Developing Dynamic Profiling and Debugging Support in OpenCL for FPGAs

Anshuman Verma
Virginia Tech, Blacksburg, VA
anshuman@vt.edu

Huiyang Zhou
North Carolina State University, Raleigh, NC
hzhou@ncsu.edu

Skip Booth, Robbie King, James Coole, Andy Keep, John Marshall
Cisco Systems, RTP, NC
{ebooth,robking,jcoole,akeep,jwm}@cisco.com

Wu-chun Feng
Virginia Tech, Blacksburg, VA
feng@cs.vt.edu

ABSTRACT

With FPGAs emerging as a promising accelerator for general purpose computing, there is a strong demand to make them accessible to software developers. Recent advances in OpenCL compilers for FPGAs pave the way for synthesizing FPGA hardware from OpenCL kernel code. To make this paradigm widely adopted, significant challenges remain to be overcome. This paper presents our efforts in developing dynamic profiling and debugging support in OpenCL for FPGAs. We first propose primitive code patterns, including a timestamp and an event-ordering function, and then develop a framework, which can be plugged easily into OpenCL kernels, to dynamically collect and process run-time information.

KEYWORDS

OpenCL; FPGA; Debugging; Profiling

ACM Reference format:
Anshuman Verma, Huiyang Zhou, Skip Booth, Robbie King, James Coole, Andy Keep, John Marshall, and Wu-chun Feng. 2016. Developing Dynamic Profiling and Debugging Support in OpenCL for FPGAs. In *Proceedings of Design Automation Conference, Austin, TX, USA, June 18-22, 2017 (DAC'17),* 6 pages.
DOI: http://dx.doi.org/10.1145/3061639.3062230

1 INTRODUCTION

Field programmable gate arrays (FPGAs) have been primarily used as an alternative to application-specific integrated circuits (ASICs). The recent trend of integrating FPGAs into server processors also make them a desirable reconfigurable accelerator for general purpose computations. The advances in high-level synthesis (HLS), especially OpenCL for FPGAs, open the door for software developers to make use of this type of substrate. A programmer specifies OpenCL kernel functions, from which the OpenCL-for-FPGA compiler synthesizes into FPGA bit-streams. It either leverages the explicit thread-level parallelism (TLP) or extracts the implicit loop-level parallelism (LLP) from kernel functions to achieve high throughput computation. Although highly promising, significant challenges remain for such OpenCL-for-FPGA design paradigms to be widely adopted, especially by software/system designers. One key challenge results from how differently the synthesized hardware operates from an apparently sequential processor. The synthesized hardware is fundamentally parallel and either the TLP or LLP is converted to pipeline-level parallelism. It is essential to provide software developers with facilities to see how operations are executed and reason about the execution results as well as the achieved performance. In this paper, we present our work on developing run-time dynamic profiling and debugging support in OpenCL for FPGAs.

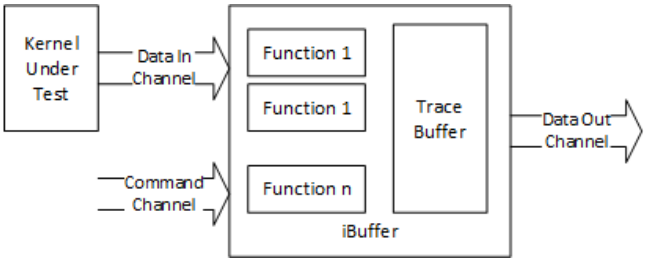

Figure 1: An overview of the iBuffer structure.

We leverage Altera AOCL channels [2] or OpenCL pipes to probe into the synthesized pipelines from kernel functions. Along with channels or pipes, we introduce two primitive patterns, one for timestamps and the other for sequence numbers, to gain information on hardware execution. We then propose an intelligent trace buffer, referred to as iBuffer, to collect and process the run-time information. The generic iBuffer framework is illustrated in Figure 1. As shown in the figure, an iBuffer contains both logic function blocks and a trace buffer. The logic function blocks provide data processing capabilities while the trace buffer serves as a flight recorder, storing the information from the data input channel(s). The command channel configures the state of the trace buffer and how the data are to be processed. The data output channel is used to forward the data in the iBuffer to the user for analysis.

We examine two use cases to show how our proposed iBuffer framework can be utilized. First, we use it to collect and analyze pipeline stalls to reason about the performance. Second, we construct watch points for selected memory locations to realize the

DAC'17, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/3061639.3062230

watch capability in common software debugging tools such as gdb. In the meanwhile, we also perform run-time address bound checking and/or value invariance checking for memory operations. Our experiments show that our proposed scheme works well in these use cases and incurs limited overhead in the FPGA area and clock frequency.

