A Unified Software Approach to Specify Pipeline and Spatial Parallelism in FPGA Hardware

Jongsok Choi, Ruolong Lian, Stephen Brown, and Jason Anderson
ECE Department, University of Toronto, Toronto, ON, Canada
legup@eecg.toronto.edu

***Abstract*—High-level synthesis (HLS) is increasingly becoming a mainstream design methodology for FPGAs. Whereas its previous applications were mostly limited to research and simple designs, it is now being used to tape-out real-world chips in production [1]. Advances in compiler and HLS research continue to improve the quality of HLS-generated hardware. Despite this, the ease-of-use of HLS tools remains a hurdle to its broad uptake, particularly by engineers without hardware skills. To this end, we propose using a well-known software technique to infer streaming parallel hardware in HLS. Specifically, we use the producer-consumer pattern, commonly used in multi-threaded programming, to infer the generation of hardware that can exploit both *pipeline* and *spatial* parallelism on FPGAs. Our proposed methodology allows one to create a design in software, using only standard software methodologies, that cannot only synthesize to streaming hardware, but also model the generated hardware more accurately than existing solutions from other state-of-the-art C-based HLS tools. We use four different real-world benchmarks to illustrate the use of our methodology, and how it can create circuits that are either pipelined, or pipelined *and* replicated, all from software. For comparison, we also use a commercial HLS tool to synthesize one of the benchmarks, and show that our methodology can produce competitive results to that of the commercial tool.**

I. Introduction

In recent years, many works have showcased the benefits of high-level synthesis (HLS) (e.g. [2, 3, 4, 5]). HLS can automatically synthesize hardware from software, allowing a designer to work more productively at a higher level of design abstraction. Compared to human-crafted HDL design, HLS offers shorter design and verification cycles, leading to shorter time-to-market – a crucial factor in product development as the lifetimes of modern electronic devices continues to shrink.

Traditionally, one of the biggest barriers to using FPGAs has been the design methodology, which required hardware design with complex and low-level HDLs that are difficult to use and time consuming to debug. HLS seeks to solve this issue, by providing software design entry, which ultimately will allow those without hardware knowledge to use FPGAs. With these advantages, and the associated promise of increased chip revenue, both Altera and Xilinx have invested heavily in HLS, with each offering a compiler that can produce high-quality circuits. With the recent advances in both industry and academia, HLS compilers have improved such that in specific cases, they automatically generate circuits with comparable performance and area to manually designed hardware [6].

The holy grail of HLS is to replace HDL design, or to minimize the need for it to certain limited applications. We have seen this in the past in both the software and hardware domains, at different levels of abstractions. In software, compilers have matured to the extent that *assembly* code is only necessary in limited circumstances, such as handling interrupts or communicating with I/Os. Even though hand optimizing assembly could lead to better performance, it is too cumbersome and time-consuming. In the hardware domain, manually laying out transistors nowadays is only done for high-performance chips with extremely high volumes, such as Intel processors. Even in these chips, only the most critical parts of the chip are manually laid out. Although a full-custom chip could provide higher performance, it is simply too costly to do. We believe that HLS compilers are also on the same trajectory. With continued improvements in quality-of-results and support for debugging/verification, HLS could replace the majority of HDL design in the future. However, aside from the quality of circuits produced by HLS, another key factor impeding HLS adoption is *usability*, which is the focus of this paper.

We believe that HLS usability by software engineers can be improved by providing mechanisms within HLS that permit widely used software techniques to be "re-purposed" to control HLS tool behaviour. In multi-threaded parallel software programming, a popular development pattern is the *producer-consumer* pattern, wherein concurrently operating threads receive (consume) "work to do" from other threads and generate (produce) results that are then consumed by other threads. In a typical producer/consumer implementation, queues/buffers are used between the threads as staging areas for work items that have been produced but not yet consumed.

We observe an analogy between the producer/consumer pattern in multi-threaded software and *streaming kernels* in hardware, i.e. hardware modules interconnected by FIFO buffers that process their inputs in a pipelined manner and deposit results into output FIFOs. Streaming hardware is popular in applications such as audio/video processing. Commercial HLS tools, such as Xilinx Vivado HLS, support the specification of streaming via special vendor-specific pragmas embedded in the source. Conversely, we propose to *automatically* infer streaming hardware behaviour by synthesizing instances of the producer-consumer pattern in software, running on Pthreads, into streaming hardware. This methodology allows streaming hardware to be specified using a well-known software methodology with software execution behaviour that closely aligns with the hardware behaviour.

In our approach, each software thread is automatically synthesized into a streaming hardware module. FIFOs between the hardware modules are automatically instantiated, corresponding to the work-queue buffers in the producer/consumer pattern. Exploiting the spatial parallelism available in a large FPGA (such as Xilinx UtraScale [7]) becomes a matter of forking multiple threads. The proposed approach brings the added benefit that the multi-threaded code can be executed in parallel fashion in both software and hardware. Debugging and visualization can be done in software – software whose parallel execution matches closely with the parallel hardware execution. We implement our ideas within the LegUp open-source HLS framework [8] from the University of Toronto.

The contributions of this work are:

1) Inferring streaming hardware from parallel software written using Pthreads. To the best of our knowledge, this is the first HLS tool with this capability.
2) Using the producer-consumer pattern in software, which models the generated streaming hardware more accurately.
3) Using only standard software techniques that can all be compiled and debugged through standard software tool-chains, such as `gcc` and `gdb`, to create streaming hardware.
4) Quantitatively evaluating the hardware generated by the proposed methodology, for both pipelined and pipelined and replicated architectures.

Our work represents a step towards improving the usability of HLS by using a well-known software technique to infer parallel streaming hardware. The remainder of this paper is organized as follows: Section II presents related work. Section III provides an overview of the producer-consumer model used in software. Section IV discusses how we implement this in hardware. An experimental study is described in Section V and conclusions with recommendations for future work are presented in Section VI.

