parallel HLS programs that use our buffer communication channel, which would need automatic tuning and recompilation of the producer/consumer tasks due to the added pipelining latencies.

To address the above challenges, in PASTA, we first design a buffer channel abstraction that is latency-insensitive and decoupled from producer and consumer tasks, supports ping-pong buffering and memory partitioning, and is compatible with vendor Vitis HLS tool. This buffer channel includes 1) dual-port partitioned memory cores that store the actual data of each buffer section, 2) a free sections FIFO, which stores the tokens representing buffer sections that contain no valid data and are free to be written to by the producer task, and 3) an occupied sections FIFO, which stores the tokens representing buffer sections that contain valid data and are ready to be read by the consumer task. The producer and consumer tasks control the access of the buffer channel via the latency-insensitive token exchange. The memory cores use the `ap_memory` protocol and the FIFOs use the `ap_fifo` protocol, which are supported by Vitis HLS.

In the frontend, we design an easy-to-use programming interface where users can easily 1) customize a buffer channel's data type, number of ping-pong sections, memory partitioning scheme, and memory core type, using C++ templates, and 2) use the declared buffer channel in producer and consumer tasks using the C++ *Resource Acquisition Is Initialization (RAII)* idiom that automatically takes care of the cumbersome procedure calls and data dependency issue happening under the hood. Our frontend tool automatically extracts the task graph with the aforementioned task communication metadata, and transforms the code of the producer and consumer tasks that use our buffer interface to be synthesized by Vitis HLS.

At the backend, based on the extracted metadata, we first instantiate the buffer channels on hardware. Then, we apply coarse-grained floorplanning [7] to place the tasks and buffer channels into local FPGA regions to alleviate local routing congestion. Since reading from a memory core is more sensitive to the added pipelining latency, we place the buffer channel on the consumer task side. Finally, we add pipeline registers between the producer task and the buffer channel as they are placed in different regions. Unlike FIFO channels that automatically check for fullness and emptiness, the buffer channel (i.e., its memory cores) has no such mechanism: the added latency for the producer task to read the buffer (at times) would cause correctness issues. Therefore, we need to automatically modify the producer task code with the updated buffer access latency and invoke Vitis HLS to recompile it.

We test our PASTA tool on a set of benchmarks derived from Rodinia-HLS [9] and achieve an average of 36% (up to 54%) frequency improvement on AMD/Xilinx HBM-based Alveo U280 FPGA board. Moreover, we confirm with on-board execution tests that our improvements in frequency indeed translate to a similar execution time speedup.

In summary, this paper makes the following contributions:

1. Analysis of challenges and design of latency-insensitive buffer channel abstraction to support scalable task-parallel HLS programs where tasks communicate via buffers.
2. PASTA, an end-to-end programming and automation framework that supports scalable task-parallel HLS programs on modern multi-die FPGAs, where tasks can communicate with each other via both FIFOs and buffers.
3. Experimental results to demonstrate superior frequency and performance improvements using PASTA.

II. Background and Motivation

A. Modern Multi-Die FPGAs and Programming Challenge

To increase the amount of on-chip resources, modern datacenter FPGAs are often made with multiple dies connected via silicon interposers [6]. For example, Fig. 1(a) shows an overview of the AMD/Xilinx HBM-based Alveo U280 FPGA board. It consists of three super logic regions (SLRs, i.e., dies) that are connected via SLR crossings, which carry additional latencies. In addition, it often consists of IP cores with fixed locations, such as the HBM memory system, IO banks, and the Vitis platform region, shown in Fig. 1(a). These IP cores consume a lot of resources around them and would redirect nearby paths to take much more expensive routes [7].

As a result, for large-scale HLS designs on such multi-die FPGAs, the routing congestion and extra delays often cause timing closure issues and severe clock frequency degradation. As will be presented in Section VII-C, for a wide range of HLS designs that use 8 or more processing elements (PEs), their achieved clock frequency is between 150MHz and 200MHz, even though the target frequency is set to 300 MHz. To better understand the impact of such frequency degradation, assuming an FPGA accelerator design could achieve 60× speedup over its CPU baseline if it ran at 300MHz, now it could only achieve 30× to 40× speedup when running at 150MHz and 200MHz. Even worse, due to the gap between HLS design and physical design, it is challenging for HLS programmers to analyze and identify the root causes, and fix their HLS design to achieve better timing closure.

