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

MOAZIN KHATTI, XINGYU TIAN, AHMAD SEDIGH BAROUGHI, and AKHIL RAJ BARANWAL, School of Engineering Science, Simon Fraser University, Canada

YUZE CHI, LICHENG GUO, and JASON CONG, Computer Science Department, University of California, Los Angeles, United States

ZHENMAN FANG, School of Engineering Science, Simon Fraser University, Canada

In recent years, the adoption of FPGAs in datacenters has increased, with a growing number of users choosing High-Level Synthesis (HLS) as their preferred programming method. While HLS simplifies FPGA programming, one notable challenge arises when scaling up designs for modern datacenter FPGAs that comprise multiple dies. The extra delays introduced due to die crossings and routing congestion can significantly degrade the frequency of large designs on these FPGA boards. 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. Recent efforts have aimed to address these issues by employing coarse-grained floorplanning and pipelining strategies on task-parallel HLS designs where multiple tasks run concurrently and communicate through FIFO stream channels. However, many applications are not streaming friendly and many existing accelerator designs heavily rely on buffer channel based communication between tasks.

In this work, we take a step further to support a task-parallel programming model where tasks can communicate via both FIFO stream channels and buffer channels. 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. Our framework introduces a latency-insensitive buffer channel design, which supports memory partitioning and ping-pong buffering while remaining compatible with vendor HLS tools. On the frontend, we provide an easy-to-use programming model for utilizing the proposed buffer channel; while on the backend, we implement efficient placement and pipelining strategies for the proposed buffer channel. To validate the effectiveness of our framework, we test it on four widely used Rodinia HLS benchmarks and two real-world accelerator designs and show an average frequency improvement of 25%, with peak improvements of up to 89% on AMD/Xilinx Alveo U280 boards compared to Vitis HLS baselines.

CCS Concepts: • **Hardware → Hardware accelerators**; **Hardware-software codesign**; **Partitioning and floorplanning**; • **Computer systems organization → Data flow architectures**; **High-level language architectures**; **Reconfigurable computing**.

Additional Key Words and Phrases: Multi-die FPGA, high-level synthesis, task-parallel programming, buffer channel, hardware acceleration, frequency optimization, coarse-grained floorplanning

Authors' addresses: Moazin Khatti, moazin_khatti@sfu.ca; Xingyu Tian, xingyu_tian@sfu.ca; Ahmad Sedigh Baroughi, ahmad_sedigh_baroughi@sfu.ca; Akhil Raj Baranwal, akhil_baranwal@sfu.ca, School of Engineering Science, Simon Fraser University, 8888 University Dr, Burnaby, BC, Canada, V5A1S6; Yuze Chi, chiyuze@cs.ucla.edu; Licheng Guo, lcguo@ucla.edu; Jason Cong, cong@cs.ucla.edu, Computer Science Department, University of California, Los Angeles, 404 Westwood Plaza, Los Angeles, California, United States, 90095; Zhenman Fang, zhenman@sfu.ca, School of Engineering Science, Simon Fraser University, 8888 University Dr, Burnaby, BC, Canada, V5A1S6.

ACM Reference Format:
Moazin Khatti, Xingyu Tian, Ahmad Sedigh Baroughi, Akhil Raj Baranwal, Yuze Chi, Licheng Guo, Jason Cong, and Zhenman Fang. 2023. PASTA: Programming and Automation Support for Scalable Task-Parallel HLS Programs on Modern Multi-Die FPGAs. *ACM Trans. Reconfig. Technol. Syst.* 1, 1, Article 1 (December 2023), 29 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Over the last decade, Field Programmable Gate Arrays (FPGAs) have become increasingly popular in the datacenter as hardware accelerators. This interest in FPGAs in the datacenter is evident by the recent acquisitions/investments of major FPGA companies, Xilinx and Altera, by major CPU companies, AMD and Intel, for $35B and $16.7B respectively [2, 13]. Furthermore, major cloud service providers such as AWS, Azure and Alibaba have all started providing instances equipped with FPGAs [3, 7, 9]. Recent works have highlighted successful FPGA-based applications and deployments in areas such as deep learning, video transcoding, graph neural networks and genome sequencing [20].

Over the last few years, we have witnessed increasing adoption of High-Level Synthesis (HLS) as an alternative means of FPGA programming, as many users at the datacenter are mainly from a software development background [20, 31]. HLS allows programmers to design FPGA applications using higher level languages such as C++ with additional hints known as pragmas. Major FPGA companies such as Intel, AMD/Xilinx and Microchip have all developed their own HLS tools to facilitate easier programming of their FPGAs [8, 10, 11]. This increasing interest in HLS is also evident by the vast landscape of research in academia around HLS tools and HLS based accelerator designs [20, 31].

While HLS lifts up the abstraction level of accelerator design, it has some limitations. One major problem is that HLS is a black-box like process that takes the input HLS C/C++ program and compiles it to RTL (Register-Transfer Level) without giving the programmer finer control of how different components are being implemented in hardware. Hardware design, and specifically FPGA design, is a very complex process and problems such as timing closure issues can come up at later stages of the physical design flow. While hardware designers have complete fine-grained control of their design and its layout and can make the necessary fixes to solve these issues, HLS users only have the HLS C/C++ code under their control, making it very difficult for them to digest these problems, let alone solving them.

A particular type of such timing closure issues arise in modern datacenter FPGAs. In order to house a plethora of resources, modern FPGAs comprise of multiple dies placed on a shared package substrate. These dies are connected to each other via silicon interposers. In AMD/Xilinx's Stacked Silicon Interconnect (SSI) technology, dies are referred to as Super Logic Regions (SLRs) and the die crossings are known as SLR crossings [4]. The existence of these silicon interposers allow a user's design to spread across multiple dies while parts in different dies can still communicate via the interposers. However, any nets in the design that go via SLR crossings suffer additional non-trivial delays, which can be as much as eight times the delay of a medium length wire on the same die [23]. Furthermore, these FPGAs also consist of large IPs such as DDR and PCIe which consume a lot of resources near the IO banks and may cause nearby signals to take longer routes with larger delays [1].

Hardware designers, who work with a typical RTL-based flow, have a fine-grained control of their design as well as its layout planning, and therefore are capable of solving these problems [4]. However, this is quite challenging for HLS tools and their users. Current HLS tools use pre-characterized approximate delay models for different components of the design [15, 21]. While these models are reasonably sufficient for logic delays, they are highly inaccurate for net delays, as net delays are heavily dependent on the placement and routing which becomes known only late in the flow. By the time the backend tool is able to model the delays accurately, it is already too late for it to fix these problems as the design's cycle-accurate semantics are already decided by the HLS tool and cannot be changed.

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. A recent work, TAPA/Autobridge [24] takes a great initial step in this direction. Autobridge [25], combined with its programming model of TAPA [17], requires programmers to develop their accelerator as a task-parallel HLS design where tasks can run in a coarse-grained parallel and/or pipelined fashion and tasks communicate via latency-insensitive FIFO channels. The tool then extracts the task-parallel graph with metadata, and performs 1) coarse-grained floorplanning to locally place and route each task (node) onto a smaller FPGA slot for better local timing closure, and 2) pipelining FIFO channels (edges) between tasks that go across slot boundaries to ensure that they do not become global timing bottlenecks.

While TAPA/Autobridge [24] shows excellent frequency improvements, the tool limits programmers to a strict programming model which only supports parallel tasks that communicate via FIFO based stream channels. However, many applications are not purely streaming and many existing accelerators heavily rely on ping-pong buffer channels [18, 19, 34, 45]. Similarly, many existing automatic design-space exploration tools rely on the widely used load-compute-store computation pattern where tasks communicate via ping-pong buffers [22, 39].

In this work, we present our framework PASTA, which provides programming and automation support for a broader range of task-parallel HLS designs where tasks can communicate via FIFO stream channels as well as latency-insensitive buffer channels. We build our framework by extending the TAPA/Autobridge [24] framework.

Supporting the additional buffer channels introduces additional challenges: 1) This buffer channel needs to enable separate tasks with separate RTL modules containing separate FSMs (Finite State Machines) to communicate with each other in a dataflow fashion. 2) The buffer channel needs to support efficient implementation of multi-dimensional arrays with different partitioning schemes such as cyclic, block and complete as these are heavily utilized by HLS designs for optimizations such as loop pipelining and unrolling in the producer and consumer tasks. 3) The buffer channel needs to have an IO interface with its producer and consumer tasks, which is compatible with and can be implemented in vendor HLS tools such as Vitis HLS. 4) The above requirements may make the channel design cumbersome to use and an easy-to-use programming interface needs to be designed for programmers to easily incorporate large number of buffer channels in their designs. 5) The buffer channel's implementation needs to be latency-insensitive such that long paths can be pipelined, which is much more challenging than pipelining the FIFO channels.

To address the above challenges, we present a buffer channel abstraction that is latency-insensitive, decoupled from producer and consumer tasks, supports ping-pong buffering and memory partitioning, and is compatible with existing vendor HLS tools. Our proposed design is composed of multiple memory cores and two FIFOs. Multiple dual-port memory cores allow ping-pong enabled partitioned logical memories for the producer/consumer tasks while the two FIFOs, namely free and occupied sections FIFOs, house access tokens that synchronize the communication between the producer and consumer tasks. To efficiently utilize on-chip memories, we analyze the buffer usage of the producer and consumer tasks and automatically select between Simple Dual-Port (S2P) and True Dual-Port (T2P) memories. As the IO interface in our buffer channel is limited to `ap_fifo` and `ap_memory` ports, it is supported by existing commercial HLS compilers such as Vitis HLS. To support arbitrary buffer channel configurations, we implement a buffer channel generator to automatically generate the custom buffer channel designs for all the required unique buffer configurations.

To make the buffer channel usable by users in the frontend, we design easy-to-use APIs for buffer channel declaration and usage. For buffer channel configuration, the APIs use C++ template types to specify the width, shape, depth, number of sections, partitioning schemes and memory resource types. For the usage, the APIs utilize the C++ *Resource Acquisition Is Initialization (RAII)* idiom to automatically take care of cumbersome token exchanges. Since, it is very common in HLS designs to have a large number of buffer channels of identical configuration, we also implement syntactic macros to make instantiation and usage of such arrays of buffer channels easy for the programmers. We implement these APIs in two different contexts. The first is a pure C++ thread-safe software implementation that integrates in the multi-threaded simulator of the task-parallel HLS programming framework TAPA [17]. This implementation enables the programmers to quickly prototype and verify the functional correctness of their designs when using the buffer channel. The second is an HLS implementation that facilitates the generation of correct interface ports and helps take care of the cumbersome token exchanges and a special dependency problem that comes up in the design. We also extend TAPA's parser utility (i.e., compiler) to add appropriate directives to the code, making sure that ports for the buffer channel are correctly generated and used in the HLS generated design.

To support frequency optimizations in the backend, we devise a method to place the buffer channel via coarse-grained floorplanning and pipeline paths that can become timing bottlenecks. To accomplish this, we place the free sections FIFO with the producer task and occupied sections FIFO near the consumer task, and then pipeline them from the opposite sides. The memory cores are placed near the consumer task and pipelined on the producer side. We pipeline on the producer side as read operations on pipelined memory channels have longer latency and producer tasks are less likely to read. To ensure functional correctness, we selectively recompile producer tasks that read from memory channels and have been pipelined to ensure that they remain functionally correct. We further present some analysis on the effects of pipelining the buffer channel and the latency overhead it may introduce. Our analysis shows that when the producer task also reads the buffer channel, there can be rare cases where pipelining the buffer channel may add large latency overhead. To prevent this, by default, PASTA maps the producer and consumer tasks to the same FPGA slot if the producer task reads; meanwhile, it still gives users an option to separate them to different slots if they want to.

We evaluate our PASTA tool on four widely used Rodinia HLS [19] benchmarks and two real-world accelerators. Experimental results show that PASTA achieves an average of 25% (up to 89%) frequency improvement for these designs on AMD/Xilinx HBM-based Alveo U280 FPGA board, compared to Vitis baselines. Moreover, we confirm with on-board execution tests that our improvements in frequency indeed translate to a similar execution time speedup.

In summary, the main contributions of this work are:

1). Analysis of challenges and design of a latency-insensitive buffer channel abstraction to support scalable task-parallel HLS programs where tasks communicate via buffer channels.
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. This includes the easy-to-use buffer channel APIs, front-end library and compiler implementations and backend frequency optimizations. PASTA is open sourced here: https://github.com/sfu-HiAccel/pasta.
3). Experimental results to demonstrate superior frequency and performance improvements using PASTA.

This journal submission further presents the following new contributions on top of our FCCM 2023 publication [30].