II. Related Work

There are a number of academic works which have used threads on an FPGA. HybridThreads (hthreads) provides a library which allows to execute threads on a hybrid CPU/FPGA system [9]. While similar to Pthreads, hthreads is not a standard software library, thus not portable to other platforms. It does not create *streaming* hardware from software threads, and also requires a CPU/FPGA hybrid system, neither of which are the case in our work. [10] describes a framework which employs Pthreads to generate hardware accelerators at run-time using on-chip CAD tools. This framework also requires a hybrid system, but with an ARM processor running an OS. [11, 12] provide OS abstractions to allow communicating with hardware threads on an FPGA. Each provides their own thread APIs, which are not standard software APIs, and do not provide the HLS capability of synthesizing software threads to hardware.

On the commercial front, there are a number of HLS tools that can generate streaming hardware. Altera's OpenCL Compiler [13] automatically creates deeply pipelined hardware from OpenCL kernels. Vivado HLS [14] and Impulse CoDeveloper [15] drive hardware generation with the use of pragmas in the code. In Vivado HLS, an entire function can be compiled to a *pipelined* hardware module by specifying the `HLS pipeline` pragma on the function. Data can be passed into the kernel as a stream using a Xilinx-specific type from a library, which gets turned into a FIFO in hardware. Impulse CoDeveloper can also pipeline a loop using its vendor pragma, `CO PIPELINE`. It also provides its own APIs to stream inputs/outputs to the pipeline.

We compare our work mostly to Vivado HLS, since it bears the most similarity to LegUp HLS in terms of its programming model and the input language. A code snippet is shown below for Vivado HLS.

```
void func_A(hls::stream<int>& in, hls::stream<int>& temp) {
  #pragma HLS pipeline II=1
  // read from FIFO
  int a = in.read();
  // do work
  ...
  // output to FIFO
  temp.write(b);
}
...

void top(hls::stream<int>& in, hls::stream<int>& out) {
  ...
  #pragma HLS dataflow
  func_A(in, temp); func_B(...); func_C(...);
  ...
}

int main() {
  // declare FIFOs
  hls::stream<int> in, out;
  ...
  for (i=0; i<SIZE; ++i) {
    // write to input FIFO
    in.write(in_array[i]);
    // invoke top-level function
    top(in, out);
    // get result from the output FIFO
    out_array[i] = out.read();
  }
  ...
}
```

This example creates a streaming circuit for the top-level function `top`, which calls three sub-functions, `func_A`, `func_B`, and `func_C` (for clarity, only `func_A` is shown). Each of the sub-functions are fully pipelined with an II (initiation interval) of 1, meaning that a new input is received and a new output is produced by the circuit every clock cycle when it is in steady-state. The `HLS dataflow` pragma in the `top` function makes the sub-functions execute concurrently and in a pipelined fashion, rather than sequentially one after another (normal software semantics). Meaning that, `func_A`, `func_B`, and `func_C` operate in a *dataflow* style, commencing execution as soon as their inputs are ready. There are also intermediate FIFOs (such as `temp`), which connect the sub-functions together. The `main` function pushes data into the input FIFO, invokes `top`, then fetches results via the output FIFO.

This methodology is simple and intuitive. However, there are a number of issues with this approach, which make the software behaviour *different* from the generated hardware behaviour. In hardware, a streaming module is always running. It is not invoked a fixed number of times (`SIZE` times in the example above). A streaming module simply processes data whenever its input FIFO is non-empty. This differs significantly from the semantics of the software code for Vivado HLS.

A larger discrepancy arises owing to the `HLS dataflow` pragma. This tool-specific feature internally transforms sequential software to parallel hardware. Its parallel execution, however, cannot be compiled or debugged using standard software toolchains, such as `GCC` or `GDB` (the software will just execute sequentially, as it is written). This also means that any existing software needs to be re-written in this style to exploit parallelism, which increases design time. The Xilinx-specific pragmas are foreign concepts that are difficult to comprehend for software engineers, or even for hardware engineers who are not familiar with the tool. Another discrepancy also comes from the FIFOs. In Vivado HLS, streams are assumed to be of infinite size in software [16]. Therefore, it is not possible to validate in the `C` whether a stream (FIFO) is full. When compiled to hardware, the FIFOs have a default size of 1, unless specified otherwise by the user.

Broadly speaking, the Vivado HLS software streaming specification is different from the actual hardware that is produced. These discrepancies can lead to bugs in hardware

that are *invisible* in the software. It also makes the visualization of the generated hardware more difficult for a designer. In our work, we propose a method of writing software for streaming hardware which more closely models the hardware produced.

III. Producer-Consumer Threads in Software

The producer-consumer programming pattern comprises a finite-size buffer and two classes of threads, a *producer* and a *consumer* [17]. The producer stores data into the buffer and the consumer takes data from the buffer to process. This *decouples* the producer from the consumer, allowing them to naturally run at different rates, if necessary. The producer must wait until the buffer has space before it can store new data, and the consumer must wait until the buffer is non-empty before it can take data. The *waiting* is usually realized with the use of a software variable, `semaphore`. A semaphore is a `POSIX` standard [18], which allows processes and threads to synchronize their actions. It has an integer value, which must remain non-negative. To *increment* the value by one, the `sem_post` function is used, and to *decrement* the value by one, `sem_wait` function is called [19]. If the value is already zero, the `sem_wait` function will block the process, until another process increases the semaphore value with `sem_post`.