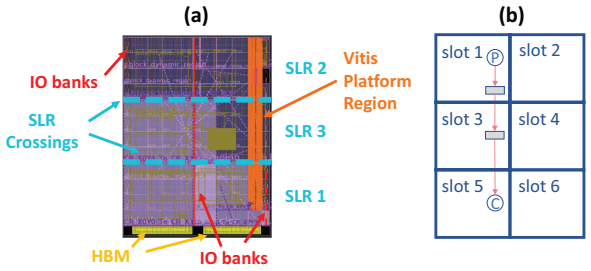

Fig. 1. Device overview of Alveo U280 FPGA and its slot representation.

Next, we present a high-level overview of the task-parallel HLS programming model and its floorplanning optimization [7], [8], as well as its limitation that motivates our work.

B. Task-Parallel Model and Motivational Example

In a task-parallel programming model, a program consists of a set of *tasks* that run in parallel and/or in dataflow and exchange data via *FIFO* or *buffer* based *communication channels*. Generally, a FIFO channel is used when data need to be sent and received in a pre-determined streaming fashion. In contrast, a buffer based channel allows random memory access from both producer and consumer sides. Moreover, a buffer based channel also enables partitioned parallel memory access and easier programming. Hence, it is common to see existing applications heavily rely on buffering. Fig. 2 shows a task-parallel implementation of the Pathfinder algorithm [14]. It is a dynamic programming based algorithm that finds a path on a 2D grid from the bottom of the grid to the top with the least accumulated weight. This algorithm iterates from the bottom most row to the top most row, and increases each point's weight by a function of the three points directly below it, as illustrated in Fig. 2(b).

To scale the accelerator design using task parallelism, as shown in Fig. 2(c), it divides the input grid into three parts, one for each PE (processing element). Fig. 2(a) shows its *task graph*, which consists of three load tasks, three compute tasks (named PEs), and one merge task. The load task loads tiles of data from off-chip memory one by one and outputs them via a *buffer channel*. This data is received by the PEs and processed row by row. Before processing each row, bordering data is sent to and received from the neighboring PEs via *FIFO channels*. Once all the PEs are done processing their part of the grid, the top most rows are sent to the merge task via another buffer channel. These last rows are received by the merge task, the solution is computed and written to off-chip memory.

In this motivational example, tasks communicate with each other via both *FIFO channels* and *buffer channels*.

C. Floorplanning for Task-Parallel HLS Programs

For a task-parallel HLS program, we can extract its actual tasks and distribute the tasks across multiple small FPGA regions for local placement and routing optimizations. Meanwhile, we can also extract the communication channels between tasks for global routing optimization by inserting pipeline registers for global long wires. There are two key requirements behind this idea. First, we need a task-parallel

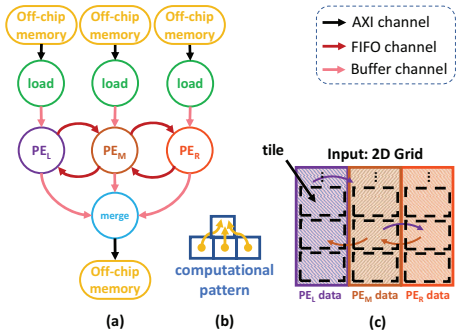

Fig. 2. Motivational example Pathfinder in task-parallel model: (a) task graph, (b) computation pattern of each point, (c) input 2D grid with partition for PEs.

programming model that decouples the actual tasks and communication channels, so that they can be easily extracted from the user HLS program. Second, the communication channel needs to be latency insensitive, so that we can add pipeline registers between this channel and its producer/consumer task.

Unfortunately, it is often nontrivial to develop a task-parallel program in vendor HLS that satisfies the above two requirements, especially when tasks communicate with each other via buffer channels. In vendor HLS, the declaration of communication channels (e.g., the declaration of ping-pong buffers with memory partitioning) and the usage of communication channels in producer and consumer tasks (e.g., the reads and writes of those partitioned ping-pong buffers) are often mingled together and hard to extract out. In addition, due to the programming style, the access of communication channels is often latency sensitive: adding extra latencies to a buffer channel access may cause correctness issues.

The recent TAPA/AutoBridge work [7], [8] has made a successful attempt. It uses a task-parallel HLS programming model that decouples the actual tasks and the latency-insensitive FIFO communication channels between tasks, which is compatible with Vitis HLS. After extracting the tasks and FIFO communication channels, it applies coarse-grained floorplanning and pipelining to improve the design frequency. Basically, it models a multi-die FPGA device as a grid of local slots based on the SLR boundaries and IP core locations: Fig. 1(b) shows an example of six slots for the Alveo U280 FPGA. Then, it formulates an integer linear programming (ILP) based algorithm to find a mapping of tasks to the local FPGA slots, such that wires crossing slots are minimized and the resource consumption within slots is kept within certain limits. As directed by the output of the algorithm, FIFOs that cross slot boundaries are pipelined (Fig. 1(b)). And the algorithm's output mapping is used to give constraints to the vendor implementation tool (i.e., Vivado).