1). We extend the tool flow to automatically decide between True Dual-Port (T2P) and Simple Dual-Port (S2P) memories depending on the user's needs, thereby decreasing the resource consumption when possible while still giving users the flexibility to use T2P memories if they need it. (Section 3.2.2 and 5.1)
2). We design and implement additional 'using short_buffer_t' and 'buffers' APIs to make the buffer channel more usable by allowing easier instantiation and usage of a large number of buffer channels. (Section 4.3 and 4.4)
3). We also describe in more detail how the frontend parser and library implementation work in PASTA. (Section 4.4)

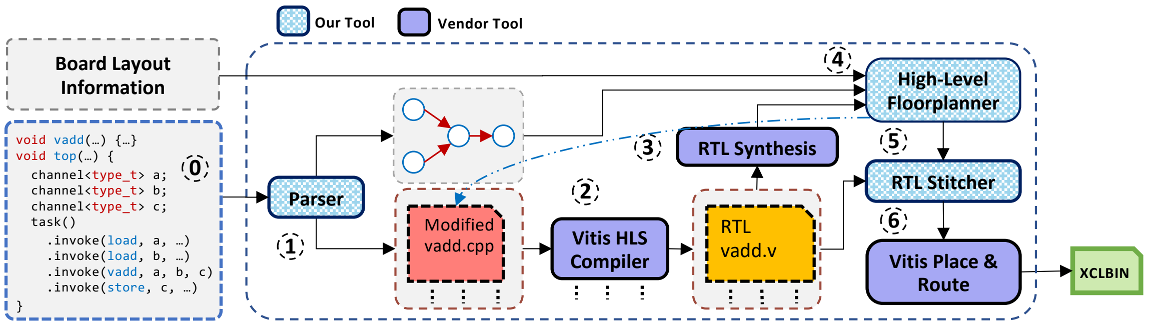
4). In the backend we present more analysis on the effects of pipelining the buffer channel. To avoid the effects of harmful latency increase due to the pipelining of buffer channels in certain rare cases, we map them to the same slot by default while still separating them if the user desires. (Section 5.2)

5). Lastly, we test our framework on two additional real-world accelerator designs: a highly optimized SpMV accelerator that uses both the buffer and FIFO channels [38] and a Minimap genome sequencing accelerator that uses buffer channels [27]. (Across Section 6)

Section 2 gives an overview of our overall toolflow. **Section 3** goes into the design and implementation of the proposed buffer channel abstraction. **Section 4** presents the design and implementation of the frontend programming model for the buffer channel. **Section 5** describes how the frequency optimization techniques for the buffer channel were implemented in the backend. **Section 6** presents the experimental results. **Section 7** discusses related work and **Section 8** concludes the paper.

2 PASTA TOOLFLOW OVERVIEW

This section describes the overall toolflow of our PASTA framework as shown in Figure 1. The contributions in each part of the framework will be explained in detail in the following sections.



```
void vadd(…) {…}
void top(…) {
  channel<type_t> a;
  channel<type_t> b;
  channel<type_t> c;
  task()
    .invoke(load, a, …)
    .invoke(load, b, …)
    .invoke(vadd, a, b, c)
    .invoke(store, c, …)
}
```

Fig. 1. An overview of our PASTA toolflow

2.1 Programming Model

At step 0, the user writes a design using a task-parallel programming model with PASTA APIs which are based on TAPA [17]. The programming model essentially consists of a directed graph, where vertices are tasks (written in Vitis HLS) and directed edges are uni-directional channels that connect tasks and allow them to communicate. These channels transmit data from their producer tasks to the consumer tasks (hence uni-directional) and can either be FIFO `stream` channels or ping-pong `buffer` channels. For off-chip memory communication, `MAXI` channels are allowed between off-chip memory banks and user written tasks. Lastly, constant scalar values are allowed to be propagated from host side to one or more tasks. Figure 2 shows a task graph for a simple vector addition example.

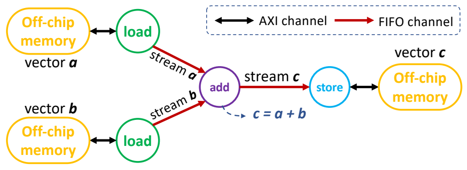


Fig. 2. Task graph for vector addition

```cpp
void load(mmap<int32_t> memory, ostream<int32_t>& outStream, int n_pts) {
    for (int i = 0; i < n_pts; i++)
        outStream.write(memory[i]);
}
void add(istream<int32_t>& in_stream_a, istream<int32_t>& in_stream_b, ostream<int32_t>& out_stream, int n_pts) {
    for (int i = 0; i < n_pts; i++)
        out_stream.write(in_stream_a.read() + in_stream_b.read());
}
void store(mmap<int32_t> memory, istream<int32_t>& inStream, int n_pts) {
    for (int i = 0; i < n_pts; i++)
        memory[i] = inStream.read();
}
void VecAdd(mmap<int32_t> memory_a, mmap<int32_t> memory_b, mmap<int32_t> memory_c, int n_pts) {
    stream<int32_t> load_a_stream, load_b_stream, store_c_stream;
    task()
      .invoke(load, memory_a, load_a_stream, n_pts)
      .invoke(load, memory_b, load_b_stream, n_pts)
      .invoke(add, load_a_stream, load_b_stream, load_c_stream, n_pts)
      .invoke(store, memory_c, load_c_stream, n_pts);
}
```

Listing 1. Vector addition example in PASTA

The task graph is represented in C++ files using some special APIs and syntax. The C++ program corresponding to Figure 2 is shown in Listing 1. Each of the `load`, `add` and `store` tasks are to be instantiated from C++ functions which contain Vitis HLS code for those tasks. The channels that connect to these tasks are represented as special arguments in these functions. For example, `ostream` means the task connects to a FIFO stream acting as a producer, while `istream` means the task connects to a FIFO stream acting as a consumer. Similarly, `ibuffer` or `obuffer` signify the task as a consumer or producer of a ping-pong buffer channel respectively. Additionally, off-chip memory channels are represented as `mmap` arguments while constant scalar values coming from host are simple pass-by-value arguments.

While the task definitions are represented as individual C++ functions, the tasks themselves, the edges connecting them and their connectivity with each other is defined in a top-level function. The top function is named `VecAdd` in the example shown in Listing 1. Connections to off-chip memory banks and scalar values from host side are both represented as arguments to the top-level function from where it can be passed to any number of instantiated tasks. All channels are instantiated in the body of top-level function by instantiating variables of the corresponding type (e.g. `stream` in case of FIFO streams). Tasks are instantiated by calling the `invoke` method of the `task()` object by passing the task name, channel variable declarations and scalar arguments.

PASTA, which is based on the TAPA/Autobridge [24] framework, allows a programmer to create a single design consisting of a task-parallel kernel design and a host code written using our APIs and library. This single design can be used to do software simulation, HLS compilation and hardware generation, hardware simulation as well as a hardware run on an actual board, which is a big time saver for the programmers.

2.2 Internal Working of PASTA

2.2.1 *Parser.*

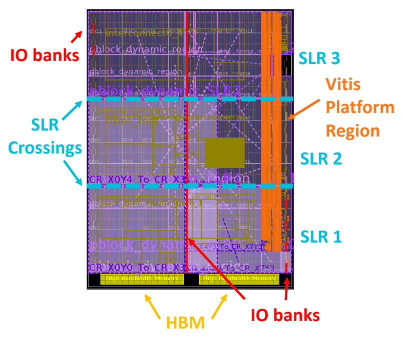
The parser is a utility built using the widely used `LLVM/clang` compiler [6, 32]. It is essentially a recursive *Abstract Syntax Tree (AST)* visitor that walks through the whole C++ file and does two important things. Firstly, it extracts the HLS source code for each of the tasks and transforms it. Secondly, as it traverses the C++ file, it builds an internal representation of the task-graph. This utility is described in detail in Subsubsection 4.4.3.

2.2.2 *HLS Synthesis.*

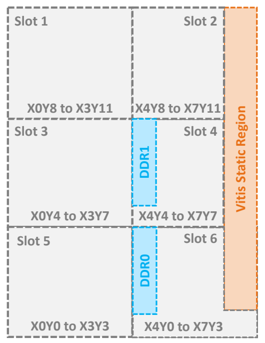
These files are individually compiled to RTL by invoking the Vitis HLS compiler in parallel on all of them. The compiler also generates log files and reports capturing the estimated resource usage of each of the tasks along with latency and timing information of various components of the design.

2.2.3 *RTL Synthesis.*

Optionally, the tool can further synthesize the generated RTL to netlist by invoking `Vivado` to get very accurate resource consumption reports, which can be very helpful for the later floorplanning stages of the flow.



(a) Device view of Alveo U280



(b) Slots modeling of Alveo U280

Fig. 3. Device view and slot modeling for Alveo U280.

2.2.4 High-Level Floorplanner. Once resource consumption reports are generated after HLS and RTL synthesis, the high-level floorplanner tool is invoked. This tool takes the resource consumption reports, the task graph as well as the target board information and performs coarse-grained floorplanning. Initially, an internal representation of the task-graph is created with all the tasks, their resource consumption, off-chip memory banks used and pre-existing floorplanning constraints. While the resource consumption of all the channels (FIFO streams and ping-pong buffers) can be measured by invoking Vivado to synthesize them, we choose to calculate it using internal resource models to save floorplanning time. In order to perform floorplanning, the tool models the device's fabric as multiple local slots. Consider the AMD/Xilinx Alveo U280 board shown in Figure 3, it consists of three Super Logic Regions (SLRs) and a series of IO banks in the middle. This board's programmable fabric is defined as consisting of six slots as defined by the SLR crossings and IO banks. The tool accurately models the programmable resources available in each slot as well as the routing resources available for SLR crossings. To model the resource budget of each slot, the tool appropriately subtracts areas reserved for the Vitis Static Region or the DRAM IPs.

Based on this modeling, the tool formulates an *Integer Linear Programming (ILP)* problem that attempts to map each of the tasks to slots while keeping area consumption per slot within certain limits and minimizing the number of wires crossing slot boundaries. The floorplanner tool solves the ILP problem by using an ILP solver and ultimately produces a mapping from tasks to slots. Additionally, it formulates another ILP problem to determine the routes for every channel that crosses slot boundaries based on the available routing resources and the cost incurred by each routing. This is done to ensure that ultimately the routing is realizable by Vivado as well as to avoid making channels go through slots that have a high resource utilization. Based on the results of both ILP formulations, the tool emits a mapping from tasks to slots as well as the exact routes (at the granularity of slots) for the channels that cross slot boundaries. This information is used by the RTL stitcher to pipeline the channels and produce placement constraints. For details on the floorplanning ILP formulation, we refer our readers to the Autobridge [25] paper that describes the formulation thoroughly and its repository for details on the routing formulation.

2.2.5 RTL Stitching. At this stage, the tool instantiates all the RTL modules needed and stitches them together. The task RTL is taken from the HLS compilation stage. The FIFO channels are instantiated based on existing RTL files and the buffer channels are generated via the buffer channel generator component of the tool. The tool makes sure to introduce pipelining on the FIFO, buffer as well as the AXI channels as indicated by the floorplanning and routing output. Note that the tool may re-trigger HLS compilation on producer tasks that read from the buffer channel to ensure they have right semantics of the operation due to any pipelining that gets introduced during the floorplanning and pipelining process. This is indicated by the back arrow shown in Figure 1.

The stitcher finally generates a packed `xo` file that can be targeted for hardware simulation. Along with this, it generates a constraints file and a shell script which can invoke Vitis `v++` to do placement and routing. The constraints file specifies the placement constraints to inform Vivado regarding where to place each component. This includes the placement of the tasks, pipeline registers as well as channels.

2.3 New Contributions in PASTA Flow

We build our work on top of the recent works of TAPA/Autobridge [24] and present the following new contributions.

1). We present a buffer channel abstraction and present an implementation of it that is latency-insensitive, supports ping-pong buffering and memory partitioning. (Section 3)

2). In the frontend, we design and implement an easy-to-use programming model that abstracts away the otherwise cumbersome mechanisms needed to operate the buffer channel and makes it easy for users to incorporate it in their designs. (Section 4)

3). In the backend, we implement techniques to appropriately and automatically place and pipeline the buffer channels. (Section 5)

3 BUFFER CHANNEL ABSTRACTION & DESIGN

In this section, we discuss the buffer channel abstraction and design, which has to address the following challenges.

1). Since the tasks are separate modules which are separately compiled, the buffer channel needs to allow two such separate tasks—which are separate RTL modules, each with its own finite-state machine—to communicate with each other in a ping-pong manner. To enable this, an external synchronization mechanism is needed.

2). The buffer channel needs to be latency-insensitive, i.e., we should be able to arbitrarily place the producer and consumer tasks away from each other at different FPGA regions and pipeline the channel to prevent timing degradation while functional correctness is ensured. This necessitates that the buffer channel consists of signals with handshake mechanisms.

3). The buffer channel needs to support multi-dimensional arrays with different partitioning schemes so that the producer and consumer tasks can perform widely used optimizations such as *loop pipelining* and *loop unrolling*, which are crucial to HLS designs to achieve a good performance. Supporting partitioning schemes require that the buffer channel can house multiple memory cores to facilitate parallel access by the tasks.

4). The I/O interface between the tasks and the buffer channel needs to be compatible with state-of-the-art vendor HLS tools (such as Vitis HLS), so that the user can write the tasks in vendor HLS. This puts restrictions on what kind of buffer designs are acceptable for the framework.

5). The buffer design needs to provide convenient APIs that are easy to use by programmers to incorporate it in their designs developed in vendor HLS.

3.1 Buffer Channel Abstraction

To address the aforementioned challanges, we present our buffer channel abstraction and describe its overall working mechanism, as shown in Figure 4. The buffer channel consists of 1) one memory module that consists of multiple dual-port memory cores, and 2) two FIFO streams. The number of memory cores inside the memory module relates to the partitioning scheme requested by the user. Each core's memory space is divided into multiple sections, which allows

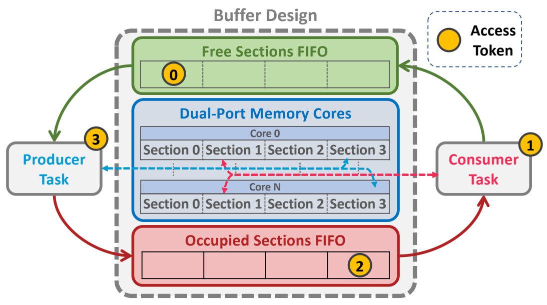


Fig. 4. Buffer channel abstraction.