The pseudo-code below (taken from [20]) shows the typical producer-consumer pattern using two threads.

```
producer_thread {
  while (1) {
    // produce something
    item = produce();
    // wait for an empty space
    sem_wait(numEmpty);
    // store item to buffer
    lock(mutex);
    write_to_buffer;
    unlock(mutex);
    // increment number of full spots
    sem_post(numFull);
  }
}

consumer_thread {
  while (1) {
    // wait until buffer has data
    sem_wait(numFull);
    // get item from buffer
    lock(mutex);
    read_from_buffer;
    unlock(mutex);
    // increment number of empty spots
    sem_post(numEmpty);
    // consume data
    consume(item);
  }
}
```

In a producer-consumer pattern, the independent producer and consumer threads are continuously running, thus they contain infinite loops. The buffer is implemented as a circular array. Two semaphores are used, one to keep track of the number of spots available in the buffer, and another to keep track of the number of items in the buffer. Observe that updates to the buffer are within a critical section – i.e. a `mutex` is used enforce mutual exclusion on changes to the buffer itself.

IV. Producer-Consumer Threads in Hardware

As mentioned above, we believe that the producer-consumer pattern is an ideal software approach to describe streaming hardware. Streaming hardware is always running, just as the producer-consumer threads shown above. Different streaming hardware modules execute concurrently and independently, as with the producer-consumer threads. To fork threads, we use Pthreads, which is a standard known and used by many software programmers. Inputs and outputs are typically passed between streaming modules through FIFOs. The circular buffer described above is essentially a FIFO, with the producer writing to one end, and the consumer reading from the other end.

Using the producer-consumer pattern with Pthreads, we can re-write the example code shown in Section II as below.

```
void *func_A(PTHREAD_FIFO *in, PTHREAD_FIFO *temp) {
  ...
  while (1) {
    // read from FIFO
    int a = pthread_fifo_read(in);
    // do work
    ...
    // output to FIFO
    pthread_fifo_write(temp);
  }
}
...

void top(PTHREAD_FIFO *in, PTHREAD_FIFO *out) {
  ...
  pthread_create(func_A, ...);
  pthread_create(func_B, ...);
  pthread_create(func_C, ...);
  ...
}

int main() {
  // declare and size FIFOs
  PTHREAD_FIFO *in =
    pthread_fifo_malloc(/*width*/32, /*depth*/1);
  PTHREAD_FIFO *out =
    pthread_fifo_malloc(/*width*/32, /*depth*/1);
  // invoke top-level function
  top(in, out);
  // fill up the input FIFO, as soon as the FIFO has data
  // the hardware executes
  for (i=0; i<SIZE; ++i) {
    pthread_fifo_write(in, in_array[i]);
  }
  // get output from the output FIFO
  for (i=0; i<SIZE; ++i) {
    out_array[i] = pthread_fifo_read(out);
  }
  // free FIFOs
  pthread_fifo_free(in); pthread_fifo_free(out);
  ...
}
```

The main differences here are the use of an infinite loop, Pthreads, and `PTHREAD_FIFOs`[1]. The infinite loop keeps the loop body of the kernel function continuously running. We *pipeline* this loop, to create a streaming circuit. The advantage of using *loop pipelining*, versus pipelining the entire function, is that there can also be parts of the function that are not streaming (only executed once), such as for performing initializations. The `top` function, which is called only once, forks a separate thread for each of its sub-functions. The user does not have to specify the number of times the functions are executed – the threads automatically start executing when there is data in the input FIFO. This closely matches the *always running* behaviour of streaming hardware. In this example, each thread is both a consumer *and* a producer. It consumes data from its previous stage and produces data for its next stage.

The `PTHREAD_FIFO` functions provide users with a software API which they can use to create streaming hardware in HLS. `Pthread_fifo_malloc` sizes the FIFOs in software

[1] LegUp can also create streaming hardware using sequential `C` code without the use of Pthreads, similar to Vivado HLS, and the keyword `FIFO` is reserved for the type of FIFOs used for this style of code.

to be the same as those in hardware. Pthread_fifo_write pushes data into one end of a FIFO; previously stored data can be read from the other end with pthread_fifo_read. The pthread_fifo_read/write functions provide the blocking capability with the use of semaphores. This is described in more detail below. Pthread_fifo_free frees any memory allocated by pthread_fifo_malloc[2].

The multi-threaded code above can be compiled, concurrently executed, and debugged using standard software tools. We believe that portability is an important design consideration, and that a design should not be tied to a particular vendor, as is what happens when many vendor-specific pragmas are required to produce the desired hardware. Our method aims to keep the HLS source code as a standard software program.

A. FIFO Details

This section describes how we create a PTHREAD_FIFO and its associated functions. The PTHREAD_FIFO is defined as a *struct*:

```
typedef struct {
  // bit-width of the elements stored in the FIFO
  int width;
  // the number of elements that can be stored
  int depth;
  // data array holding the elements
  long long *mem;
  // keeps track of where in the array to write to
  unsigned writeIndex;
  // keeps track of where in the array to read from
  unsigned readIndex;
  // keeps track of the number of occupied spots
  sem_t numFull;
  // keeps track of the number of empty spots
  sem_t numEmpty;
  // mutual exclusion for data array access
  pthread_mutex_t mutex;
} PTHREAD_FIFO;
```

The elements of the *struct* are used to define the storage, its width/depth, and where to read/write from/to in the storage. The data array is used as a *circular* buffer to create the FIFO behaviour. Its type is a long long, making it capable of handling the largest standard C integer data type, though it can also be used to hold anything smaller. When compiled to hardware, the *width* variable is used to parametrize the hardware FIFO, which can be of any arbitrary width. Semaphores are employed to create the producer-consumer behaviour between threads and a mutex is used to ensure atomic access to the shared storage. When pthread_fifo_malloc is called, it allocates the data array and initializes all member variables, including the semaphores and the mutex. pthread_fifo_free frees all memories which have been allocated.

Using the struct, pthread_fifo_write follows the logic described in the producer_thread of the pseudo-code in Section III, and pthread_fifo_read follows the logic of the consumer_thread. Pthread_fifo_write first waits until there is an empty spot in the FIFO (using sem_wait on the numEmpty semaphore), then gets the lock, stores the data into the writeIndex position of mem, updates writeIndex, releases the lock, and finally increments numFull. Pthread_fifo_read waits until the FIFO is non-empty (using sem_wait on the numFull semaphore), gets the lock, reads the data at the readIndex position of mem, updates readIndex, releases the lock, and finally increments numEmpty.