D. Limitation of Existing Work

While TAPA/AutoBridge [7], [8] shows promising results of HLS and physical design co-optimization for large-scale accelerator designs on modern multi-die FPGAs, it is limited to task-parallel HLS programs in which tasks communicate via FIFOs only. However, a lot of applications do not have streaming friendly computation patterns and many existing

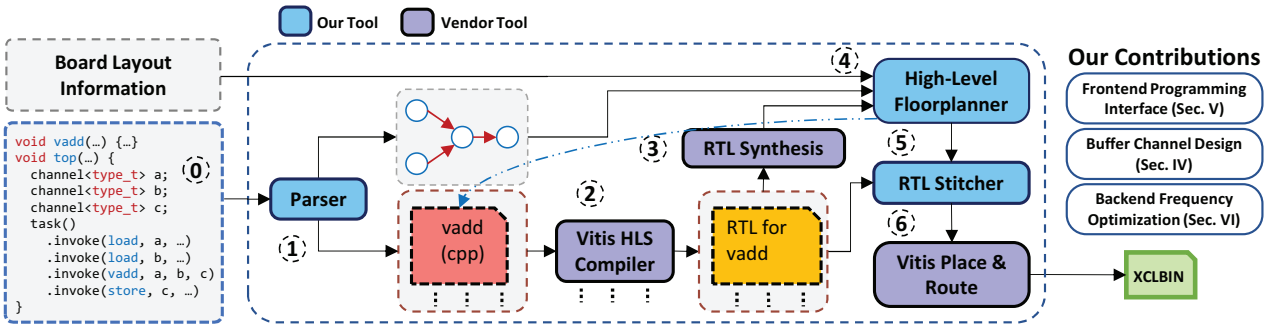

Fig. 3. An overview of our PASTA toolflow and our new contributions.

accelerator designs heavily rely on on-chip buffer based communication (often via ping-pong buffering) [9] [10] [11] [12] [13]. Moreover, some applications require both FIFO and buffer based communication channels, such as the Pathfinder example explained in Section II-B.

III. Overview of PASTA

In this work, our goal is to build an end-to-end programming and automation framework (called PASTA) that supports scalable task-parallel HLS programs on modern multi-die FPGAs, where tasks can communicate with each other via both FIFO and buffer based latency-insensitive communication channels. We build PASTA based on the open source TAPA/AutoBridge framework [7], [8] and focus on addressing the new challenges listed in Section I to enable buffer based communication.

A. Overall PASTA Toolfow

Fig. 3 shows the complete flow of our framework PASTA. A user writes their programs using a task-parallel HLS programming model which consists of parallel tasks that communicate via FIFOs and/or buffers. We will present more details on the frontend programming interface in Section V.

First, our parser goes through the input program and extracts all the task definitions (i.e., source code), buffer and FIFO declarations (including their metadata, such as width and depth of FIFO, size of the buffers, number of buffer dimensions, data type, and partition scheme) and constructs a task graph. The extracted task definitions are then transformed by our parser to generate the right interfaces. Second, each (transformed) task is compiled to RTL via Vitis HLS in parallel. Third, the generated RTL modules are synthesized via Vivado in parallel to get accurate resource consumption reports. Fourth, the resource reports and the task graph with its metadata are used by our high-level floorplanner to find a mapping of tasks to FPGA slots (based on input board layout information), such that the number of wires crossing SLRs is minimized and the resource utilization per slot is kept within certain limits. Based on this mapping, global channels of both FIFOs and buffers are pipelined. It may have a back loop to update the transformed HLS tasks to ensure the correctness after the pipelining step. Fifth, both global and local channels are then instantiated and stitched together with the tasks' RTL modules by our RTL stitcher. The final design is packed into an XO file and a constraint file is separately generated to capture the placement constraints given by our floorplanner. Finally, Vivado is invoked with these two files to perform placement and routing and generate the final design (i.e., xclbin file).

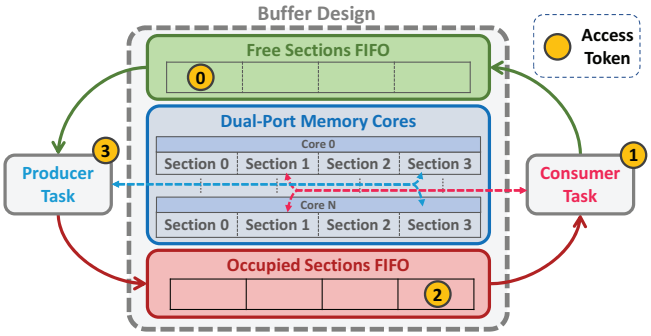

Fig. 4. Buffer channel abstraction with token-based access.