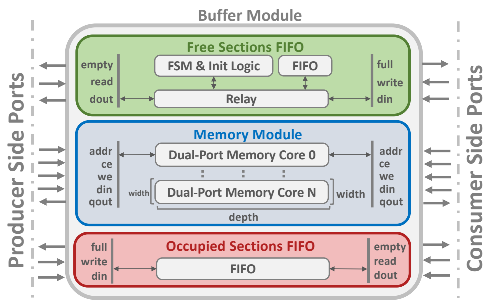


Fig. 5. Buffer channel design.

producer and consumer tasks to do ping-pong buffering: while one task writes to one section, another task may read from another section. The two FIFOs house access tokens that capture the ownership of each of the memory sections. These tokens are shown as yellow circles in Figure 4. The *free sections FIFO* represents all the sections in memory that do not contain any valid data and are free to be written to by the producer task. The *occupied sections FIFO* represents all sections in memory that contain valid data and are ready to be consumed by the consumer task. Initially, the *free sections FIFO* is initialized with all the tokens as all sections do not contain any valid data in the beginning.

3.1.1 Working Mechanism. When the producer desires to produce a chunk of data, it attempts reading a token from the *free sections FIFO* and uses that token to write the data to the corresponding memory region. Once the data is written, it writes that token read earlier to the *occupied sections FIFO* indicating that the buffer section now contains valid data.

When the consumer desires to consume a chunk of data, it attempts reading a token from the *occupied sections FIFO* and uses that token to read the data from the corresponding memory region. Once the data is read, it writes that token read earlier to the *free sections FIFO* indicating that the buffer section does not contain any valid data and can be reused by the producer. Note that both the producer and consumer tasks can do parallel access to the memory regions as the memory cores are dual-port.

As an example, consider the interaction shown in Figure 4. The *occupied sections FIFO* contains token 2 which indicates that section 2 in the memory cores contains valid data written by the producer, which the consumer has not read yet. The *free sections FIFO* contains token 0, which indicates that section 0 does not contain any valid data and is ready to be written by the producer. In the figure, the producer wants to write a new chunk of data and has read token 3 from the *free sections FIFO* recently and is now writing the data to section 3. Once it is done writing, it will write token 3 to the *occupied sections FIFO*. Similarly, the consumer wants to read a new chunk of data and has read token 1 from *occupied sections FIFO* and is therefore reading the data from section 1. Once it is done reading, it will write the token 1 to the *free sections FIFO*, for it to be acquired by the producer task on its next write.

In summary, the interaction with the buffer is a three step process: (1) reading a token from a FIFO, (2) interacting with the memory core, and finally (3) writing the token to the other FIFO. However, from the compiler's perspective, there is no true dependency between step (3) and step (2), even though our synchronization requires that step (3) happens after step (2), which can lead to functional correctness issues. We show how our programming model's implementation solves this problem by introducing an artificial dependence between step (3) and step (2) in Section 4.

3.2 Buffer Channel Design

Now we present the detailed design of the buffer channel module as shown in Figure 5.

3.2.1 Occupied Sections & Free Sections FIFO. The *occupied sections FIFO* is a standard *Shift Register Look-up Table (SRL)* based FIFO module inherited from the TAPA [17] framework. No other special changes are required. On the other hand, the *free sections FIFO* consists of an internal FIFO as well as some extra logic. This is because we need to ensure that the FIFO is initialized with all tokens at the beginning to mark that all sections are free at the beginning.

Initially, we considered doing this initialization in the consumer task when it starts execution. We could do it transparently via source-to-source transformations so that the user does not have to worry about doing this. However, if the accelerator runs multiple times, the consumer task will attempt to write twice and if these are blocking writes, the task will get blocked. We can switch to non-blocking writes but even that can fail if we transparently modify the depth of the FIFO for any reason. Altogether, we decided it would be better to do this initialization in the channel itself.

To implement this, the *free sections FIFO* module is designed to consist of three components: the actual FIFO, an internal FSM with some initialization logic, and a relay. At RTL reset, the FSM is put in the *RESET* state in which it writes to the internal FIFO and attempts to initialize it with all the valid tokens. During this process, the relay component ensures that the FIFO is unusable to outside ports. This is done by asserting the *empty* signal for the producer task side (so it does not attempt reading) and by asserting *full* signal on the consumer task side (so it does not attempt writing). Once the initialization phase completes, the FSM switches to a *DONE* state in which the internal FIFO is connected to the external producer and consumer side ports and the buffer channel module is ready to be used.

3.2.2 Memory Module. The memory module consists of multiple (logical) dual-port memory cores where each port is connected to a producer or consumer task. The width (w-bits per entry) and depth (d_p entries) of each (logical) memory core as well as the number of cores depend on the partitioning schemes, array shape and the number of sections requested by the user. Let n be the number of dimensions of the buffer, s be the number of sections needed (in each memory core), w be the bit width of the type required. Let $D = \{d_1, d_2, d_3, \dots d_n\}$ represent the size of each of the dimensions of the buffer channel. There are four possible partition schemes for each dimension, *normal* that is no partitioning, *complete* that is complete partitioning and *cyclic*/*block* which is partitioning by a certain factor. Let $cyclicf(i)$ represent the cyclic partitioning factor for dimension d_i provided it has been partitioned cyclically. Similarly, let $blockf(i)$ represent the block partitioning factor for dimension d_i provided it was indeed partitioned by the block scheme. We define the function $f(i)$ to give us the partitioning factor for each dimension i.

$$f(i) = \begin{cases} 1 & \text{scheme = normal} \\ d_i & \text{scheme = complete} \\ cyclicf(i) & \text{scheme = cyclic} \\ blockf(i) & \text{scheme = block} \end{cases} \tag{1}$$

Let $c = \prod_i f_i$ be the number of memory cores needed, $t = s \times \prod_i d_i$ be the total number of data entries (each entry is w-bits wide) stored in the buffer channel and d_p be the depth of the memory cores (all have the same depth). Then,

$$d_p = \frac{t}{c} = \frac{s \times \prod_i d_i}{\prod_i f_i} = s \times \prod_i \frac{d_i}{f_i} \tag{2}$$

These (logical) dual-port memory cores can be implemented using physical BRAM or URAM, which is dictated by a user configurable parameter. Assuming each physical (BRAM or URAM) memory bank has a bit-width of W and a depth of D_p, then each logical memory core needs $\left\lceil \frac{d_p}{D_p} \right\rceil \times \left\lceil \frac{w}{W} \right\rceil$ physical memory banks to implement. In the case of BRAM, our tool automatically chooses between Simple Dual Port (S2P) and True Dual Port (T2P). An S2P memory block has one read port and one write port and on the contrary, a T2P has two read/write ports allowing either of the

```
void TopTask(...) {
   // Channel Declarations: a FIFO stream declaration syntax (for reference) and a buffer declaration
   stream<T, N> stream_name; // where T is the type of each word of the FIFO (e.g., int, float, some_struct_t) while N is the depth of the FIFO
   buffer<T[NX][NY][NZ], // buffer entry type, followed by buffer shape
          n_sections,       // number of sections, i.e. 2 for double buffering
          array_partition<  // partitioning scheme to apply on each dimension
                          normal,    // no partitioning on 1st dim
                          cyclic<F>, // cyclic partitioning with factor F on 2nd dim
                          complete>, // complete partitioning on 3rd dim
          memcore<core_type>> // core type, bram or uram
      buffer_name;           // name of the buffer
}
```

Listing 2. Buffer declaration syntax

tasks to read or write. The advantage of using S2P is that it allows higher port width (W) and thus reduces BRAM resource consumption by 2x at higher widths of 36 and 72 bits. The tool analyzes the producer and consumer tasks to see if the producer only writes and the consumer only reads. If that is the case, S2P BRAMs are used, otherwise T2P BRAMs are used. We will discuss this in detail when discussing the resource modeling in Section 5.1. We create parameterized templates of S2P BRAM core, T2P BRAM core and T2P URAM core (URAM does not support S2P). These templates can be instantiated with any arbitrary depth and width; and the vendor synthesis tool (i.e. Vivado) takes care of implementing it correctly using resources of the corresponding type. We set a default maximum BRAM/URAM cascade height of 16 in our memory core templates. We also conducted a study on the effect of the maximum cascade height on the frequency (fmax) in Section 6.7.

3.2.3 Memory Module Generator. While a single memory core can be a simple template that can be instantiated with any type, the buffer channel must support any arbitrary memory partitioning scheme. Verilog does not allow having parameterized module interfaces, so we have to implement a custom memory module generator that can generate a memory module in Verilog, given a memory partitioning scheme, width, and depth (incorporates number of sections). At compile time, this process is optimized and only buffer modules of unique configurations are generated to save time.

4 FRONTEND PROGRAMMING MODEL

The following requirements are associated with the design of the frontend programming model. Firstly, the programming model should be easy to use for end users so that they can incorporate it in their HLS designs easily. This includes ease of declaring buffer channels of varying configurations, defining their connectivity with tasks and their transparent usage, specifically the cumbersome process of token exchanges. Secondly, the programming interface should also have mechanisms to facilitate easier handling of a large number of buffers which can be a common scenario in realistic designs. Thirdly, it should be implementable as a pure C++ code for software simulation as well as in HLS compilation; the latter includes solving the dependency problem highlighted in Section 3.1. In the next few subsections, we discuss the design of the buffer channel APIs and their implementations.

4.1 Buffer Channel Declaration

The buffer channel is added to the framework such that it can be instantiated in a similar manner to the existing FIFO stream channels except that it requires more parameters as detailed below because of its complex nature. The buffer declaration syntax is depicted in Listing 2. The FIFO stream declaration is also shown for reference.

1). **Type and Shape**: This is similar to the syntax of an array declaration in standard C++. For example, `float[20][40]` is a buffer of `float` type with two dimensions, the first one being of size 20 and the second one of size 40.

```
 1 void producer(obuffer<float[20][40], 2, array_partition<normal, cyclic>, memcore<bram>>& buf,...) {
 2     for (...) {
 3         auto section = buf.acquire(); // blocked till a token is read
 4         auto& buf_reference = section();
 5         // use buf_reference as any array of type float[2][4]
 6         for (int i = 0 ; i < 20; i++)
 7             for (int j = 0; j < 40; j++)
 8                 buf_reference[i][j] = someSourceOfValue();
 9         // buffer released as `section` goes out of scope
10     }
11 }
12 // identical to 'producer' task, acquire() gives a section object containing a memory reference that can be consumed.
13 void consumer(ibuffer<float[20][40], 2, array_partition<normal, cyclic>, memcore<bram>>& buf, ...) { ... }
14 void TopTask(...) {
15     buffer<float[20][40], 2, array_partition<normal, cyclic>, memcore<bram>> buf;
16     task().invoke(producer, buf, ...).invoke(consumer, buf, ...);
17 }
```

Listing 3. Buffer channel usage

2). **Number of Sections**: This indicates the number of sections in the buffer channel. For a single-section buffer channel (no overlap between producer and consumer possible) this should be 1, while for double-buffering it should be 2. One can also have triple buffers and quadruple buffers if desired.

3). **Partitioning Scheme**: This represents the array partitioning scheme to be applied on each of the dimensions of the buffer channel. It should be a comma separated list having as many items as the number of dimensions. Four types of partitioning schemes are supported, that is, `normal` meaning no partitioning, `complete` which refers to complete partitioning and `cyclic`/`block` which means partitioning by a given factor. These array partitioning schemes have the same meaning as they have in Vitis HLS.

4). **Memory Core**: This represents the type of memory resource to be used to implement the buffer channel. Currently available options are `uram` and `bram`.

4.2 Buffer Channel Usage

After declaring the channel it can be passed to a task invocation as an argument similar to how stream channels are passed. As shown in Listing 3, a producer task of a buffer channel needs to have an `obuffer` argument reference and similarly, a consumer task needs to have an `ibuffer` argument reference.

In order to produce a buffer, the producer code calls the `acquire()` method (line 3). This call blocks until a valid token can be read from the *free sections FIFO*. Once a valid token is read, the `acquire()` call returns with a `section` object. The `section` object represents the section in memory that the acquired token refers to and internally contains a reference to that region in memory. The producer task can then call the parenthesis operator (i.e. `()`) on the section object which returns the internal memory reference (line 4). This reference can be accessed and interacted with as any regular array of type `float[2][4]` (line 8). If partitioning was requested, parallel access can be made to this array and under the hood that will translate to concurrent accesses to the individual cores which implement that logical memory. Once the `section` object goes out of scope (line 9), it writes its token to the *occupied sections FIFO*, indicating that the producer task has written the chunk of data and that data is ready to be consumed by the consumer task. Altogether, the process of reading a token, accessing the correct section in memory as dictated by the token and ultimately writing the token back to *occupied sections FIFO* is transparently taken care of by this design.