Existing works on source-level debugging for FPGAs [3] are mainly build upon logic analyzers such as SignalTap [1] by Altera, and ChipScope [9] by Xilinx. The logic analyzers are used to observe selected signal values, which are then stored in a trace buffer to record a window of execution. Compared to these works, our proposed scheme is built in OpenCL and does not depend upon any logic analyzer. More importantly, our software-centric approach enables intelligent data processing rather than merely recording the selected signals. To the best of our knowledge, our proposed scheme is the first open-source design for profiling and debugging entirely coded in high-level programming languages such as OpenCL.

The rest of the paper is organized as follows. Section 2 presents our experimental methodology. Section 3 provides the details of our primitive patterns, and two approaches are explored to implement the timestamp pattern. Section 4 discusses the design of the iBuffer framework. The two use cases of iBuffer are presented in Section 5. Section 6 discussed the related work. Section 7 concludes the paper.

2 EXPERIMENTAL METHODOLOGY

In this work, we use three different FPGA platforms to validate and evaluate our proposed framework. They include a discrete Stratix V FPGA, a discrete Arria 10 FPGA, and an integrated Arria 10 FPGA in an Intel Broadwell-EP processor. We use Altera SDK for OpenCL, 64-Bit Offline Compiler V16 to compile and synthesize circuitry for the FPGAs. We mainly report the results using the Stratix V system as other platforms show similar trends.

3 PRIMITIVE PATTERNS

3.1 Timestamp

Timestamps are useful to reveal run-time information on hardware execution. To enable the timestamp functionality, we propose the following two schemes. The first is to use a persistent autorun kernel, which contains a free-running counter, and feed the counter value through an Altera channel with depth 0, meaning that the channel always contains the most up-to-date counter value. The code of this persistent kernel is shown in Listing 1. As shown in the code, the counter value is written to the channel in a non-blocking manner, which will not affect the logic to increment the counter each cycle. To access the timestamp, a read channel function call is inserted in the kernel under test, as illustrated in Listing 2. In this example, there are two read sites of the timestamp and the difference between the two timestamps would show the latency of the event of interest, a vector dot-product in this case. Note, that since each channel can only support one producer and one consumer, multiple channels are used for multiple read sites.

The potential limitation of the persistent kernel based timestamp is two folds. First, the OpenCL compiler may try to optimize the channel depth although it is explicitly set to 0, which may result in stale timestamps. In our experiments, we found that we have to use one persistent kernel to drive one channel rather than letting multiple channels driven by the same free-running counter in a single persistent kernel. This may be a problem if different persistent kernels are not launched in the same cycle and there could be offsets among the separate free-running counters. Second, the read sites of timestamps have no data dependency upon any variables in the computation of interest. Therefore, the OpenCL compiler might move the read sites of the timestamps to optimize pipeline schedules, although such movement, i.e., moving the `read_channel` functions, has not been observed in our experiments.

```
channel int time_ch1 __attribute__((depth(0)));
__attribute__((autorun))
kernel void timer_srv(void) {
  int count = 0;
  while(1) {
    bool success;
    count++;
    success = write_channel_nb_altera(time_ch1,
        count);  } }
```

Listing 1: The timestamp pattern using a persistent kernel with a free-running counter.

```
int start_t, end_t;
//Read site 1
start_t = read_channel_altera(time_ch1);
//Event of interest
sum = 0;
for(int i = 0; i < num; i++) {
  sum += x[i+1]*y[i]; }
z[k] = sum;
//Read site 2
end_t = read_channel_altera(time_ch2);
```

Listing 2: Read site(s) of the timestamp.

```
In the file "timer.h"
ulong get_time(ulong command);
In the file "timer.cl"
ulong get_time(ulong command) {
    return (command + 1); }
In the file "timer.v"
module get_time (input clk, ..)
always @(posedge clk)
  if(~rstn) counter_time <= 'h0;
  else counter_time <= counter_time + 1;
...
```

Listing 3: The timestamp pattern using a Verilog module containing a free-running counter and its OpenCL interface.

```
int start_t, end_t;
start_t = get_time(sum); //read site1
//event of interest
sum = 0;
for(int i = 0; i < num; i++) {
  sum += x[i+1]*y[i]; }
z[k] = sum;
end_t = get_time(sum); //read site2
```

Listing 4: The read sites(s) of the timestamp using HDL counters.

```
channel int seq_ch __attribute__((depth(0)));
__attribute__((autorun))
kernel void seq_srv(void) {
  int count = 0;
  while(1) {
    count++;
    write_channel_altera(seq_ch,count);
  } }
```

Listing 5: Sequencing number: The persistent kernel containing a sequence counter.

Our second approach for timestamps aims to overcome the aforementioned limitations. Instead of using a persistent kernel containing a free-running counter, we resort to hardware-design language (HDL) to initiate such a counter and embed it into the kernels under test. Listing 3 and 4 shows the code of this approach and how it is used. As shown in Listing 3 , we define an OpenCL function `get_time`. The function defined in OpenCL has one input parameter, *command*, and simply returns the sum of (*command*+1) as its output. Such an OpenCL definition is used for emulation while the actual implementation for synthesis is defined in a Verilog module. All such information is encapsulated in a library to be integrated during the synthesis process.