B. Our Contributions

In PASTA, we present the following new contributions. First, we design a buffer channel abstraction (Section IV) that is latency-insensitive and decoupled from producer and consumer tasks, supports ping-pong buffering and memory partitioning, and is compatible with the vendor Vitis HLS tool. Second, we design an easy-to-use programming interface to use our buffer channels and its corresponding frontend parser (Section V). Third, we implement a backend tool (Section VI) to automatically and appropriately place and pipeline the global buffer channels to improve the timing closure.

IV. Buffer Channel Abstraction

A. Overall Buffer Channel Design

Our proposed buffer channel design is shown in Fig. 4. It includes 1) a free sections FIFO, 2) an occupied sections FIFO, and 3) multiple dual-port memory cores to support memory partitioning, where each memory core consists of multiple sections to support ping-pong buffering. The producer task connects with each memory core via one of its ports, while the consumer task connects with the other port. To support ping-pong buffering, each memory core includes multiple sections, allowing the producer to write the next data tile into one of the free sections while the consumer reads one of the occupied (i.e., previously written) sections.

To enforce access control and correct ordering, access tokens are used. Naturally, there are as many tokens as there

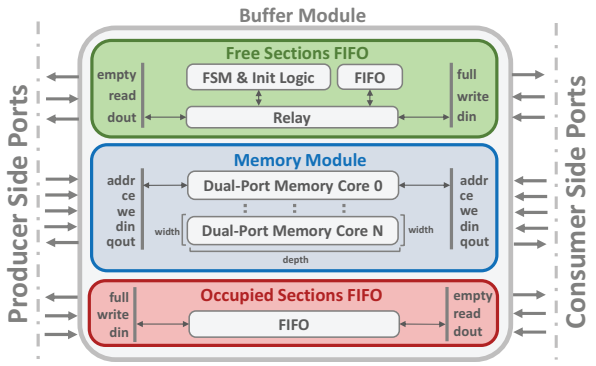

Fig. 5. Detailed buffer channel implementation.

are sections in the memory cores. The free sections FIFO stores the tokens of all the sections that do not contain any valid data and are ready to be written to by the producer task. The occupied sections FIFO contains tokens of all the sections that contain valid data and are ready to be read by the consumer task. The producer can read a token from the free sections FIFO, and access the corresponding section from the memory cores. Once the producer has written valid data to the accessed section, it writes the previously held token to the occupied sections FIFO. The consumer task reads that token from the occupied sections FIFO, accesses valid data from the corresponding section that was written by the producer. Once done, it writes the held token to the free sections FIFO, so that the producer can write to the corresponding section again.

Our design works on top of existing vendor HLS tools as it only uses `ap_fifo` and `ap_memory` protocols for the FIFOs and memory cores; both are supported by Vitis HLS.

B. Detailed Buffer Channel Implementation

Next, we discuss the detailed implementation of our buffer channel design, shown in Fig. 5.

Free sections FIFO. This FIFO is read by the producer to get access tokens for free sections and is written to by the consumer to release token access for free sections. While this module looks like a FIFO from the outside interface, internally, it consists of multiple components. Before it is ready for use, it needs to be initialized with all the sections' tokens. While one could manually do this initialization from the consumer side, this is cumbersome and could fail if the accelerator runs twice. Therefore, we decide to perform this initialization on RTL reset by creating an initialization logic inside the free sections FIFO module. The module contains an FSM with two possible states: *init* and *done*. On RTL reset, the module is put in the *init* state in which the relay isolates the FIFO from outside access by asserting the `full` and `empty` signals. While the FIFO is isolated from outside, it is connected to the initialization logic which performs the initialization. Once it is done, the state transitions to the *done* state in which the FIFO is connected to the outside interface.

Occupied sections FIFO. This is a simple FIFO module inherited from TAPA [15] that stores the tokens of occupied sections. It is written by the producer to release a section with freshly written data and read by the consumer to acquire access to a previously written section.

Memory cores. The memory module contains multiple dual-port memory cores. The number of memory cores is determined by the user memory partition factors of all buffer dimensions. The width of each memory core is determined by the user data type of a single buffer element. The number of sections in each memory core is determined by the user ping-pong factor. The depth of each memory core is determined by the user ping-pong factor, the size of each buffer dimension and the partition factor. In addition, a user can configure the memory core type to be either BRAM or URAM. For example, if a user requests a 2D buffer channel of type `float[10][4]` with two ping-pong sections and completely partitions the second dimension, we need to generate four dual-port memory cores, each memory core with a width of 32 bits and a depth of 20.