[2]pthread_fifo_free is only used in software to prevent memory leaks. It is stripped away when compiled to hardware, as is described in Section IV-B.

Fig. 1: FIFO interfaces.

B. Hardware Architecture

In hardware, a PTHREAD_FIFO struct gets compiled to a hardware FIFO. Fig. 1 shows the interfaces between a FIFO and its producer, module A, and its consumer, module B. For a FIFO interface, modules use RVD (Ready, Valid, Data) signals, which is a typical hand-shaking interface used in streaming architectures. The semaphores of the PTHREAD_FIFO struct, which keep track of whether the FIFO is full/empty in software, are simply turned into the not_full and not_empty signals in hardware. On a call to pthread_fifo_write for module A, the not_full signal is checked, and if it is high, the data is written to the FIFO via the write_data signal. If the not_full signal is low, meaning the FIFO is already full, the out_ready signal of module A is de-asserted, which stalls module A. The stall logic is described below in more detail in Section IV-D. For pthread_fifo_read from module B, the not_empty signal is checked, and if it is high, the data is returned via the read_data signal. If the not_empty signal is low (FIFO is empty), the in_valid signal is de-asserted, which stalls module B. This implementation removes any additional hardware overheads from the semaphores/mutex, while still allowing software to be executed like hardware.

C. Multiple Software Threads to Multiple Streaming Hardware

In a streaming architecture, multiple streaming modules may be chained together, transferring data from one streaming module to the next, as shown in Fig. 2a. This is a typical architecture used in image/video processing applications. We can create this architecture by simply forking a thread for each of A, B, and C, as described above, and passing in FIFO0 as an argument to A and B, and FIFO1 and FIFO2 to B and C. As per Pthread standards, multiple arguments to a thread must be passed by creating a struct which contains all of the arguments, and then passing a pointer to that struct in the pthread_create() routine [21]. We use a points-to analysis to automatically determine which FIFOs need to be connected to which hardware modules. The tool also determines whether a module *writes* to the FIFO, or *reads* from the FIFO, and the integrated system generator [22] automatically connects the appropriate input/output FIFO ports to their corresponding streaming module ports.

With the producer-consumer threads, all processes, in both software and hardware, start executing as early as possible (i.e. as soon as there is data in the input FIFO). As previously mentioned, the dataflow pragma in Vivado HLS achieves a similar effect in hardware, but its parallel execution, which can often be the source of bugs, cannot be debugged in software using standard debugging tools. Since we only use standard software methodologies, all of our code, including the FIFO functions, can be compiled with GCC and debugged with GDB. As such, most of the design effort can be spent at the software stage.

Another advantage of using Pthreads is that one can also easily replicate streaming hardware. In LegUp, each thread is

mapped to a hardware instance [23], hence forking multiple threads of the same function creates replicated hardware instances[3]. For instance, if the application shown in Fig. 2a is completely parallelizable (say data-parallel), one can exploit spatial hardware parallelism by forking two threads for each function, to create the architecture shown in Fig. 2b[4]. This methodology therefore allows exploiting *both* spatial (replication) and pipeline hardware parallelism all from software.

For replication, other HLS tools require the hardware designer to manually instantiate a synthesized core multiple times and also make the necessary connections in HDL. This is cumbersome for a hardware engineer and infeasible for a software engineer. More recently, HLS tools have introduced system generator tools, such as the Vivado IP Integrator, which uses a schematic-like block design entry, and allows a user to interconnect hardware modules by drawing wires. This, also, is a foreign concept in the software domain. Our methodology uses purely software concepts to automatically create and connect multiple parallel streaming modules together.

Our approach is also able to handle more complex architectures, where multiple consumers receive data from a single producer through a single FIFO, as shown in Fig. 2c, and where multiple producers can feed data to a single consumer through a single FIFO, as shown in Fig. 2d. The former architecture can be useful for applications with a *work queue*, where a producer writes to the work queue, and multiple workers (consumers), when ready, take work-items from the queue to process. The latter architecture can be used for applications such as *mapReduce* [24], where multiple mappers can map to the same reducer. Both architectures can be created from software by giving the same FIFO argument to the different threads. We automatically synthesize arbiters to handle contention that may occur when multiple modules try to access the same FIFO in the same clock cycle – modules may stall if not given immediate access. We believe this one-to-many, or many-to-one FIFO architecture, with automatic synthesis of arbitration logic, is a unique aspect of our work, as both Vivado HLS and Altera's OpenCL Compiler require that there is only a *single* writer and a reader to/from a FIFO.

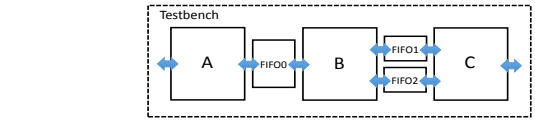
D. Streaming Datapath and Stall Logic

The datapath and its stall logic for streaming hardware is shown in Fig. 3. In the figure, there are two input FIFOs, a non-FIFO argument input, and two output FIFOs. The `S`'s denote pipeline stages, with registers at each stage to pipeline data. The valid bits are used to indicate which stages of the pipeline contain valid data. The streaming circuit is a straight-line datapath, without any control flow. Like other HLS tools, we remove any diverging branches with if-conversion and back edges by unrolling any internal loops (those residing inside the while loop). Any sub-functions called within the while loop are inlined. During if-conversion, we also predicate operations with side effects (i.e. load/store, FIFO read/write) so that they trigger for the correct if/else conditions.