The process is similar for the consumer task, with the exception of FIFO reads/writes. In the consumer task, the `acquire()` call reads from the *occupied sections FIFO* as it is looking for a valid section of memory to consume and when the `section` object goes out of scope, it writes the token to the *free sections FIFO*.

```
 1 // represents a buffer channel with two sections
 2 using buffer_t = buffer<float[20][40], 2, ...>;
 3 using obuffer_t = buffer<float[20][40], 2, ...>;
 4 using ibuffer_t = buffer<float[20][40], 2, ...>;
 5 // represents three buffer channels with two sections
 6 using three_buffers_t = buffers<float[20][40], 3, 2, ...>;
 7 using othree_buffers_t = obuffers<float[20][40], 3, 2, ...>;
 8 using ithree_buffers_t = ibuffers<float[20][40], 3, 2, ...>;
 9 // represents two buffer channels with two sections
10 using itwo_buffers_t = ibuffers<float[20][40], 2, 2, ...>;
11
12 void single_producer(obuffer_t& buf, ...) { ... }
13 void single_consumer(ibuffer_t& buf, ...) { ... }
14 void double_consumer(itwo_buffers_t bufs, ...) {
15     for (...) {
16         // the operator[] can be used on a `buffers` object to retrieve, the individual buffers and use them as shown below
17         auto section_0 = bufs[0].acquire(); // acquires three_buffers[0]
18         auto section_1 = bufs[1].acquire(); // acquires three_buffers[1]
19         // section_0() and section_1() returns references to arrays of type float[2][4]
20     }
21 }
22
23 void TopTask(...) {
24     three_buffers_t three_buffers;
25     task()
26        // 3 invocations of `single_producer`: 1st produces three_buffers[0], 2nd produces three_buffers[1] and 3rd produces three_buffers[2]
27       .invoke<join, 3>(single_producer, three_buffers, ...)
28       // single invocation of double_consumer that consumes three_buffers[0] and three_buffers[1]
29       .invoke(double_consumer, three_buffers, ...)
30       // single invocation of single_consumer that consumes three_buffers[2]
31       .invoke(single_consumer, three_buffers[2], ...);
32 }
```

Listing 4. ‘buffers’ type usage and ‘using’ keyword usage

4.3 Convenience APIs

The buffer types are long and very verbose as they capture the type, shape, partitioning scheme and memory core type of the channel. This can make it very cumbersome for the users to use the buffer channel. To make it worse, the users may often want to instantiate numerous buffers of identical type while scaling up their designs, which is hectic and compromises the readability. To this end, we introduce two convenience APIs.

4.3.1 Using ‘using’ *to Shorten Type Declarations.* To shorten the types, the user can simply declare short names by using the ‘using’ keyword in C++. Listing 4 shows how this can be done. At the software simulation level, this is standard C++ syntax and trivial to implement. In order to have this syntax supported in the PASTA hardware flow, we have to make sure the parser utility can peel behind the using keyword to get the actual buffer type.

4.3.2 Array of Buffers Type: ‘buffers’*.* To simplify the declaration of multiple buffers of identical types and using them, the buffers type is provided, as illustrated in Listing 4. A declaration of buffers type essentially represents multiple buffer declarations of identical buffer configuration. The individual buffers within this single declaration can be used in multiple ways. (1) The individual buffers in the buffers type can be utilized by using the *multiple invoke syntax* of the TAPA [17] programming model. It allows a programmer to have multiple invocations of the same task, each invocation taking one item out of array types such as buffers, streams or mmaps. The *multiple invoke syntax* is essentially calling the invoke function with the invoked task as the first argument and the number of invocations as the second argument, e.g., line 27 of Listing 4. (2) A buffers object can be passed down to a task that accept a buffers argument of the same length (i.e. containing the same number of individual buffer channels within it) or a smaller length and the task can further use the [] operator to get each of the buffers. This is the method used by double_consumers in Listing 4 (line 29). (3) A single buffer can be passed down to a task that consumes a single buffer by using the [] operator on the buffers object as done with single_consumer (line 31). Note that our compiler keeps count of the individual buffers that are passed down to tasks to make sure the connectivity is correctly defined.

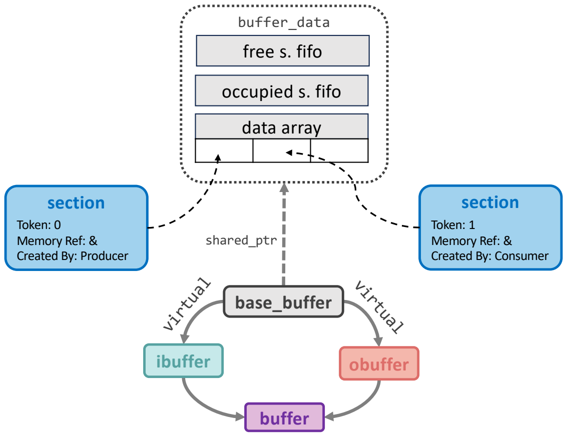


Fig. 6. Buffer channel API implementation for software simulation using a diamond inheritance pattern

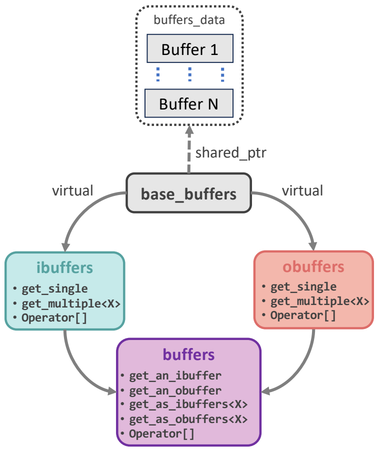


Fig. 7. Implementation of ‘buffers’ API

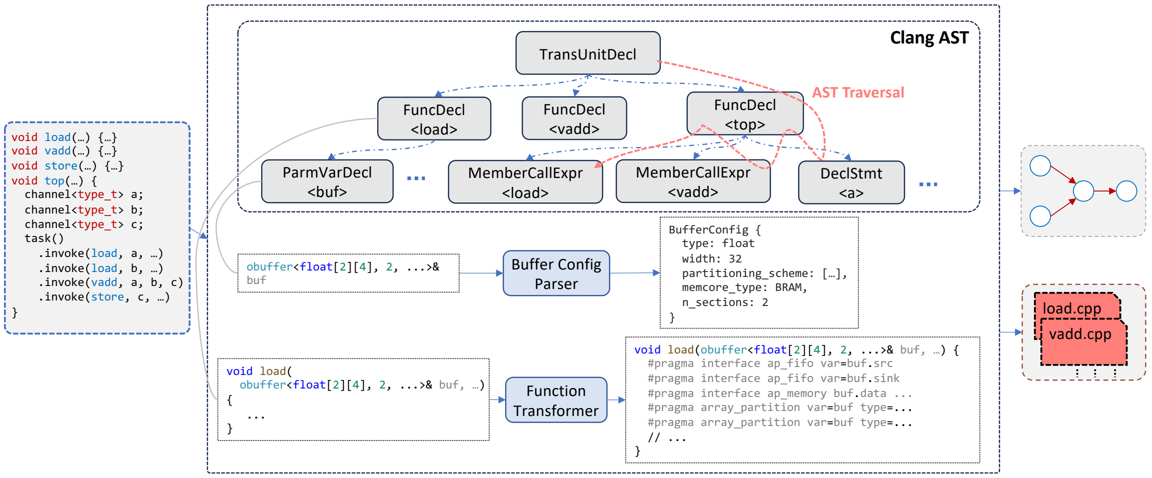
4.4 Implementations of Buffer Channel API

The proposed buffer channel APIs need to have two implementations. The first should be a pure C++, thread-safe and functional software implementation that can be used for software simulation. The second implementation should facilitate the generation of correct interface ports in each producer or consumer task as well as correct usage of the buffer channel inside the RTL modules generated by the HLS compiler.

4.4.1 Buffer Channel API Implementation for Software Simulation. Implementing the C++ version of buffer channel APIs for software simulation has the following challenges. The key problem is that the producer and consumer tasks need to differentiate on the same single buffer channel using two different types. First, when buffer channels are passed down to invoked tasks, the framework may create copies of the buffer channel objects and it needs to make sure that all object copies point to and share the same data. It also needs to ensure that ‘`ibuffer`’ and ‘`obuffer`’ references can be created from ‘`buffer`’ objects and passed down to the consumer and producer tasks, respectively. Lastly, it should ensure that ‘`ibuffer`’ and ‘`obuffer`’ type references cannot be cast to each other as it is invalid behavior.

Our functional C++ implementation is shown in Figure 6. In order to be able to pass the references as required, we use a diamond inheritance pattern. The only class that can be instantiated is the `buffer` class which can then be cast to an `ibuffer` or `obuffer` reference for consumer and producer tasks, respectively. The `ibuffer` and `obuffer` class virtually inherit from the `base_buffer` class to prevent double copies of the base class. To protect against multiple copies of the `buffer` object getting created, the FIFOs and data objects are stored inside a `buffer_data` structure which is held via a shared pointer. Even if multiple copies of the object are created, they all point to the same data. The section objects, which get created when the `acqurie()` methods are called, house references to the shared data array and the token so that the correct region (i.e., section) in memory can be accessed. The interaction is shown in detail in Figure 6.

4.4.2 Buffers Convenience API Implementation for Software Simulation. The ‘`buffers`’ (discussed in Subsection 4.3) object implementation is very complicated, especially because of the special ways in which they can be passed down to tasks. Here we give a brief overview of how the design looks like. The implementation’s inheritance diagram is shown in Figure 7. A `base_buffers` class hosts a `buffers_data` object via a shared pointer which contains all the actual buffer objects. In the diagram, it contains `N` buffers of identical type. The `buffers`, `ibuffers` and `obuffers` objects all have an `operator[]` overload that is public and can be utilized by users to pass individual `buffer` objects from `buffers` object



```
void load(…) {…}
void vadd(…) {…}
void store(…) {…}
void top(…) {
  channel<type_t> a;
  channel<type_t> b;
  channel<type_t> c;
  task()
    .invoke(load, a, …)
    .invoke(load, b, …)
    .invoke(vadd, a, b, c)
    .invoke(store, c, …)
}
```

```
obuffer<float[2][4], 2, ...>&
buf
```

```
BufferConfig {
  type: float
  width: 32
  partitioning_scheme: […],
  memcore_type: BRAM,
  n_sections: 2
}
```

```
void load(
  obuffer<float[2][4], 2, ...>& buf, …)
{
  ...
}
```

```
void load(obuffer<float[2][4], 2, ...>& buf, …) {
  #pragma interface ap_fifo var=buf.src
  #pragma interface ap_fifo var=buf.sink
  #pragma interface ap_memory buf.data ...
  #pragma array_partition var=buf type=...
  #pragma array_partition var=buf type=...
  // ...
}
```

Fig. 8. Overview of the parser utility built based on Clang/LLVM [6]

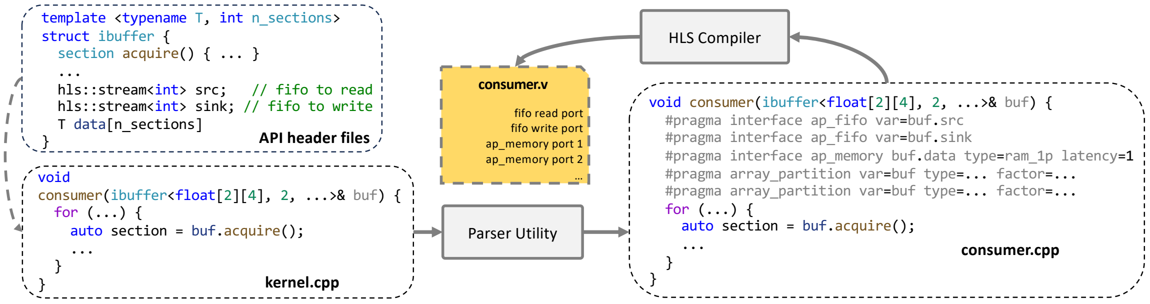
to lower level tasks, or to access and use them from producer or consumer tasks. All other methods shown in Figure 7 are not accessible by the users; they are used by a separate convenience layer that enables the different mechanisms of usage shown in Listing 4, which is discussed earlier in Section 4.3.2.

Essentially, each `ibuffers` and `obuffers` instance keeps an internal count that keeps track of how many buffers have been passed. The methods shown allow the following different patterns to get the next in sequence: 1) `ibuffer` or `obuffer` object from `buffers`, 2) `ibuffers` or `obuffers` object from `buffers`, 3) `ibuffer` object from `ibuffers`, 4) `obuffer` object from `obuffers`, 5) `ibuffers` object from `ibuffers`, and 6) `obuffers` object from `obuffers`.

4.4.3 Parser Utility for Buffer Channel HLS Implementation. An overview of the parser utility is shown in Figure 8. It ingests the FPGA side HLS C++ code written using the PASTA programming model and outputs two things: (1) A task-graph representation capturing the tasks, channels, their configuration and connectivity with each other, and (2) the transformed HLS source code for task. The parser utility is built using `Clang/LLVM` (version 8.0) [6]. The tool takes the input source code and converts it into a Clang *Abstract Syntax Tree (AST)*. A pruned and simplified view of the AST is shown in Figure 8. The dashed and dotted line indicates that the child node may not be a direct child of the parent and there may be intermediate children nodes which have been removed for simplification in the figure.

The tasks themselves appear as *FunctionDecl* nodes in the AST under the *TranslationUnitDecl* node. One such node is the *FunctionDecl* for the top task which in this example is named ‘top’. Upon being called, the parser utility is made aware of the name of the top task and hence can identify it by its name. The buffer channel declarations appear as *DeclStmt* nodes under the top level function’s node and invoke calls appear as *CXXMemberCallExpr* objects. At the heart of the parser utility is an *AST Consumer* object which uses a *Recursive AST Visitor* to traverse the AST.

The parser utility traverses the tree and identifies the top task. It then makes a list of all the tasks that have been invoked and goes through their function arguments to identify their interfaces. For example, if a task contains an `ibuffer` argument, it is supposed to be the consumer of a buffer channel. The parser utility then goes through the channel declarations in the top task to identify all the channels that have been instantiated and extracts their configurations. For the FIFO stream channel, this includes the width and depth of the FIFO. For the buffer channels, the configuration is more complicated and is extracted by a *Buffer Config Parser* object as shown in Figure 8.



```
template <typename T, int n_sections>
struct ibuffer {
  section acquire() { ... }
  ...
  hls::stream<int> src;   // fifo to read
  hls::stream<int> sink; // fifo to write
  T data[n_sections]
}
```

```
void
consumer(ibuffer<float[2][4], 2, ...>& buf) {
  for (...) {
    auto section = buf.acquire();
    ...
  }
}
```

```
void consumer(ibuffer<float[2][4], 2, ...>& buf) {
  #pragma interface ap_fifo var=buf.src
  #pragma interface ap_fifo var=buf.sink
  #pragma interface ap_memory buf.data type=ram_1p latency=1
  #pragma array_partition var=buf type=... factor=...
  #pragma array_partition var=buf type=... factor=...
  for (...) {
    auto section = buf.acquire();
    ...
  }
}
```

Fig. 9. Buffer channel API HLS implementation.