Buffer channel generator. Depending on multiple user configurations, the memory module configuration becomes complex and variable. Unlike the FIFOs that can be simply instantiated from one RTL module, we implement a buffer channel generator to automatically compose a different RTL module for each buffer channel based on the user configuration.

Note that the only resource overhead of our proposed buffer design is due to the FIFOs which is neglible. For example, when the number of sections is two, each FIFO consumes less than 30 LUTs and 50 FFs.

V. Frontend Programming Interface

The buffer channel presented in Section IV is quite complex to use by HLS programmers. First, the programmer needs to manually configure the memory cores' parameters as discussed in Section IV-B. Second, to use the buffer channel in the producer and consumer tasks, the programmer needs to manually write the code to read a token from one FIFO, access the corresponding section from the memory core as dictated by the token, and then push the token back to the other FIFO. Third, since there is no data dependency between the memory read and write operations which are sandwiched between the read from one FIFO and the write to the other FIFO, HLS compilers will often schedule both the read and write together which breaks the synchronization requirements. In order to mitigate this, the user would have to create an artificial dependency every time they want to access a tile of data from the buffer channel, which is cumbersome to deal with.

To make our buffer channel design friendly for programmers, we implement a C++ template based interface to declare and configure a buffer channel, and a C++ *Resource Acquisition Is Initialization (RAII)* idiom based programming model to use the buffer channel in the producer and consumer tasks and transparently take care of the token exchange, buffer access and artificial dependency creation. Finally, our frontend tool automatically parses the user task-parallel HLS program, extracts the task graph with the task communication metadata, and transforms the producer and consumer task code to be synthesized by Vitis HLS.

```
1  void top(..) {
2    buffer<
3      float[2][4],      // type and size decl, can be multi-D
4      2,                // number of ping-pong sections
5      array_partition< // partitioning scheme for each
6        normal,         // dim, can be normal, complete,
7        cyclic<2> >,    // block<factor>, cyclic<factor>
8      memcore<BRAM>     // can be URAM or BRAM
9    > buf;
10   task()
11     .invoke(producer, buf, ...)
12     .invoke(consumer, buf, ...)
13 }
```

Listing 1. Buffer channel declaration example

A. Buffer Channel Declaration

Our buffer channel is highly configurable. Listing 1 shows an example declaration of a buffer channel, where a user can customize the following configurations using a C++ template.

1. **Type**: This is the type declaration of the buffer in the standard C++ syntax. Multiple dimensions are supported. For example, `float[2][4]` in Listing 1 is a 2 x 4 array of floats. The data type of a single array element (e.g., `float`) translates to the width of each memory core. The number of dimensions and their dimension sizes, combined with the following number of ping-pong sections and partitioning schemes, determine the depth of each memory core.
2. **Sections**: This is the number of ping-pong sections. The value of one means a regular buffer, two means a double buffer, three means a triple buffer, and so on.
3. **Array partitioning**: This specifies the partitioning scheme to be applied to the buffer. It is specified by writing `array_partition` followed by a list of template arguments for each dimension of the buffer. The possible arguments can be `normal` for no partitioning, `complete` for full partitioning, `cyclic` for cyclic partitioning, and `block` for block partitioning. For `cyclic` and `block`, a partition factor can be passed as a template argument. These types refer to exactly what they refer to in Vitis HLS. Internally, we use this information along with the dimension sizes to determine how many cores to instantiate and their sizes.
4. **Memory core type**: We support both BRAM and URAM core types. This is specified by writing `memcore` and passing `BRAM` or `URAM` as a template argument to it. Internally, we use fixed templates for BRAM and URAM based memory cores to make Vivado infer the right type.

Coming back to Listing 1, our frontend tool will parse the code and get the metadata: it declares a buffer of type `float`, which is two dimensional with sizes 2 and 4, has 2 sections, is partitioned only on the second dimension cyclically by a factor of 2, and uses `BRAM` as the underlying memory core.

B. Vector Addition Example and Buffer Channel Usage

To illustrate the buffer channel usage, we take vector addition as an example, whose task graph is shown in Fig. 6. It loads two vectors of floating points from the off-chip memory as input, adds them, and writes the output to the off-chip memory. Listing 2 shows the vector addition program written

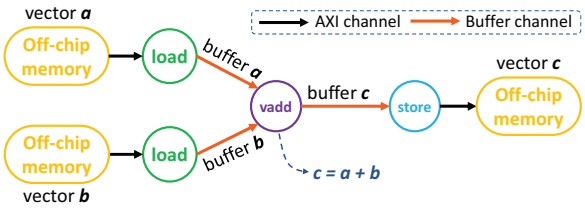

Fig. 6. Task graph of vector addition example.

using PASTA's programming model. The off-chip memory types (denoted by `mmap` in PASTA, lines 21-22) are widened to utilize the off-chip memory bandwidth effectively.