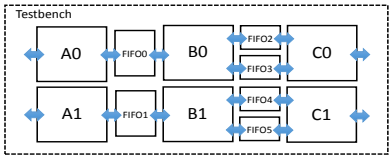
The stall logic ensures that the hardware can stall appropriately and produce a functionally correct result. It directly

[3]Prior work in [23] allowed replication of hardware via Pthreads, but did not permit hardware instances to be *streaming*. All work described in this paper, including FIFO implementations in software/hardware, generation of one or more connected streaming hardware modules, as well as their data-path/stall logic, are all new implementations of this work.

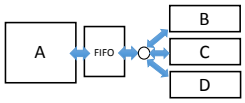
[4]It is not only limited to massively parallelizable architectures. Depending on the nature of the application, one can select which function to parallelize.



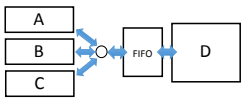
(a) Multiple streaming modules.



(b) Multiple replicated streaming modules.



(c) One-to-many streaming modules.



(d) Many-to-one streaming modules.

Fig. 2: Multiple streaming modules connected through FIFOs.

impacts the QoR (quality-of-result) of the circuit, as stalls increase circuit latency, and the stall logic affects circuit area and Fmax. It is desirable to stall only when necessary, and also to minimize the stall circuitry. For the architecture shown in Fig. 3, there are two scenarios wherein the circuit can stall: 1) When any of the input FIFO becomes *empty*, and 2) when any of the output FIFOs become *full*. In both cases, a stall does not necessarily stall the *entire* pipeline, but only those pipeline stages which absolutely need to stall. For instance, in the case of `Input FIFO0`, its data is required in `S0` (pipeline stage 0). Consequently, if this FIFO becomes empty, only `S0` stalls. Data from `Input FIFO1` is needed in `S1`, so if this FIFO is empty, `S1` and `S0` stall. `S0` also needs to stall in this case since its next stage is stalled (allowing it to continue would overwrite valid data in `S1`). `Output FIFO0` is written from `S2`, hence when this FIFO is full, it stalls `S2`, `S1`, and `S0`. When `Output FIFO1` is full, the entire pipeline stalls. In general, a FIFO being full/empty stalls the first pipeline stage where its data is read/written from, and all of the prior pipeline stages. This architecture allows the later pipeline stages to continue making forward progress, even when a FIFO becomes empty/full. For instance, when `S0` stalls due to `Input FIFO0` only, `S1`, `S2`, `S3` can continue. When `Output FIFO0` is full, valid data in `S3` can continue and be written to the `Output FIFO1` (given that it is not full).

There are also scenarios where stall circuitry is unnecessary. For instance, a *constant* argument (such as an integer value), is stored in registers when the module starts and remains unchanged during its execution. We do not create any stall logic for this argument, as it will not be overwritten during the execution. This helps to reduce circuit area and the fan-out of the stall signals, which can become large when there are many FIFOs and pipeline stages.

In summary, there are three conditions for a pipeline stage to be enabled: 1) Its valid bit must be asserted to indicate there is valid data, 2) any input FIFOs, from which its data is needed in this or a downstream pipeline stage, must not be empty, and 3) any output FIFOs, which are written to from this or a downstream pipeline stage, must not be full. A FIFO can also be shared between multiple modules through an arbiter, as was shown in Figs. 2c and 2d. In such cases, we stall in the

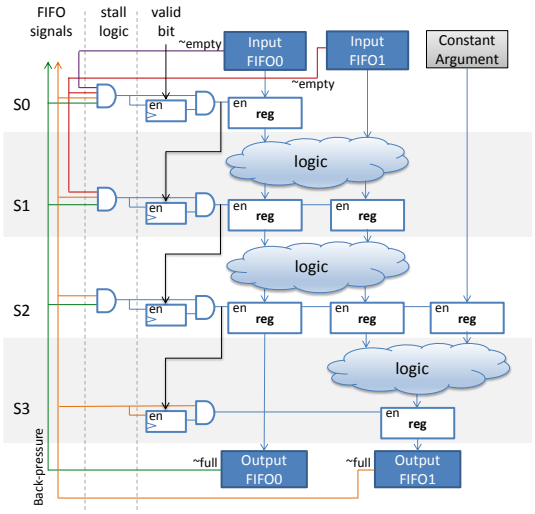


Fig. 3: Streaming circuit data-path and stall logic.

same manner, depending on whether it is an input or an output FIFO[5]. It is worth noting that, although we primarily discuss about FIFO memories in this work, streaming hardware can also access non-FIFO RAMs, with arbitration and stall logic created as necessary [22].

V. Experimental Study

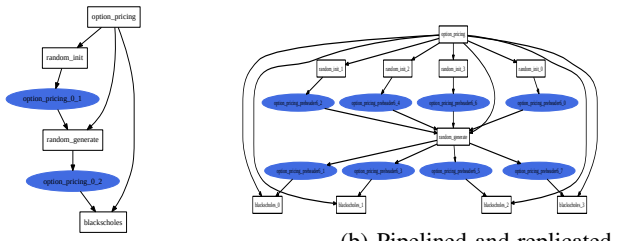
In this section, we first discuss the streaming benchmarks which use the producer-consumer pattern with Pthreads, as well as their resulting hardware. We use four different applications from various fields, including image processing, mathematics/finance and data mining. For each benchmark, we create two versions, a *pipelined-only* version and a *pipelined-and-replicated* version. In the pipelined-only version, there are one or more functions which are connected together through FIFOs, as in Fig. 2a, but no modules are replicated. For the pipelined-and-replicated version, we parallelize each benchmark with one or more functions (modules) executing on multiple threads, yielding architectures similar to Figs. 2b and 2d. In both versions, all kernel functions are fully pipelined with multiple pipeline stages, and receive/output new data every clock cycle (II=1).