The *Buffer Config Parser* goes through the C++ template type belonging to the buffer channel and its arguments to parse the entire configuration mentioned above as shown in Figure 8. The buffer configuration includes the following:

1). The buffer's element type. For example, if the declared buffer is of type `float[10][20]`, the element type is `float`.

2). Width of the buffer's element type. If the element type is `float`, the width is 32 bits. The width one of the most important parameters used to create a buffer channel module. For simple primitive types, the utility can simply ask Clang to return the width of the type. However, this may not match with HLS's interpretation when `ap_uint<N>` (arbitrary precision integer) types are used or the type is a structure containing multiple other types. To mitigate this, the utility treats `ap_uint<N>` as a special case and parses its template argument to get the actual bit width. For a structure, the utility goes through each of its member elements and add up the widths of each of the elements.

3). The shape of the buffer. In case of `float[10][20]`, the shape is a vector containing the integers 10 and 20.

4). The number of sections (e.g., 2 for double buffering).

5). The requested memory core type of the buffer (i.e., URAM or BRAM).

6). The partitioning scheme requested on the buffer. This is a list having the same size as the number of dimensions of the buffer. Each item of the list is an enum describing the type of partitioning (i.e., cyclic, block, complete and normal) along with optionally containing a factor in case of cyclic and block partitioning.

Along with extracting the channel declarations and configurations, the parser utility also extracts information regarding how tasks are invoked and the connectivity between those invocations with the channels. Ultimately, it captures all this information in a graph-like representation and exports it for rest of the PASTA flow to use.

Finally, in addition to build the task graph, the parser also performs source-to-source code transformations on the tasks' HLS code. The goal is to add some HLS pragmas (hints) to ensure correct generation of ports for the buffer channel and make sure that HLS interacts with those ports correctly. The details of these transformations are explained in the following Section 4.4.4, which is coupled with our HLS library implementation for the buffer channel APIs.

4.4.4 *Buffer Channel API HLS Implementation.*

There are two requirements from the HLS side implementation of the buffer channel APIs. Firstly, it needs to abstract away the exact interaction of token exchange and memory access so that the users have easy-to-use APIs to work with. This also includes solving the dependency problem discussed in Section 3.1. Secondly, it needs the correct interface ports and array partitioning for buffer channels used in the producer and consumer tasks, which are automatically done by our parser utility via source-to-source code transformations.

Our HLS implementation is shown in Figure 9. In order to ensure the correct generation of interfaces, we firstly design an HLS side implementation of the `ibuffer` and `obuffer` classes, which are passed by reference to the tasks. An overview of the implementation for `ibuffer` and `obuffer` is shown in Figure 9. These types are identical and consist of two `hls::stream` objects, namely the `src` FIFO to read from and the `sink` FIFO to write to, and the memory array `data`. The types `ibuffer` and `obuffer` are identical as we can connect the `src` FIFO port to *free s. fifo* if it is the `obuffer` type and to `occupied s. fifo` if it is the `ibuffer` type at compile time. Note that these types are never instantiated as objects in C++ and are just passed by reference to the tasks; the actual buffer channels are RTL modules. However, they do contain methods which are used to implement the APIs similar to those for the software simulation discussed in Section 4.4.1 . This includes the implementation of the `section` object and the `acquire()` method.

Then our parser utility automatically performs the following code transformations specifically for the buffer channel: (1) Two `interface` pragmas are added to ensure that `ap_fifo` ports are generated for both the `src` and `sink` FIFOs. (2) One `interface` pragma is inserted to ensure that `ap_memory` ports are generated for the `data` component of the buffer. Note that only one pragma is used even if multiple ports are required due to memory partitioning. (3) Furthermore, if the buffer channel is pipelined at the later stages of the flow and the producer task reads, the `latency` component of this `ap_memory` interface pragma is modified to take into account the added latency, which will be detailed in Section 5.2. (4) Multiple `array_partition` pragmas are inserted to partition each of the dimensions of the buffer channel that the user has requested partitioning on.

Finally, note that as described in Section 3.1, vendor HLS compiler may schedule the operations incorrectly due to a lack of true dependence between the interaction with the data array and the write to the `sink` FIFO. To ensure correct ordering, we create a fake dependency by creating a **volatile boolean** variable that is always written everytime the memory core is interacted with. The write to the `sink` FIFO is conditioned upon this boolean variable, which creates a fake dependency and thereby ensures correct ordering.

5 BACKEND FREQUENCY OPTIMIZATION

An overview of the floorplanning component of our tooflow was described in Section 2.2.4: It basically takes a task graph with metadata (extracted from our parser utility, as discussed in Section 4.4) as input and then maps these tasks (nodes) to regions (i.e., slots) on the FPGA board to minimize the number of wires (edges) that cross slot boundaries while keeping the area consumption within each slot under a certain limit by solving an ILP problem [25]. This section discusses how the frequency optimizations are implemented specifically for the buffer channel. Firstly, we discuss the resource modeling for the buffer channel, which is a key input to the ILP solver (i.e., floorplanning tool). Secondly, we discuss how the buffer channel is placed and pipelined, including the potential impact of the pipelining.

5.1 Resource Modeling for Buffer Channel

To make the floorplanning functional with buffer channels, a resource model for the buffer channel needs to be implemented. The task graph extracted from our parser utility is passed down to the floorplanning utility with information such as all task node names, their resource consumption, all the edges connecting the tasks with each other and with off-chip memory as well as other other constraints. More specifically, for each buffer channel, the following information is passed: 1) producer and consumer task identifiers, 2) width and depth of the buffer channel, 3) partitioning scheme on each dimension, 4) memory core type (e.g., BRAM, URAM), 5) number of sections, and 6) a flag indicating whether Single Dual-Port (S2P) or True Dual-Port (T2P) memory cores are required.

Such information is passed to a buffer resource model which gives a resource estimate for the buffer channel, where we need to model the resource consumption of the two FIFOs and the memory module. For resource modeling of FIFOs, the resource model inside Autobridge [25] is inherited for *Shift Register Lookup Tables (SRL)* based FIFOs. For the memory module, we implement our own model based on Vivado's Ultrascale memory resources documentation [5]. As

Width	Depth			
	BRAM18K		BRAM36K	
	S2P	T2P	S2P	T2P
1	16,384	16,384	32,768	32,768
2	8,192	8,192	16,384	16,384
4	4,096	4,096	8,192	8,192
9	2,048	2,048	4,096	4,096
18	1,024	1,024	2,048	2,048
36	512	-	1,024	1,024
72	-	-	512	-

Table 1. BRAM configurations.

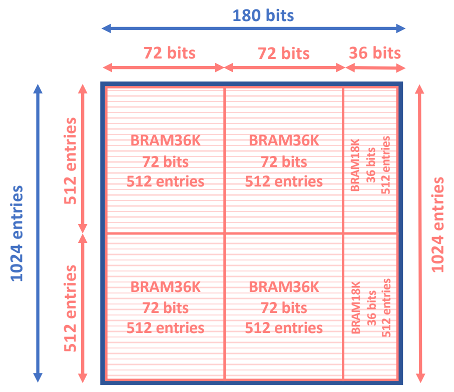


Fig. 10. Memory cores implementation via BRAM.

discussed in Section 3.2.2, assuming each physical (BRAM or URAM) memory bank has a bit-width of W and a depth of D_p, and each logical memory core in the buffer channel requires a width of w and depth of d_p, then each logical memory core needs $\left\lceil \frac{d_p}{D_p} \right\rceil \times \left\lceil \frac{w}{W} \right\rceil$ physical memory banks to implement.

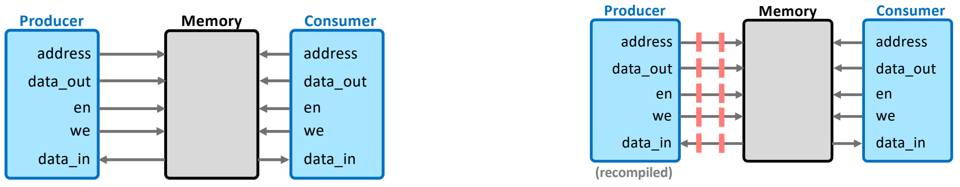
5.1.1 URAM Memory. In the Ultrascale series, URAM (UltraRAM) is a dual-port 288 Kb memory block. These memory blocks are hard configured with a width of $W = 72$ bits and depth of $D_p = 4096$. The whole range of memory can be accessed by two independent ports, allowing one read or write operation per port per cycle. The number of URAM blocks required to implement one logical memory core is $\left\lceil \frac{d_p}{4096} \right\rceil \times \left\lceil \frac{w}{72} \right\rceil$.

5.1.2 BRAM Memory. In the Ultrascale series, BRAM (BlockRAM) is a highly configurable resource, which makes its modeling much more challenging. A BlockRAM unit can house up to 36 Kbits of data and can be configured either as two independent `BRAM18K` blocks or a single `BRAM36K` block. Furthermore, either of the blocks can be configured as a Simple Dual-Port (S2P) or a True Dual-Port (T2P) block. S2P refers to a configuration where the memory block has two ports, one being a read-only port and the other being a write-only port. In the T2P configuration, the memory block has two ports, each one supporting either a read or write per clock cycle. Note URAM only supports T2P.

The variety of possible configurations for each BRAM block are shown in Table 1. Note that for the same BRAM resource type (e.g., BRAM18K or BRAM36K), S2P and T2P offer the same amount of memory, but S2P can allow a wider width W. Therefore, it is beneficial to use S2P memories whenever possible as that can lead to resource saving in certain cases. For example, if a memory core with a width of 36 bits and depth of 512 is needed, based on Table 1, a single BRAM18K is sufficient if S2P cores are required. If T2P cores are required, a BRAM36K core needs to be used.

Upon conducting numerous experiments to analyze how Vivado implements memory cores, we devise a simple algorithm to model BRAM resource consumption by attempting to satisfy the width and depth requirements based on Table 1 while minimizing the total size of BRAM resources consumed.

Figure 10 shows an example implementation of a memory core with a width of 180 bits and a depth of 1024 entries using S2P BRAM blocks. First, a total width of 180 bits is desired. A single S2P BRAM18K core can give only 36 bits, and a single S2P BRAM36K core can give 72 bits. So neither of them is sufficient. One can realize that two S2P BRAM36K cores configured at width of 72 each and another S2P BRAM36K (or S2P BRAM18K) core configured at width of 36 bits stacked side by side can give us a total width of 180 bits. Second, a total depth of 1024 entries is desired. Each of the two S2P BRAM36K cores configured at 72 bits can give a depth of 512 entries. Therefore, we need to vertically stack them



(a) Memory channel w/o pipelining (b) Pipelined memory channel

Fig. 11. Placing and pipelining memory channel.

twice to get a total depth of 1024. For the remaining width, two S2P BRAM18K cores vertically stacked together suffice to provide the total depth of 1024.

5.1.3 Buffer Model Accuracy. To test the accuracy of our buffer channel resource modeling, we conduct an experiment on a total of 240 memory core configurations with various widths, depths and simple/true dual-port. For each configuration, we compare our resource modeling with the synthesis results from Vivado. We find that our model has an average error rate of 0.35 BRAM banks off. For the six real benchmarks evaluated in this paper, our buffer channel model accuracy is 100%.

5.2 Coarse-Grained Floorplanning and Pipelining for Buffer Channel

Here we discuss the placement and pipelining of the buffer channel, which consists of two FIFO channels and one or more memory channels. The placement and pipelining of the FIFO channel is discussed in the Autobridge [25] paper, we briefly describe it here for completeness sake. Then, we focus on the placing and pipelining of a memory channel, which is crucial in order to pipeline the buffer channel.

5.2.1 Pipelining a FIFO Channel. The FIFO channel is pipelined from the producer side by introducing pipeline registers on the ‘write’, ‘full’ and ‘data’ signals. This ensures correct functionality when the producer wants to write data, except that the additional delay on the ‘full’ signal would prevent the producer from immediately knowing when the FIFO is full. Therefore, the producer may end up writing more data to the FIFO when in reality it is already full and no longer has the capacity to take more. This is solved by increasing the depth of the FIFO to be able to take this extra data while at the same time modifying it so that it asserts its full signal when it reaches its original depth, not when it reaches the new depth.

5.2.2 Placing and Pipelining a Memory Core. Consider the `ap_memory` interface as shown in Figure 11a. In order to write to the buffer channel, it needs to assert the `address`, enable (`en`), write_enable (`we`) and the data at `data_out` before the clock edge hits and make sure it is held till after the edge. In order to read a value, it needs to assert the `address` and `en` before the clock edge hits and read the value at the next clock edge hit. If extra latency is added on these paths, as shown in Figure 11b, write operations still happen, except that it takes more cycles for them to complete. For example, if two (l) registers are added on the paths, it will take an extra two (l) cycles for the write to actually happen. For the read, however, the task needs to wait for more cycles before it is able to read. The amount of extra wait it needs to do will be $2l$ if l is the number of registers added. In order to ensure functional correctness, this task needs to be recompiled to make sure it respects the added latency. Since a producer task is less likely to do any reads, we decide to place the memory core near the consumer side, and insert pipeline registers between the producer and memory core to shorten its critical path.