The top level function (lines 21-30) declares all the three buffer channels (lines 23-25), one to accommodate a tile from input vector `a`, another to accommodate a tile from input vector `b` and lastly one to store a tile of the output vector `c`. The top level function shows all the task invocations (lines 26-29). Naturally, the `load` tasks need access to off-chip memory from where they copy data in the form of tiles to buffer based channels. The *vadd* task (lines 5-20) receives these input tiles of data via `buf_in_a` and `buf_in_b` buffer channels, performs addition via unrolled loops and stores the data of resultant tiles in buffer channel `buf_out`. The `store` task takes that as input and writes the data to off-chip memory.

Buffer channel usage. To make it easy to use our buffer channel in the producer and consumer tasks, we use the C++ RAII idiom to transparently take care of the cumbersome token exchange, buffer access and artificial dependency creation as discussed earlier. As shown in listing 2, the `vadd` task (lines 5-20) acts as a consumer task for the buffers `buf_in_a` and `buf_in_b`, and acts a producer task for the buffer `buf_out`. In general, the producer and consumer tasks take the buffer as a reference argument of type `obuffer` and `ibuffer` (lines 5-7), respectively, so that the tool can distinguish between them. Note that the `buffer` template class inherits from both `ibuffer` and `obuffer` template classes, which have a different implementation of the `acquire` member function.

To use a buffer, its `acquire()` member function (line 10) can be called, which blocks on the source FIFO (i.e., the free sections FIFO for a producer task and the occupied sections FIFO for a consumer task) until a token can be retrieved from it. Once a token is received, a new object of type `section_t` is instantiated that has a reference to the corresponding section. The user can call the `operator()` on the `section_t` object to get this reference (line 12). This reference can just be used as a regular array of the same type and dimensions.

Once the returned `section_t` object goes out of scope, the token is written back to the sink FIFO (i.e., the occupied sections FIFO for a producer task and the free sections FIFO for a consumer task) by the `section_t` object's destructor function, marking the release of the buffer. As the communication with memory cores and writing the token to the sink FIFO do not have any data dependency, vendor HLS will often do the FIFO write before the memory communication. To prevent this, we introduce an artificial dependency in the implementation of `section_t` to ensure the ordering of these operations.

```
1  // template arguments of ibuffer and obuffer are identical
2  // to the declarations in the `top` task, pruning to save space
3  void load(mmap<WIDE_TYPE> mem, obuffer<...> &buf, ...) { ... }
4  void store(mmap<WIDE_TYPE> mem, ibuffer<...> &buf, ...) { ... }
5  void vadd(ibuffer<float[LEN], N_SECTIONS, ...>& buf_in_a,
6            ibuffer<float[LEN], N_SECTIONS, ...>& buf_in_b,
7            obuffer<float[LEN], N_SECTIONS, ...>& buf_out,
8            int n_tiles ) {
9    for (int i = 0; i < n_tiles; i++) {
10     auto section_in_a = buf_in_a.acquire(); // buffer acquired
11     // buf_in_a_ref references to the actual buffer section
12     auto &buf_in_a_ref = section_in_a();
13     // omitted similar code to get buf_in_b_ref and buf_out_ref
14     for (int j = 0; j < LEN; j++) {
15       #pragma HLS unroll factor=FACTOR
16       buf_out_ref[j] = buf_in_a_ref[j] + buf_in_b_ref[j];
17     }
18     // buffers automatically released when they are out of scope
19   }
20 }
21 void top(mmap<WIDE_TYPE> vec_a, mmap<WIDE_TYPE> vec_b,
22          mmap<WIDE_TYPE> vec_c, int n_tiles) {
23   buffer<float[LEN], N_SECTIONS,
24     array_partition<cyclic<FACTOR>>,
25     memcore<BRAM>> buf_a, buf_b, buf_c;
26     task().invoke(load, vec_a, buf_a, n_tiles)
27           .invoke(load, vec_b, buf_b, n_tiles)
28           .invoke(vadd, buf_a, buf_b, buf_c, n_tiles)
29           .invoke(store, vec_c, buf_c, n_tiles);
30 }
```

Listing 2. Vector addition (vadd) example in PASTA

Summary. In short, to use a buffer channel in a producer or consumer task, a user only needs two lines of code (line 10 and 12): one is to call the `acquire()` member function of the buffer to return the buffer section (`section_t`) object, and the other is to call the `operator()` on the `section_t` object to get the reference to the buffer section with the actual data type. All the rest are transparently handled in PASTA. Note that in PASTA, besides the task invocation code and the buffer channel (and/or FIFO channel) related code, inside each task, the majority of the code remains the same as Vitis HLS code.