Each benchmark also includes golden inputs and outputs to verify correctness. Each generated circuit was synthesized into the Altera Stratix V FPGA (5SGSMD8K1F40C2) with Quartus 15.0. For performance and area comparison, we also use a commercial HLS tool to synthesize one of the pipelined-only benchmarks, Canny, targeting the Xilinx Virtex 7 FPGA (XC7VX980TFFG1930-2). The commercial tool does not support replicating hardware from software, thus none of the pipelined-and-replicated benchmarks were used for this tool. For both LegUp and the commercial HLS tool, a 3ns (333MHz) clock period constraint was supplied; this is used by the scheduling stage of HLS to determine which operations can be chained together in a single clock cycle.

A. Benchmarks

Mandelbrot is an iterative mathematical benchmark which generates a fractal image. For each pixel in a 512 × 512 image, it iteratively computes whether it is bounded (inside the Mandelbrot set) or diverges to infinity (outside the Mandelbrot set), and displays its colour accordingly. Computations are done in fixed-point for this benchmark. Each pixel is independent from others, hence this application is easily parallelizable. In the pipelined-and-replicated version with four threads, each thread processes a quadrant of the image.

[5]For an input FIFO, the `grant` signal from the arbiter is `AND`'ed with the `not_empty` FIFO signal, and this output goes to the stall logic. For an output FIFO, the `grant` signal is `AND`'ed with the `not_full` FIFO signal.



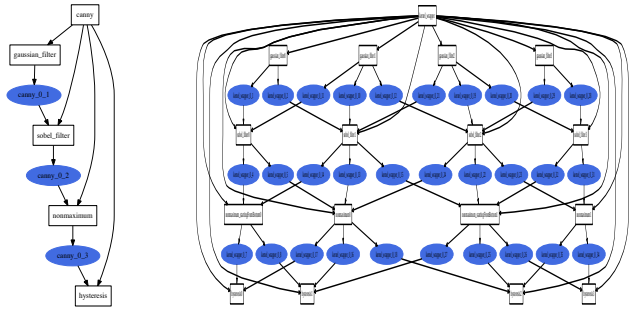
(a) Pipelined-only.

(b) Pipelined-and-replicated.

Fig. 4: System diagram for the Black-Scholes option pricing benchmark.

The *Black-Scholes* benchmark estimates the price of European-style options. It uses Monte Carlo simulation to compute the price trajectory for an option using random numbers. 10,000 simulations are conducted, with 256 time steps per simulation. The system diagram for the pipelined-only version is shown in Fig. 4a as a dot graph. This dot graph, automatically created by our system generator, shows the different modules, as well as the connections between them. White boxes are hardware modules; blue ovals are FIFOs; dark-blue overalls (appearing later) are memory block. This benchmark consists of three kernel functions, `random_init`, `random_generate`, and `blackscholes`, and the wrapper function, `option_pricing`, which creates the necessary intermediate FIFOs between the kernel functions and forks their threads. The `random_init` and `random_generate` are an implementation of the *Mersenne twister* [25], which is a widely used pseudo-random number generator. These two kernels were adapted from [26], originally written in OpenCL. The init function initializes the random number generator in the generate function. The blackscholes function uses the random numbers to price an European option using the Black-Scholes formula. In the pipelined-and-replicated version, shown in Fig. 4b, we parallelize the initialization and the Black-Scholes functions: each with four threads. For the generate function, we modify its logic so that it can receive four initializations from the initialization threads, and generate four random numbers concurrently. Each random number is used by an independent Black-Scholes' thread, with four threads concurrently computing four prices.

The *Canny* benchmark implements the well-known Canny edge detection algorithm [27] for a 512 × 512 image. The multi-stage algorithm is implemented with four kernel functions, `gaussian filter`, `sobel filter`, `nonmaximum suppression`, and `hysteresis`, as well as its wrapper function `canny`, as shown in Fig. 5a. The Gaussian filter first smooths the input image to remove noise. The Sobel filter then finds the intensity gradients. The non-maximum suppression removes pixels not considered to be part of an edge. Then finally, hysteresis finalizes the edges by suppressing all the other weak edges. Every clock cycle, each kernel receives a new pixel from the previous kernel stage and outputs a pixel to its next-stage kernel.



(a) Pipelined-only. (b) Pipelined-and-replicated.

Fig. 5: System diagram for the Canny benchmark.

In the pipelined-and-replicated version, we parallelize each kernel function with four threads. We again divide the image into four sections (this time with 128 rows each), with each section to be processed by a set of replicated modules (i.e. rows 0–127 are processed by a first set of copies of the Gaussian, Sobel, non-maximum suppression, and hysteresis kernel modules). The data required by each set of modules, however, is not completely mutually exclusive, since each kernel uses either a 5×5 or a 3×3 filter. For instance, the Gaussian filter, which uses a 5×5 filter, requires up to 2 rows outside of its assigned section. For example, when working on row 127, values of pixels in rows 128 and 129 are needed, which belong to the next section of rows. To manage this, pixel values for border rows are communicated between adjacent copies of the kernels. Moreover, to minimize stall time arising from needed data in border rows, even-numbered sections (containing rows 0–127 and rows 256–383) proceed from the bottom row to the top; odd-numbered sections (containing rows 128–255 and rows 384–511) proceed from the top row to the bottom. The architecture for this parallelized version is shown in Fig. 4b

The `k-means` benchmark implements the k-means clustering algorithm [28] used in data mining. It partitions `n` data points into one of `k` clusters defined by centroids. Our version has 1,000 data points with four clusters. We use the *mapReduce* programming paradigm to implement k-means. A `mapper` iteratively maps each data point to a cluster, and a `reducer` updates the centroids with each data point. In the pipelined-only version, there is a single mapper and a single reducer. The mapper maps all data points to one of the clusters, and the reducer updates the centroids for all clusters. In the pipelined-and-replicated version, there are four mappers and four reducers. Each mapper maps to a single cluster, and each reducer updates the centroid for a single cluster. The architecture for the parallelized version is shown in Fig. 6. The figure shows nine hardware modules (four mappers, four reducers, and the wrapper function), eight memories (four x and y centroid coordinates stored in registers), and many FIFOs. These FIFOs are used to send data inputs to each mapper, pass data from each mapper to each reducer (each mapper can write to any of the reducer FIFOs using the architecture shown in Fig. 2d), and also indicate when a mapper or a reducer is done computing for this iteration. For each iteration, a mapper needs to know when all reducers have finished (updated the centroids), so that it can start the next iteration using the updated centroids, and a reducer also needs to know when all mappers have finished so that it can average the accumulated centroid value. We use a 1-bit-wide depth-of-1 FIFO, which is implemented in registers, to send the *done* signal.