5.2.3 Impact of Pipelining a Memory Core on Producer Task. If the producer task only writes data and never reads it, pipelining the memory channel has no effect on the producer as highlighted in Section 5.2.2. If the producer task also reads, pipelining the memory channel increases the individual latency of each of the read operations by $2l$ where l is the number of registers added on the channel. However, in many cases, the compiler can overlap the read operations and the added latency can be hidden. It depends on the task's design itself whether this can be hidden or not.

1). Case 1: If a pipelined memory channel is read in a pipelined loop that has an initiation interval of *II*, and the added latency on the memory channel only increases the depth of the pipeline without changing its *II*, then the total latency of the loop may increase by a small constant, which does not depend on the loop trip count.
2). Case 2: Otherwise, in the rare case that the loop *II* is dependent on the memory channel's read latency, e.g., in the case of a *read-after-write* dependency on the memory channel, the *II* can potentially increase after the memory channel is pipelined, which increases the total loop latency by a factor that depends on the loop trip count.
3). Case 3: Similarly, in the rare case of a memory channel being read in loop that is not pipelined, pipelining the memory channel can increase the loop iteration latency, which ultimately increases the total loop latency by a factor depends on the loop trip count.

To avoid the harmful latency in cases 2 and 3, when PASTA detects that the producer task also reads the buffer channel, by default, PASTA adds another constraint during floorplanning to map the producer and consumer tasks to the same FPGA slot. That is, no pipelining for the buffer channel would be needed and no extra latency would be added. If the producer task does not read the buffer channel, no such constraint is added. Meanwhile, even if the producer task also reads the buffer channel, PASTA still gives users an option (a flag to the tool) to separate the producer and consumer tasks to different slots if they want to. In such cases, as discussed in Section 4.4.4, the `latency` component of the `ap_memory` interface pragma is modified to take into account such added latency.

In our experience, in all practical designs, there is always an opportunity to design the tasks such that the tool can have the freedom to distribute them without having to deal with harmful latencies.

5.2.4 Placing and Pipelining the Buffer Channel. As shown in Figure 12, to floorplan the buffer channel, its *occupied sections FIFO* is placed on the consumer side and needs to be pipelined from the producer side, the *free sections FIFO* is placed on the producer side and needs to be pipelined from the consumer side, while the memory channels need to be placed near the consumer and pipelined from the producer side. If the producer task reads the memory channel, it needs to be recompiled to respect the additional latency that can affect the read operations.

To put things altogether, the floorplanning stage first decides which slot a task is mapped to. Following that, a routing stage takes place which decides how every edge between a producer and a consumer is laid out on the board. For every edge e_i whose producer is placed in slot s_p and consumer is placed in slot s_c, the routing output gives a path $s_p \longrightarrow s_1 \longrightarrow s_2 \longrightarrow \cdots \longrightarrow s_c$ depicting how the the channel is to be placed. Figure 12 shows an example: the producer has been placed in slot 1 and the consumer has been placed in slot 4. And the routing stage decides to place the channel along $s_1 \longrightarrow s_2 \longrightarrow s_4$. The occupied sections FIFO and the memory cores module are placed near consumer and pipelined from the producer side, free sections FIFO is placed near the producer and pipelined from the consumer side.

6 EXPERIMENTAL RESULTS

This section discusses the experimental results. We first briefly delve into the benchmarks themselves and how their task graphs look like, and describe other experimental setup. Then, we present the frequency and execution time improvements, the resource utilization, and explain the trends.

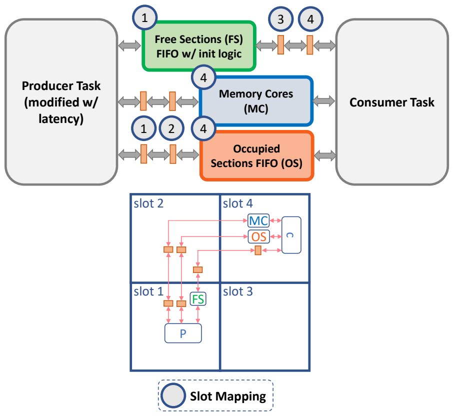

Producer Task (modified w/ latency)
Free Sections (FS) FIFO w/ init logic
Memory Cores (MC)
Occupied Sections FIFO (OS)
Consumer Task
slot 1
slot 2
slot 3
slot 4
MC
OS
FS
P
C
Slot Mapping

Fig. 12. Floorplanning of buffer channel

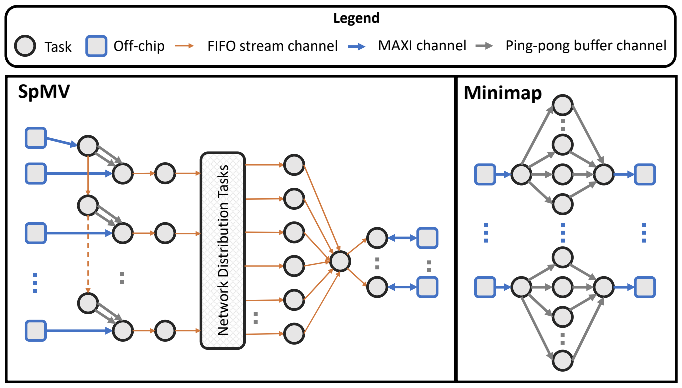


Fig. 13. Task graphs for Minimap and SpMV benchmarks

6.1 Benchmarks

To test the frequency optimizations on the buffer channel, we use benchmarks that heavily rely on buffer channels. There are a total of six such benchmarks. The first is a Sparse Matrix-Vector Multiplication (SpMV) accelerator called HiSpMV [38] that is designed using our framework PASTA by our colleagues at the HiAccel lab. The second is a genome sequencing accelerator called Minimap2 [27] from UCLA's VAST research group. Lastly, we port four benchmarks from the widely used Rodinia HLS benchmark suite [19] which builds upon the popular Rodinia GPU benchmarks. The task graphs of these six benchmarks are shown in Figure 13 and 14.

1). **SpMV [38].** As shown in Figure 13, the HiSpMV accelerator [38] performs sparse matrix-vector multiplication for imbalanced matrices. The input vector is loaded and given to the computation tasks via a hybrid buffering scheme that is implemented using buffer channels from our framework. The input sparse matrix tiles are streamed in and given to the computation tasks using FIFO channels. Afterwards, a series of tasks perform computations in a distributed manner (the network distributions tasks block is a set of many tasks shown as one in the figure) and all communicate using FIFO channels. And ultimately the resultant vectors are stored back in off-chip memory. HiSpMV uses both buffer channels and FIFO channels in the design.

2). **Minimap [27].** As shown in Figure 13, the Minimap accelerator [27] is essentially an accelerator for the *chaining* component of the overlap detection algorithm of the Minimap2 tool[1]. A set of loader tasks read sequences of anchor points and supply it to compute tasks via buffer channels. The compute tasks perform the actual chaining and the results are gathered by a set of store tasks via buffer channels and ultimately written back to off-chip memory.

3). **KNN [19].** As shown in Figure 14, the KNN benchmark implements the K-Nearest Neighbors algorithm. In our benchmark, the algorithm operates in 2D space and finds the 8 closest points. The input data is loaded by a set of loader tasks and sent to a set of distance computing tasks via buffer channels. The distance computation tasks consume the points buffers, compute the distance from the query point and produce buffers containing the points and indices for K partially closest points. These are consumed by a merge task which ultimately produces the top K candidate points and their indices.

4). **KMeans [19].** As shown in Figure 14, the KMeans benchmark implements the KMeans Clustering algorithm. In our benchmark, the algorithm operates in 2D space and clusters the input data around 8 centroids. The input centroids

[1]https://github.com/lh3/minimap2

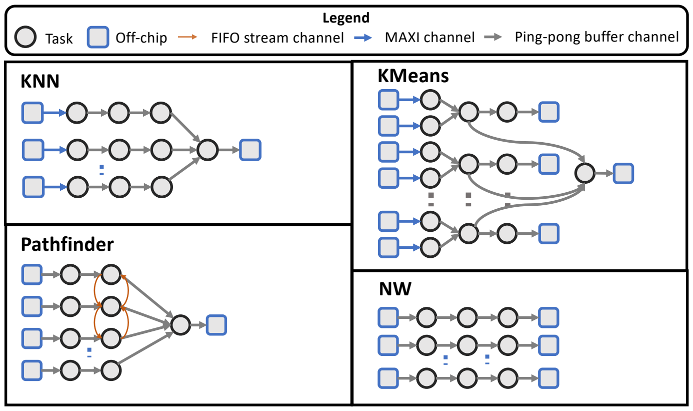


Fig. 14. Task graphs for four benchmarks derived from Rodinia HLS

are loaded from off-chip memory to distance calculating tasks via buffer channels. A set of data loader tasks load the input points and send it to distance calculating tasks via buffer channels. The distance calculating tasks find the closest centroid for each point and accordingly change the membership of that point. Along with this, it also computes new centroids based on the share of the data it sees. The membership is sent to a set of store tasks via buffer channels which ultimately writes it to off-chip memory. The new estimates of centroids from each of the distance calculating tasks are sent to a merge task via buffer channels which finds the final centroids and write it back to off-chip memory from where they can be loaded again in the next iteration.

5). **Pathfinder [19].** As shown in Figure 14, the Pathfinder benchmark implements the pathfinder algorithm. The input image is divided vertically into columns. Each column is taken care of by each PE. Each PE loads a tile of data from its column and performs computations row by row, bottom to top. At every row, PEs exchange data from neighboring PEs using FIFOs. Ultimately, the final data is merged and written back to off-chip memory by a merge task. Pathfinder uses both buffer channels and FIFO channels in the design.

6). **NW [19].** As shown in Figure 14, the NW benchmark implements the Needleman-Wunsch algorithm. A set of input loader tasks load the input data sequences and send it to computation tasks via buffer channels. The computation tasks align the sequences and send the aligned sequences to store tasks which write them to off-chip memory.

Among these benchmarks, KNN, KMeans, Pathfinder and NW are all compute-bound implementations, while SpMV and Minimap are memory-bound implementations. Pathfinder and SpMV use both buffer channels and FIFO channels in the design, while the rest only use buffer channels.

6.2 Other Experimental Setup

We implement our framework PASTA on top of TAPA/Autobridge [24] and test it using the aforementioned benchmarks. All these benchmarks are scalable, in the sense that we can use more memory banks and replicate more PEs to process more data in parallel. For each benchmark, we sweep the design across the number of PEs as shown in Figure 15. For the KNN, KMeans, Pathfinder and NW benchmarks shown in Figure 15, the baseline is written purely in Vitis HLS C++ and compiled using the standard Vitis HLS flow. For the SpMV and Minimap benchmarks shown in Figure 15, the baseline is written using PASTA but the frequency optimizations are disabled. Our experiments have been performed on the AMD/Xilinx Alveo U280 FPGA that have three SLRs and HBM banks. All benchmarks are built with a target

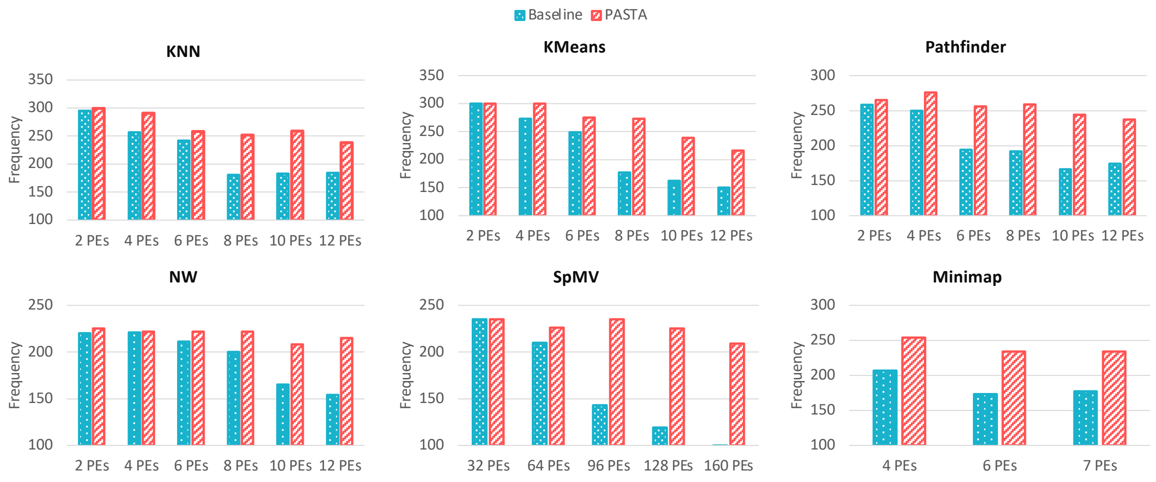


Fig. 15. Frequency improvement for KNN, KMeans, Pathfinder, NW, SpMV, and Minimap benchmarks on Alveo U280

frequency of 300 MHz except SpMV which requires to be built with a target frequency of 235 MHz to ensure correctness of the results. For all design points, we do on-board hardware execution and measure the execution time.

6.3 Frequency Results

The frequency improvement results for all design points on the AMD/Xilinx Alveo U280 FPGA board are shown in Figure 15. We test across six benchmarks and 32 design points with different number of PEs. In all of these sub-figures, the x-axis shows the number of PEs which is a measure of the scale of the design. The more PEs in an accelerator, the more data it can process in parallel. The y-axis shows the achieved frequency after placement and routing by the Vivado tool. The blue bars show the baseline's achieved frequency while the red one shows PASTA's achieved frequency. Complete post placement and routing frequency and resource utilization results are also shown in Table 2.