C. Buffer Channel Library Implementation

The above mentioned buffer types (template classes)—i.e., `buffer`, `ibuffer` and `obuffer`—have two implementations. One is a functional thread-safe C++ implementation that is used in the software simulation of the program for quick prototyping and correctness verification. The second is a dummy implementation that is merely used to force Vitis HLS to generate the correct interface ports for the FIFOs and the memory cores in each task's RTL module. Additionally, the dummy implementation takes care of all the transparent actions described in the previous section.

D. Parser and Program Transformation

We extend TAPA's parser [15], which is a source code analyzer and transformer based on Clang [16], to parse the buffer channel declarations and task invocations that take buffer arguments. Our parser traverses the C++ abstract syntax tree (AST) to analyze the buffer declarations, get all the metadata such as the data type, number of dimensions, size of each dimension, number of ping-pong sections, memory partitioning scheme requested on each dimension, and the underlying memory core implementation requested. Then it builds the task graph with this metadata, which is later used by the high level-floorplanner (shown in Fig. 3).

Based on the metadata, our parser also transforms the source code of the producer and consumer tasks (as well as the top level function) to add appropriate interface pragmas and partitioning pragmas for the buffer channels—including their free sections FIFOs, occupied FIFOs, and memory cores—such that Vitis HLS can generate the right RTL modules for later RTL stitching with the generated buffer channel modules.

VI. Backend Frequency Optimization

To improve the timing closure and clock frequency of a task-parallel HLS program that supports both FIFO and buffer based communication channels, we extend the TAPA/AutoBridge [7], [8] backend and mainly highlight the optimizations needed to support the newly added buffer channels in PASTA (FIFO channels are also inherited and supported). It has two major steps: 1) a coarse-grained floorplanning step to place the tasks and communication channels into local FPGA slots, and 2) a pipelining step to add pipeline registers between tasks and global communication channels.

Shown in Fig. 3, based on the metadata extracted by our parser (Section V-D) and resource consumption reports of each task, the high-level floorplanner calculates the total number of wires in the buffer channel and the resource consumption of the buffer channel. It then uses this information to find a mapping of tasks to local FPGA slots so as to minimize the total number of wires crossings SLRs while keeping resource consumption of each slot within certain limits. This is done by solving an ILP program as detailed in AutoBridge [7].

After the high-level placement, global buffer channels that cross slot boundaries need to be pipelined. This is complicated as the buffer channel consists of two FIFOs and one memory module consisting of multiple memory cores. The FIFOs are placed in slots and pipelined the same way as it is done in AutoBridge [7]. The placement and pipelining of memory cores has some subtleties that need to be addressed.

A. Placement and Pipelining of Entire Buffer Channel

Assume a buffer channel that consists of a single memory core. Fig. 7 (a) and (b) show two alternatives of placing and pipelining the memory core. In Fig. 7(a), the memory core itself is mapped to be nearby the consumer task and pipeline registers are added between the producer task and the memory core. Therefore, the consumer can read/write the memory core as it normally would without extra latency. The producer task can write data as it normally would, since extra latency on the paths will not affect write operations. However, if the producer needs to read some data (as we observed in applications like KMeans), it suffers from an additional latency that is twice the number of registers added on the path (both for the address and data signals). Since read operations are sensitive to the access latency, the producer task code would need to be updated and recompiled with this additional latency taken into account; otherwise, it may read incorrect data at an earlier clock cycle. Another alternative is to map the memory core nearby the

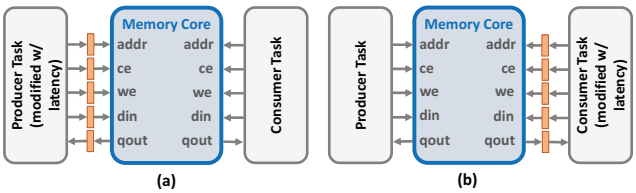

Fig. 7. Two alternatives of placing and pipelining memory cores: (a) placing on consumer task side and pipelining producer access, (b) placing on producer task side and pipelining consumer access.

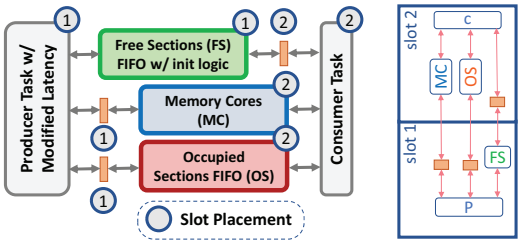

Fig. 8. Placement and pipelining of global buffer channel components.

producer task and add pipeline registers between the consumer task and the memory core, shown in Fig. 7(b). Similarly, the consumer task code would need to be updated and recompiled with the additional read latency taken into account.