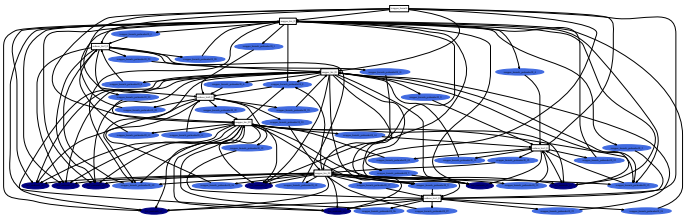


Fig. 6: System diagram for the pipelined-and-replicated k-means benchmark.

B. Results

Table I shows the performance and area results for all the pipelined-only benchmarks compiled with LegUp HLS. There are three performance metrics (total wall-clock time (# cycles $\times$ clock period), total number of clock cycles, and Fmax) and four area metrics (number of ALMs, registers, DSPs, and M20Ks). As previously mentioned, all circuits have an II=`1`, and were given a clock period constraint of `3ns` (333 MHz), except for Black-Scholes, which was given `6ns` (167 MHz). All circuits run roughly within +/-10% of the target frequency. For Black-Scholes, due to a recurrence in the benchmark, we had to lower the clock period constraint supplied to LegUp to meet II=`1`.

Table II shows the commercial HLS tool's result for the Canny benchmark. The performance results are nearly identical to that of LegUp HLS, with the total wall-clock time 0.6% higher than LegUp. Targetting the Virtex 7 FPGA, the area is reported in terms in LUTs, registers, DSP48s, and 18KB Block RAMS. The circuit generated by the commercial tool uses 15% more LUTs, but it also uses 19% less registers and half the number of RAMs. For this performance/area comparison, we note that there are differences in the FPGA architectures and the vendor FPGA CAD tools that can lead to different results. For example, although Virtex 7 and Stratix V are fabricated in the same `28nm` TSMC process, Stratix V uses fracturable 6-LUTs that are more flexible than Virtex 7's fracturable 6-LUTs. Likewise, we expect that two vendor's FPGA CAD tools employ different RTL/logic synthesis, place-and-route algorithms. Despite these potential sources of error, the similarity in the results for the Canny benchmark gives us confidence that LegUp HLS produces a reasonably good-quality implementation. In LegUp's case, this is achieved through software-only methodologies requiring no special pragmas.

Table III shows the results for LegUp HLS, for the pipelined-and-replicated benchmarks. Compared to pipelined-only, we see a geometric mean speedup of $2.8\times$ in terms of total wall-clock time. Clock cycle improvement is higher with $3.29\times$, but Fmax drops 15% on average, due to higher resource utilization and more complex hardware architectures. On a per benchmark basis, Black-Scholes shows close to linear speedup in wall-clock time: $3.89\times$. Mandelbrot also shows linear speedup in clock cycles, but Fmax drops due to the use of 448 DSP blocks. Canny shows $3.74\times$ speedup in clock cycles, and $2.98\times$ speedup in wall-clock time. For k-means, the work load for each mapper/reducer, and thus the speedup from parallelization, is dependant on the initial coordinates of the centroids and the data points. We initialize each centroid to be at the centre of each quadrant of the entire x/y space, and randomly generate the initial data point coordinates. With this, the four mappers/reducers obtain $1.95\times$ speedup in clock cycles and $1.67\times$ in wall-clock time.

TABLE I: Performance and area results for *pipelined-only* benchmarks for LegUp HLS.

Benchmark	Time (μs)	Cycles	Fmax(MHz)	ALMs	Registers	DSPs	M20Ks
Mandelbrot	738.6	262208	355	1101	2746	112	0
Black-Scholes	16736.7	2560714	153	8575	28963	45	5
Canny	787.95	264752	336	1246	2415	0	10
K-means	70.4	20908	297	8499	20681	16	115
Geomean	910.01	246910.57	271.33	3162.11	7938.86	16.85	8.71

TABLE II: Performance and area results for *Canny* benchmark for a commercial HLS tool.

Benchmark	Time (μs)	Cycles	Fmax (MHz)	LUTs	Registers	DSP48s	BRAMs
Canny	792.64	264743	334	1427	1948	0	5
Ratio vs. Table I	1.006 (0.994×)	1.00 (1.00×)	0.99	1.15	0.81	1	0.5

In terms of area, the pipelined-and-replicated benchmarks show average increases of $2.86\times$, $2.75\times$, $7.45\times$, and $2.32\times$, in ALMs, registers, DSPs, and M20Ks, respectively. For DSP usage, all benchmarks increased linearly by a factor of four, with the exception of Canny. In the pipelined-only case, the compiler was able to optimize multiplications with constant filter coefficients, however this optimization did not occur in the replicated case, due to the structural code changes, utilizing 48 DSP blocks. For ALMs, the biggest relative increase was with Canny, which again, for the replicated scenario, the compiler was not to optimize the program as effectively as the pipelined-only, and we also had added additional logic and FIFOs to allow communication of the border rows. The smallest relative increase was with k-means, where most of the ALMs and M20Ks were used by eight dividers, used to average the x and y coordinates for the four centroids. Eight dividers were also needed in the pipelined-only case to meet II=1. In the pipelined-and-replicated case, each reducer handled one cluster, with two dividers each, thus the total number of dividers remained the same.