As a general trend, we observe that as the number of PEs increases, the baseline frequency starts to drop. The drop in frequency becomes severe at the resource heavy designs. Consider for example, the 12 PE design of KMeans and 128 PE design of SpMV, the achieved frequency for baseline designs are only 150 MHz and 119 MHz, respectively. In one extreme case of 160 PE for the SpMV benchmark, the baseline fails routing. The degradation in frequency with increasing number of PEs happens because of nets crossing FPGA slot boundaries becoming timing bottlenecks.

The PASTA version achieves similar frequencies to the baseline when the number of PEs is small. However, as the number of PEs increase, the PASTA version sustains a much better frequency than the baseline. PASTA improves the frequency by 25% on average across the 32 design points and up to 89% frequency improvement. Note that design points which are smaller in size (e.g., consume relatively less resources) tend to achieve decent frequencies in the baseline as well. Therefore, we do not observe large gains in frequency and this is expected. Such design points decrease the average frequency improvement number. If we average the resource consumption of each design point across different resource types (e.g, LUT, FFs, DSPs) and pick only the design points with an average resource consumption higher than 25%, we get an average frequency improvement of 46% on AMD/Xilinx Alveo U280 FPGA. Moreover, for the 160 PE design of the SpMV benchmark, the baseline fails routing while the PASTA version achieves a frequency of 209 MHz.

The NW benchmark achieves a comparatively lower frequency of ~220 MHz for both the baseline as well as the PASTA version even for its 2 PE design point which consumes ~20% resources. Upon investigation, we find out that this happens because the design inherently has some timing paths which have high logic delays and therefore become timing

Table 2. Resource utilization, frequency and performance gain results for all benchmarks on AMD/Xilinx Alveo U280

Benchmark	Config	LUTs (%)		FFs (%)		BRAMs (%)		URAMs (%)		DSPs (%)		Frequency (MHz)		F. Improv	Expected Perf. Gain	Actual Perf. Gain
		Baseline	PASTA	Baseline	PASTA	Baseline	PASTA	Baseline	PASTA	Baseline	PASTA	Baseline	PASTA			
Alveo U280																
KNN	2 PEs	16.87	16.88	9.36	9.59	20.41	20.61	0.00	0.00	4.30	5.01	295	299	1.01	1.01	1.01
	4 PEs	25.43	25.71	12.92	13.21	30.78	31.27	0.00	0.00	9.97	9.97	257	291	1.13	1.13	1.12
	6 PEs	33.99	34.61	16.21	16.78	41.15	41.48	0.00	0.00	14.94	14.94	242	258	1.07	1.07	1.06
	8 PEs	42.60	41.98	19.63	19.26	51.51	52.41	0.00	0.00	19.90	19.90	181	252	1.39	1.39	1.38
	10 PEs	51.55	52.26	23.05	24.17	61.88	62.97	0.00	0.00	24.87	24.87	183	259	1.42	1.42	1.40
	12 PEs	60.26	57.93	26.44	26.07	72.25	73.54	0.00	0.00	29.83	29.83	184	238	1.29	1.29	1.28
KMeans	2 PEs	16.12	16.41	12.35	12.61	15.50	15.67	0.00	0.00	9.11	9.11	300	300	1.00	1.00	1.01
	4 PEs	24.19	24.57	18.27	19.59	21.01	21.38	0.00	0.00	18.17	18.20	273	300	1.10	1.10	1.12
	6 PEs	32.22	32.72	25.19	26.47	26.51	26.98	0.00	0.00	27.22	27.24	249	275	1.10	1.10	1.14
	8 PEs	40.26	41.22	31.66	33.38	32.12	32.59	0.00	0.00	36.30	36.28	177	273	1.54	1.54	1.55
	10 PEs	49.00	49.54	38.06	40.26	37.62	38.19	0.00	0.00	45.35	45.35	162	239	1.48	1.48	1.46
	12 PEs	58.09	58.56	44.45	47.10	43.13	43.80	0.00	0.00	54.39	54.41	150	216	1.44	1.44	1.43
Pathfinder	2 PEs	10.12	9.69	6.69	6.72	17.93	17.93	0.00	0.00	0.04	0.04	258	265	1.03	1.03	1.04
	4 PEs	12.51	11.59	7.67	7.83	25.82	25.92	0.00	0.00	0.04	0.04	250	276	1.10	1.10	1.09
	6 PEs	14.88	13.48	8.65	8.92	33.71	33.80	0.00	0.00	0.04	0.04	194	256	1.32	1.32	1.32
	8 PEs	16.60	15.37	9.62	10.10	41.59	41.69	0.00	0.00	0.04	0.04	192	259	1.35	1.35	1.32
	10 PEs	17.98	17.22	10.61	11.23	49.48	49.58	0.00	0.00	0.04	0.04	166	244	1.47	1.47	1.43
	12 PEs	20.19	19.01	11.59	12.21	57.37	57.47	0.00	0.00	0.04	0.04	174	237	1.36	1.36	1.34
NW	2 PEs	12.23	12.29	8.73	8.85	20.44	20.44	0.00	0.00	0.04	0.04	220	225	1.02	1.02	1.02
	4 PEs	16.73	16.37	11.88	12.14	30.85	30.95	0.00	0.00	0.04	0.04	221	222	1.00	1.00	1.00
	6 PEs	20.44	20.70	15.03	15.42	41.27	41.37	0.00	0.00	0.04	0.04	211	222	1.05	1.05	1.00
	8 PEs	24.69	25.03	18.18	18.71	51.79	51.79	0.00	0.00	0.04	0.04	200	222	1.11	1.11	1.10
	10 PEs	28.90	29.35	21.33	22.13	62.20	62.20	0.00	0.00	0.04	0.04	165	208	1.26	1.26	1.23
	12 PEs	33.13	33.80	24.48	25.75	72.62	72.62	0.00	0.00	0.04	0.04	154	215	1.40	1.40	1.42
SpMV	32 PEs	18.09	18.27	11.61	11.45	22.82	22.82	10.00	10.00	7.23	7.23	235	235	1.00	1.00	1.00
	64 PEs	27.55	27.89	16.74	16.59	35.52	35.52	20.00	20.00	12.90	12.90	210	226	1.08	1.07	1.00
	96 PEs	37.95	38.66	22.59	22.85	48.21	48.21	30.00	30.00	20.01	20.01	143	235	1.64	1.57	1.48
	128 PEs	47.47	48.62	27.71	28.11	60.91	60.91	40.00	40.00	25.69	25.69	119	225	1.89	1.89	1.81
	160 PEs	45.73	47.65	24.80	25.48	73.61	73.66	50.00	50.00	17.18	17.18	-	209	-	-	-
Minimap	4 PEs	43.41	43.81	24.79	25.13	36.90	37.00	0.00	0.00	12.50	12.50	207	254	1.23	1.09	1.04
	6 PEs	61.36	61.81	34.37	34.35	50.45	50.45	0.00	0.00	18.73	18.73	173	234	1.35	1.30	1.24
	7 PEs	70.29	70.60	39.12	39.19	57.17	57.17	0.00	0.00	21.84	21.84	177	234	1.32	1.27	1.21

bottlenecks. Our work does not target frequency problems due to high logic delays. As the number of PEs increase, the baseline's frequency drops even more due to cross-slot boundary timing paths becoming timing bottlenecks. Therefore, the PASTA version achieves a good speedup in frequency of up to 40%.

6.4 Execution Time Results

The execution time gains are shown in the last column of Table 2, which are obtained by measuring the actual on-board execution time on hardware. For all design points, we list the frequency gain obtained, the expected performance gain and the actual performance gain based on on-board execution time measurements in Table 2.

For benchmarks that are compute-bound (KNN, KMeans, Pathfinder and NW), the performance expected gain should be exactly the same as the frequency gain, which we have verified using hardware simulations. There are some minor differences (1% on average and up to 5%) between the expected performance gain (i.e., actual frequency gain here) and actual performance gain, which are due to variance in the measurement of on-board execution time.

For benchmarks that are memory-bound (SpMV and Minimap), the frequency improvement will no longer directly translate to performance improvement once the accelerator starts to utilize the full memory bandwidth. When using HBM banks with an MAXI port width of 512 bits, the off-chip memory bandwidth of each single HBM bank gets fully utilized at 225 MHz [35, 36]. Therefore, operating the kernel at a frequency higher than 225 MHz does not lead to a proportional improvement in performance in such cases. To take this into account, we adjust the expected performance gain based on this frequency accordingly in the Minimap and SpMV benchmarks. There is an average of 6% (and up to 9%) difference between the expected performance gain and actual performance gain, and the expected performance gain is always larger than than the actual performance gain. We suspect this may be due to the dynamic access pattern

Maximum Cascade Height	Frequency
1 (No cascading)	242.5 MHz
2	233.3 MHz
3	218.1 MHz
4	227.9 MHz
5	229.3 MHz
16 (PASTA default)	234 MHz

Table 3. Cascade configurations

which we cannot easily model in the SpMV and Minimap benchmarks. The good news is that for these two benchmarks, PASTA does give a decent performance gain for on-board execution.

Across the total set of 32 design points on both boards, we observe an average frequency improvement of 25% , expected performance improvement of 24% and an measured performance improvement of 22%. Across large points (higher than 25% average resource consumption) on the Alveo U280 board, we observe an average frequency improvement of 46%, expected performance improvement of 45% and measured performance improvement of 42%.

6.5 Resource Results

The post placement and routing resource numbers are shown in Table 2. The baseline and PASTA version have an average resource difference of 0.18% and up to 2.65% in some extreme cases. This concludes that 1) our frequency optimizations have negligible resource overhead, and 2) our frequency improvements do not come from resource utilization improvements. Moreover, Table 2 also confirms that PASTA can work well for large designs that utilize 60%-70% of the FPGA resources to achieve a good frequency, which was quite challenging (if not impossible) for Vitis (HLS) baselines.

6.6 Floorplanning Overhead

We have profiled the overhead introduced by the floorplanning step on our benchmarks and have found it to be at the scale of minutes, while the actual placement and routing step takes hours, which could be 15-24 hours for large designs. Across all design points, we observe an average floorplanning time of 7 minutes. Percentage wise, we observe that the biggest overhead is in the SpMV-64PE design, where floorplanning takes about 9% (30 minutes) of the total placement and routing time.

6.7 Discussion on Cascade Height

In PASTA, we set a default maximum cascade height of 16 for all the buffers. As part of this work, we attempted to study the relationship of the maximum cascade height with the frequency (fmax). Examining our benchmarks, we observe that minimap is the only one that has buffers deep enough to potentially get affected by the maximum cascade height. We did a sweep from no cascading to maximum cascade height of 5 for the minimap benchmark, with the results shown in Table 3. We find that the build with no cascading achieves the highest frequency (242.5MHz vs our default 234 MHz) in this benchmark. However, there is no clear relationship between the maximum cascade height and the frequency. We think more experiments are needed on benchmarks that have buffer channels with deep memory cores in order to further study the impact of the maximum cascade height on frequency.

7 RELATED WORK

As we build our work on top of TAPA/AutoBridge[17, 24, 25], it is our closest related work. As explained in Section 1, while TAPA/AutoBridge is a great step in the direction of alleviating routing congestion by HLS and physical design co-optimization, its programming model is limited to task-parallel applications where tasks communicate via FIFOs only. In this work, we extend the support to a much broader class of designs where tasks can communicate via both FIFOs and buffers, by addressing a set of unique challenges to support buffer-based task communication as summarized in Section 1 and 3. Section 1 and 2.3 also summarized the new contributions in our PASTA framework.

Next, we discuss other related literature that focuses on optimization for multi-die FPGAs, HLS and physical design co-optimization, floorplanning algorithm and task-parallel programming models.

Optimization for multi-die FPGAs. Modern datacenter FPGAs often consist of multiple dies, which introduce both opportunities and challenges to scale up accelerator designs. [28] tries to minimize the number (not the length) of cross-die signals by modifying the cost function of placement. However, their approach is not HLS aware and does not shorten the critical paths that do cross dies. On the other hand, by being aware of the HLS design, we can pipeline latency-insensitive communication channels that cross dies to shorten the critical path. [44] proposes a solution to virtualize an FPGA by standardizing the partitioning of FPGA resources and mapping different applications (similar to independent tasks without communication channels) to different partitions. We focus on scaling up a single application which is more challenging. [43] places a NoC (Network-on-Chip) on the FPGA fabric and then partitions the application into components that communicate via the NoC to reduce the FPGA compilation (syntehsis) time.

HLS and physical design co-optimization. [47] proposes an approach to iterate between HLS and physical design, back annotating critical information in HLS to improve the frequency. While effective, it is not suitable for modern datacenter FPGA designs that are huge in size, where a single run of physical design tools can take many hours to a day. [41] uses graph neural networks to predict operation mapping and get better delay estimates. However, it is limited to operations while we are tackling the problem of routing congestion that has high net delays. [12, 46] work on predicting routing congestion in advance using machine learning methods and [40] even back-annotates it on the high-level source code. While effective in identifying congestion regions, solving it is left to the user. [26] analyzes common patterns that can lead to timing problems and suggest potential fixes; but again, is left to the user to fix the timing issues. [29] and [42] target pipelining dynamic HLS circuits at a fine-grained level to improve critical paths and throughput with register buffers. Our work targets coarse-grained floorplanning and pipelining in a layout aware manner to prevent communication channels that cross slot boundaries from becoming timing bottlenecks.

Floorplanning optimization. Floorplanning for heterogeneous FPGA has been studied in many previous efforts to fit hierarchical applications. [16] presents one of the first FPGA floorplanning algorithms for FPGAs with heterogeneous resources. It uses a slicing tree to present FPGA floorplan and develops an algorithm to find the optimal slicing structure. [14] uses a three-stage solution that generates a partition tree with all modules first, then generates a set of floorplan topologies and lastly realizes a feasible slicing tree for a specific FPGA. Unlike [16] and [14] that use simulated-annealing mode and deterministic model respectively to generate feasible topology, we focus on coarse-grained floorplanning with a partitioning-based approach that is similar to TAPA/AutoBridge [24, 25].