Since most often, a producer task writes the buffer channel and a consumer task reads the buffer channel, we decide to place the memory core nearby the consumer and add pipeline registers on the producer side. As a result, besides rare cases where a producer also reads from the buffer channel, neither the producer nor the consumer suffers from any additional latency. To accommodate for the rare producer reads, we modify the producer code and recompile it again with the added latency by inserting a 'latency' parameter in the interface pragma, which is indicated by the dashed arrow from the high-level floorplanner to the CPP source files in Fig. 3.

Fig. 8 shows how the whole buffer channel is placed and pipelined. The free sections FIFO is constrained nearby the producer task that consumes this FIFO, while the occupied sections FIFO is constrained nearby the consumer task that consumes this FIFO. The memory cores are constrained nearby the consumer task as discussed before. The figure shows how each of the modules and pipeline registers are mapped to FPGA slots, using an example where the producer is to be placed in slot 1 and the consumer is to be placed in slot 2.

VII. Experimental Results

A. Benchmark Description and PASTA's Expressiveness

We implement four benchmarks in PASTA derived from the widely used Rodinia-HLS [9] benchmark suite to demonstrate the expressiveness and effectiveness of our framework. The high-level task graphs of our benchmarks are shown in Fig. 9. All of them have a load task to read input from off-chip memory and a store task (sometimes incorporated in the merge task) to write output data to off-chip memory. All of the benchmarks use buffer based communication between tasks: all the buffers are ping-pong based and have memory partitioning. The motivational pathfinder benchmark in Section II-B uses both buffer and FIFO based communication channels.

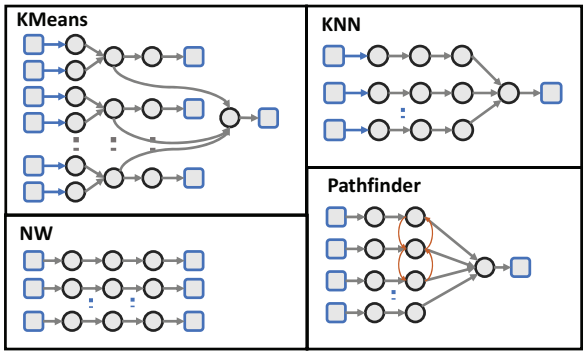

Fig. 9. High-level task graph of our benchmarks. Circles: tasks, thick arrows: buffer channels, thin arrows: FIFO channels, blue squares: HBM banks.

KNN. The k-nearest neighbors (KNN) benchmark takes a 2D query point and a set of search space points divided between different memory banks, and computes the K (in our implementation $K = 8$) nearest neighbors to the query point. Each PE processes its share of the search space and writes partial top K candidates to the merge task which picks the top K global candidates and writes it back to off-chip memory.

KMeans. The KMeans clustering benchmark takes a search space of 2D points, K clusters ($K = 8$ in our case), initial memberships of points to clusters as inputs, and performs multiple iterations. In each iteration, it finds new memberships of points to current clusters and recomputes the clusters based on that new membership. The new membership and new clusters are written back to off-chip memory, from where they are loaded again in the next iteration. Each PE of the benchmark reads from two memory banks, one for the points and the other for the memberships which it also writes back to. Once a PE is done processing all its data, its partial new clusters are sent to the merge task, which calculates the global new clusters and writes them to off-chip memory.

NW. The Needleman-Wunsch (NW) algorithm performs DNA sequencing on short reads of length 128. The algorithm is based on dynamic programming and works by filling a 2D grid of 128×128. We break this grid into 16 parts and process the diagonals in parallel. We create multiple PEs, each PE capable of handling 4 short read alignment jobs at a time. Finally, the aligned reads are written to the off-chip memory.

B. Experimental Setup

We implement our PASTA framework on top of TAPA/AutoBridge [7], [8] and test it using the aforementioned benchmarks. All of these benchmarks are scalable, in the sense that we can use more memory banks and replicate more PEs to process more data in parallel (Fig. 9). For each benchmark, we create multiple configurations by sweeping from 2 PEs to 12 PEs. For each configuration, we write the baseline purely in Vitis HLS C++ programming style and perform synthesis, place and route via standard Vitis flow. We compare it with a task-parallel version written using PASTA's programming model, which is optimized for timing closure by PASTA. For both versions, the target frequency is set to 300 MHz. All experiments are performed on the AMD/Xilinx HBM-based