Overall, our methodology allows the synthesis of a diverse space of streaming hardware architectures that can be pipelined or pipelined *and* replicated, all from software. For massively parallel applications, replication of streaming hardware is as easy as forking multiple software threads. For the Canny benchmark, our streaming hardware showed very competitive results to that of a commercial tool. Our pipelined-only circuits provide high throughput, with an II=1, while the pipelined-and-replicated circuits further improve performance, at the expense of FPGA resources.

VI. Conclusion

In this paper, we proposed a methodology that allows standard software techniques to specify pipeline and spatial FPGA hardware parallelism. Our work allows software-threaded programs to model streaming hardware more accurately than the existing solution from Vivado HLS. The closer alignment between software and hardware allows a designer to better understand the generated hardware. It also enables more debugging to happen in software, which is much less difficult and time consuming than hardware debugging. Using Pthreads can open up many options, such as creating multiple streaming kernels that work concurrently. Our work also permits the creation of circuit architectures that are not feasible to realize in other HLS tools, such as a FIFO with multiple writers, where the arbitration is also automatically generated.

TABLE III: Performance and area results for *pipelined and replicated* benchmarks for LegUp HLS.

Benchmark	Time (μs)	Cycles	Fmax (MHz)	ALMs	Registers	DSPs	M20Ks
Mandelbrot	231.8	65606	283	4192	11006	448	0
Black-Scholes	4297	640252	149	19182	55843	180	20
Canny	264.8	70706	267	7396	14232	48	76
K-means	42.2	10712	254	11218	25919	64	120
Geomean	324.81	75102.66	231.25	9037.68	21820.8	125.46	20.25
Ratio vs. Table I	0.36 (2.80×)	0.30 (3.29×)	0.85	2.86	2.75	7.45	2.32

Our results showed we can generate pipelined circuits with high-throughput, competitive to that of a commercial HLS tool. The pipelined and replicated circuits, specified entirely from standard software, provide higher performance improvements. For future work, we would like to evaluate our HLS-generated hardware against human-designed hardware, to analyze performance and area, as well as identify areas for improvement.

References

[1] D. Hansen *et al.*, *Designing ASIC IP at Higher Level of Abstraction*, Calypto.

[2] J. Cong *et al.*, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Tran. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, April 2011.

[3] *Implementing FPGA Design with the OpenCL Standard*, Altera Corp., San Jose, CA, November 2013.

[4] L. Miller, *Adaptive Beamforming for Radar: Floating-Point QRD+WBS in an FPGA*, Xilinx Inc., San Jose, CA, June 24, 2014.

[5] R. Nane *et al.*, "A survey and evaluation of FPGA high-level synthesis tools," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. PP, no. 99, December 2015.

[6] *BDTI Certified Results for the AutoESL AutoPilot High-Level Synthesis Tool*, http://www.bdti.com/Resources/BenchmarkResults/HLSTCP/AutoPilot.

[7] *Xilinx: Xcell Journal, Issue 86*, http://www.xilinx.com/publications/archives/xcell/Xcell86.pdf, 2014.

[8] A. Canis *et al.*, "LegUp: high-level synthesis for FPGA-based processor/accelerator systems," in *ACM/SIGDA FPGA*, 2011, pp. 33–36.

[9] E. Anderson *et al.*, "Enabling a uniform programming model across the software/hardware boundary," in *IEEE FCCM*, April 2006, pp. 89–98.

[10] G. Stitt *et al.*, "Thread warping: a framework for dynamic synthesis of thread accelerators," in *IEEE/ACM CODES+ISSS*, 2007, pp. 93–98.

[11] A. Ismail *et al.*, "Fuse: Front-end user framework for o/s abstraction of hardware accelerators," in *IEEE FCCM*, May 2011, pp. 170–177.

[12] E. Lubbers and M. Platzner, "A portable abstraction layer for hardware threads," in *IEEE FPL*, Sept 2008, pp. 17–22.

[13] *OpenCL for Altera FPGAs*, http://www.altera.com/products/software/opencl/opencl-index.html, 2013.

[14] *Xilinx: Vivado Design Suite*, http://www.xilinx.com/products/design-tools/vivado.html.

[15] *Impulse CoDeveloper – Impulse accelerated technologies*, http://www.impulseaccelerated.com.

[16] *Xilinx: Vivado Design Suite User Guide - High-Level Synthesis*, http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/ug902-vivado-high-level-synthesis.pdf, November 2015.

[17] *Oracle: Multithreaded Programming Guide*, https://docs.oracle.com/cd/E19455-01/806-5257/sync-31/index.html.

[18] *POSIX.1 FAQ*, The Open Group, October 5, 2011.

[19] *Linux Programmer's Manual*, http://man7.org/linux/man-pages/man7/sem_overview.7.html.

[20] *A Semaphore Solution to the Producer-Consumer Problem*, http://www.csee.wvu.edu/jdm/classes/cs550/notes/tech/mutex/pcsem.html.

[21] B. Barney, *POSIX Threads Programming*, Lawrence Livermore National Laboratory, Auguest 13, 2015.

[22] J. Choi, S. Brown, and J. Anderson, "Resource and memory management techniques for performance and area of parallel hardware in high-level synthesis for fpgas," in *IEEE FPT*, December 2015.

[23] J. Choi *et al.*, "From software threads to parallel hardware in high-level synthesis for fpgas," in *IEEE FPT*, December 2013, pp. 270–277.

[24] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[25] *Mersenne Twister Home Page*, http://www.math.sci.hiroshima-u.ac.jp/ m-mat/MT/emt.html.

[26] *Monte Carlo Pricing of Asian Options on FPGAs Using OpenCL*, https://www.altera.com/support/support-resources/design-examples/design-software/opencl/black-scholes.html.

[27] J. Canny, "A computational approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679 – 698, November 1986.

[28] J. B. MacQueen, "Some methods for classification and analysis of multivariate observations," *Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability*, pp. 281 – 297, 1967.