Programming models. Recent work [33] presents a buffer abstraction called a *unified buffer* and provides a toolflow that can take a restricted Halide program as input and compile it to an efficient implementation on CGRAs. Their implementation consists of a buffer extraction step that extracts abstract buffers from the input design and a mapping step that implements the abstract buffers on actual target hardware. Unlike our implementation, the input programming model is restricted to Halide designs and they do not target frequency optimizations to address scalability on multi-die FPGAs. [37] presents an end-to-end framework that takes an input program with implicit parallelism, extracts and builds a dynamic task-graph out of it and ultimately generates RTL code. Their approach extracts implicit fine-grained parallelism and extracts dynamic tasks out of it to generate an efficient accelerator while our work deals with explicit coarse-grained parallelism in static task-graphs with the goal of optimizing frequency on multi-die FPGAs.

8 CONCLUSION

In this work, we have developed the open-source toolflow PASTA that can automatically compile a scalable task-parallel HLS design onto a high-frequency accelerator design on modern multi-die FPGAs. Our work enables users to design their applications using a highly flexible task-parallel model that supports parallel and/or dataflow tasks that communicate via FIFO stream channels and buffer channels. We achieve this by designing a buffer channel abstraction and providing an implementation of it that is latency-insensitive, supports ping-pong buffering and memory partitioning, and compatible with Vitis HLS. In the frontend, we add easy-to-use APIs to enable users to easily use the buffer channel. In the backend, we implement placement and pipelining techniques for the buffer channel. Across 32 design points, we observe an average frequency improvement of 25% and up to 89%. Across large design points on the AMD/Xilinx Alveo U280 board, we observe an average frequency improvement of 46%. This demonstrates the effectiveness of our tool in achieving high-frequency designs on modern multi-die FPGAs. PASTA is open sourced here: https://github.com/sfu-HiAccel/pasta.

9 ACKNOWLEDGEMENT

We thank the anonymous reviewers for their valuable feedback. This work is supported in part by NSERC Discovery Grant RGPIN-2019-04613, DGECR-2019-00120, Alliance Grant ALLRP-552042-2020; CFI John R. Evans Leaders Fund; Huawei Canada, and AMD-Xilinx.

REFERENCES

[1] 2020. Alveo U280 Data Center Accelerator Cards Data Sheet. https://www.xilinx.com/support/documentation/data_sheets/ds963-u280.pdf Last accessed September 12, 2020.

[2] 2023. Altera Gives Intel a Hot Hand in Programmable Chips. https://fortune.com/2015/12/28/intel-completes-altera-acquisition/. [Accessed 01-11-2023].

[3] 2023. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/. [Accessed 01-11-2023].

[4] 2023. AMD Adaptive Computing Documentation Portal: SSI Technology Considerations. https://docs.xilinx.com/r/en-US/ug949-vivado-design-methodology/SSI-Technology-Considerations. [Accessed 01-11-2023].

[5] 2023. AMD Adaptive Computing Documentation Portal: Ultrascale Memory Resources. https://docs.xilinx.com/v/u/en-US/ug573-ultrascale-memory-resources. [Accessed 03-11-2023].

[6] 2023. Clang: a C language family frontend for LLVM. https://clang.llvm.org/. [Accessed 06-12-2023].

[7] 2023. FPGA-accelerated compute-optimized instance family - FPGA as a Service - Alibaba Cloud Documentation Center. https://www.alibabacloud.com/help/en/fpga-based-ecs-instance/latest/fpga-accelerated-compute-optimized-instance-families. [Accessed 01-11-2023].

[8] 2023. Intel High-Level Synthesis Compiler. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html. [Accessed 06-12-2023].

[9] 2023. NP-series - Azure Virtual Machines. https://learn.microsoft.com/en-us/azure/virtual-machines/np-series. [Accessed 01-11-2023].

[10] 2023. Smart HLS Compiler Software. https://www.microchip.com/en-us/products/fpgas-and-plds/fpga-and-soc-design-tools/smarthls-compiler. [Accessed 06-12-2023].

[11] 2023. Vitis HLS. https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html. [Accessed 06-12-2023].

[12] Mohamed Baker Alawieh, Wuxi Li, Yibo Lin, Love Singhal, Mahesh A. Iyer, and David Z. Pan. 2020. High-Definition Routing Congestion Prediction for Large-Scale FPGAs. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 26–31.

[13] Dave Altavilla. 2022. AMD completes Xilinx acquisition and the obvious synergies spell great potential. https://www.forbes.com/sites/davealtavilla/2022/02/14/amd-completes-xilinx-acquisition-and-the-obvious-synergies-spell-great-potential/?sh=4923115f527e

[14] Pritha Banerjee, Susmita Sur-Kolay, and Arijit Bishnu. 2009. Fast Unified Floorplan Topology Generation and Sizing on Heterogeneous FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28, 5 (2009), 651–661.

[15] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D. Brown, and Jason H. Anderson. 2013. LegUp: An Open-Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems. *ACM Trans. Embed. Comput. Syst.* 13, 2, Article 24 (sep 2013), 27 pages.

[16] Lei Cheng and M.D.F. Wong. 2004. Floorplan design for multi-million gate FPGAs. In *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.* 292–299.

[17] Yuze Chi, Licheng Guo, Jason Lau, Young-kyu Choi, Jie Wang, and Jason Cong. 2021. Extending High-Level Synthesis for Task-Parallel Programs. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM).* 204–213.

[18] Jason Cong, Zhenman Fang, Yuchen Hao, Peng Wei, Cody Hao Yu, Chen Zhang, and Peipei Zhou. 2018. Best-Effort FPGA Programming: A Few Steps Can Go a Long Way. *CoRR* abs/1807.01340 (2018). arXiv:1807.01340

[19] Jason Cong, Zhenman Fang, Michael Lo, Hanrui Wang, Jingxian Xu, and Shaochong Zhang. 2018. Understanding Performance Differences of FPGAs and GPUs. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM).* 93–96.

[20] Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru Zhang. 2022. FPGA HLS Today: Successes, Challenges, and Opportunities. *ACM Trans. Reconfigurable Technol. Syst.* 15, 4, Article 51 (aug 2022), 42 pages.

[21] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. 2011. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 4 (2011), 473–491.

[22] Jason Cong, Peng Wei, Cody Hao Yu, and Peng Zhang. 2018. Automated Accelerator Generation and Optimization with Composable, Parallel and Pipeline Architecture. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC).* 1–6.

[23] Linfeng Du, Tingyuan Liang, Sharad Sinha, Zhiyao Xie, and Wei Zhang. 2023. FADO: Floorplan-Aware Directive Optimization for High-Level Synthesis Designs on Multi-Die FPGAs. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA 2023).* 15–25.

[24] Licheng Guo, Yuze Chi, Jason Lau, Linghao Song, Xingyu Tian, Moazin Khatti, Weikang Qiao, Jie Wang, Ecenur Ustun, Zhenman Fang, Zhiru Zhang, and Jason Cong. 2023. TAPA: A Scalable Task-Parallel Dataflow Programming Framework for Modern FPGAs with Co-Optimization of HLS and Physical Design. *ACM Trans. Reconfigurable Technol. Syst.* 16, 4, Article 63 (dec 2023), 31 pages.

[25] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. 2021. AutoBridge: Coupling Coarse-Grained Floorplanning and Pipelining for High-Frequency HLS Design on Multi-Die FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '21).* Association for Computing Machinery, New York, NY, USA, 81–92.

[26] Licheng Guo, Jason Lau, Yuze Chi, Jie Wang, Cody Hao Yu, Zhe Chen, Zhiru Zhang, and Jason Cong. 2020. Analysis and Optimization of the Implicit Broadcasts in FPGA HLS to Improve Maximum Frequency. In *2020 57th ACM/IEEE Design Automation Conference (DAC).* 1–6.

[27] Licheng Guo, Jason Lau, Zhenyuan Ruan, Peng Wei, and Jason Cong. 2019. Hardware Acceleration of Long Read Pairwise Overlapping in Genome Sequencing: A Race Between FPGA and GPU. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM).* 127–135.

[28] Andre Hahn Pereira and Vaughn Betz. 2014. Cad and Routing Architecture for Interposer-Based Multi-FPGA Systems. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '14).* Association for Computing Machinery, New York, NY, USA, 75–84.

[29] Lana Josipović, Shabnam Sheikhha, Andrea Guerrieri, Paolo Ienne, and Jordi Cortadella. 2020. Buffer Placement and Sizing for High-Performance Dataflow Circuits. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '20).* Association for Computing Machinery, New York, NY, USA, 186–196.

[30] Moazin Khatti, Xingyu Tian, Yuze Chi, Licheng Guo, Jason Cong, and Zhenman Fang. 2023. PASTA: Programming and Automation Support for Scalable Task-Parallel HLS Programs on Modern Multi-Die FPGAs. In *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM).* 12–22.

[31] Yi-Hsiang Lai, Ecenur Ustun, Shaojie Xiang, Zhenman Fang, Hongbo Rong, and Zhiru Zhang. 2021. Programming and Synthesis for Software-Defined FPGA Acceleration: Status and Future Prospects. *ACM Trans. Reconfigurable Technol. Syst.* 14, 4, Article 17 (sep 2021), 39 pages.

[32] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86.

[33] Qiaoyi Liu, Jeff Setter, Dillon Huff, Maxwell Strange, Kathleen Feng, Mark Horowitz, Priyanka Raina, and Fredrik Kjolstad. 2023. Unified Buffer: Compiling Image Processing and Machine Learning Applications to Push-Memory Accelerators. *ACM Trans. Archit. Code Optim.* 20, 2, Article 26 (mar 2023), 26 pages.

[34] Alec Lu, Zhenman Fang, Nazanin Farahpour, and Lesley Shannon. 2020. CHIP-KNN: A Configurable and High-Performance K-Nearest Neighbors Accelerator on Cloud FPGAs. In *2020 International Conference on Field-Programmable Technology (ICFPT).* 139–147.

[35] Alec Lu, Zhenman Fang, Weihua Liu, and Lesley Shannon. 2021. Demystifying the Memory System of Modern Datacenter FPGAs for Software Programmers through Microbenchmarking. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '21).* Association for Computing Machinery, New York, NY, USA, 105–115.

[36] Alec Lu, Zhenman Fang, and Lesley Shannon. 2022. Demystifying the Soft and Hardened Memory Systems of Modern FPGAs for Software Programmers through Microbenchmarking. *ACM Trans. Reconfigurable Technol. Syst.* 15, 4, Article 43 (jun 2022), 33 pages.

[37] Steven Margerm, Amirali Sharifian, Apala Guha, Arrvindh Shriraman, and Gilles Pokam. 2018. TAPAS: Generating Parallel Accelerators from Parallel Programs. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 245–257.

[38] Manoj Bheemasandra Rajashekar, Xingyu Tian, and Zhenman Fang. 2024. HiSpMV: Hybrid Row Distribution and Vector Buffering for Imbalanced SpMV Acceleration on FPGAs. In *To appear in the 32nd ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 2024)*.

[39] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. 2022. AutoDSE: Enabling Software Programmers to Design Efficient FPGA Accelerators. *ACM Transactions on Design Automation of Electronic Systems* 27, 4, Article 32 (feb 2022), 27 pages.

[40] Osama Bin Tariq, Junnan Shan, Georgios Floros, Christos P. Sotiriou, Mario R. Casu, Mihai Teodor Lazarescu, and Luciano Lavagno. 2021. High-Level Annotation of Routing Congestion for Xilinx Vivado HLS Designs. *IEEE Access* 9 (2021), 54286–54297.

[41] Ecenur Ustun, Chenhui Deng, Debjit Pal, Zhijing Li, and Zhiru Zhang. 2020. Accurate Operation Delay Prediction for FPGA HLS Using Graph Neural Networks. In *Proceedings of the 39th International Conference on Computer-Aided Design* (Virtual Event, USA) *(ICCAD '20)*. Association for Computing Machinery, New York, NY, USA, Article 87, 9 pages.

[42] Hanyu Wang, Carmine Rizzi, and Lana Josipovic. 2023. MapBuf: Simultaneous Technology Mapping and Buffer Insertion for HLS Performance Optimization. In *To appear in 2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD 2023)*.

[43] Yuanlong Xiao, Eric Micallef, Andrew Butt, Matthew Hofmann, Marc Alston, Matthew Goldsmith, Andrew Merczynski-Hait, and André DeHon. 2022. PLD: Fast FPGA Compilation to Make Reconfigurable Acceleration Compatible with Modern Incremental Refinement Software Development. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 933–945.

[44] Yue Zha and Jing Li. 2020. Virtualizing FPGAs in the Cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 845–858.

[45] Chen Zhang, Guangyu Sun, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. 2019. Caffeine: Toward Uniformed Representation and Acceleration for Deep Convolutional Neural Networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 11 (2019), 2072–2085.

[46] Jieru Zhao, Tingyuan Liang, Sharad Sinha, and Wei Zhang. 2019. Machine Learning Based Routing Congestion Prediction in FPGA High-Level Synthesis. *CoRR* abs/1905.03852 (2019). arXiv:1905.03852

[47] Hongbin Zheng, Swathi T. Gurumani, Kyle Rupnow, and Deming Chen. 2014. Fast and Effective Placement and Routing Directed High-Level Synthesis for FPGAs. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '14)*. Association for Computing Machinery, New York, NY, USA, 1–10.