Listing 4 shows the read sites of the timestamp in the same kernel as in Listing 2 . The variable *sum* is passed as the input parameter to our defined `get_time` function. The reason is to generate dependency so as to avoid the compiler accidentally moving the read sites during scheduling.

We analyze the area and frequency overhead of the two approaches to implement the timestamp pattern. The synthesis reports show that both approaches have very low area and frequency overhead. Between the two approaches, the HDL implementation has lower overhead in register usage and logic unit (1.1% logic overhead including a trace buffer) than the persistent kernel approach (1.3% logic overhead with the same trace buffer). The frequency of an un-profiled kernel, which performs intensive pointer-chasing operations, reaches 233.3MHz while the one adding the OpenCL free-running counters runs at 227.8MHz, and the one including the HDL counter runs at 229.2 MHz. As it does not use the channel, thereby free from the channel depth issue, the HDL approach is preferred to implement the timestamp pattern.

3.2 Sequence Number

Besides timestamps, a sequence number can be used to establish the order of run-time events. A sequence number can also be constructed using a persistent kernel, as shown in Listing 5. Similar to the code in Listing 1, a persistent autorun kernel is used to maintain the sequencing counter as shown in Listing 5. Rather than a free-running counter for timestamps in Listing 1, the sequencing counter will not be incremented until the blocking channel write function is finished. In other words, only after the consumer reads out the counter value from the channel, the counter is incremented.

Listing 6 and Listing 7 show how we use the sequence number to reveal the execution/scheduling order of the loops/workitems in a matrix-vector-multiplication kernel. The Altera OpenCL for FPGA compiler supports both single-task and NDRange kernels. In the single-task mode, the matrix-vector-multiplication is implemented as a nested loop as shown in Listing 6 . In the NDRange mode, each workitem computes one dot-product, as shown in Listing 7. The compiler extracts the loop-level parallelism in the single-task kernel while relies on the explicit thread-level parallelism in the NDRange kernel. Here, we use the sequence number to reveal how the synthesized hardware executes/schedules such loops/-workitems. The sequence number here is used as addresses for the profiling buffers. We also use the timestamp pattern to collect the clock cycle information, from which the execution order can also be constructed/confirmed. Since there is one read site and the data dependency due to the variable *seq* prevents compiler from moving the `read_channel function`, we use the OpenCL free-running counter for the timestamps. A segment of the collected results is shown in Figure 2, Figure 2(a) for the single task kernel in Listing 6 and Figure 2(b) for the NDRange kernel in Listing 7.

```
for(int k = 0; k < N; k++) { //N=50
  l = k*num; sum = 0;  //num = 100
  for(int i = 0; i < num; i++) {
    sum += x[i+l]*y[i];
    if (i < 10) {
      int seq = read_channel_altera(seq_ch);
      info1[seq] =
        read_channel_altera(time_ch[0]);
      info2[seq]  = k;
      info3[seq]  = i;
    }  }
  z[k] = sum; }
```

Listing 6: Sequencing number: the read site of the sequence number in a single-task kernel.

```
k = get_global_id(0); //50 workitems
l = k*num; sum = 0;  //num=100
for(int i = 0; i < num; i++) {
  sum += x[i+l]*y[i];
  if (i < 10) {
    int seq = read_channel_altera(seq_ch);
    info1[seq] =
      read_channel_altera(time_ch[0]);
    info2[seq]  = k;
    info3[seq]  = i;
  } }
```

Listing 7: Sequencing number: The read site of the sequence counter in an NDRange kernel.

From Figure 2, we can see that the single-task kernel in Listing 6 and the NDRange kernel in Listing 7 result in different execution orders. For the single-task kernel, all iterations in the inner loop are executed first before going to the next iteration of the outer loop, the same as sequential execution. For the NDRange kernel, however, different workitems (equivalent to outer loop iterations) get into the pipeline before they go to the next iteration of the (inner) loop. Such different execution orders lead to different memory access patterns. In the case of the single task kernel in Listing 6, the memory access order of the array x is x[0], x[1], x[2], ..., for the first iteration of the outer $for(k)$ loop and x[100], x[101], x[102], ..., for the second iteration, etc. In contrast, the access order for array x in the NDRange kernel in Listing 7 becomes x[0], x[100], x[200],

```
            Timestamp     k     i
info_seq[51]: 260911      5     0
info_seq[52]: 261002      5     1
info_seq[53]: 261003      5     2
info_seq[54]: 261004      5     3
```

(a)

```
            Timestamp     k     i
info_seq[51]: 289634      0     1
info_seq[52]: 289635      1     1
info_seq[53]: 289636      2     1
info_seq[54]: 289637      3     1
```

(b)

Figure 2: The execution/scheduling order of the loop iterations or workitems, (a) for Listing 6 and (b) for Listing 7.

..., for the first iteration of the $for(i)$ loop, and then x[1], x[101], x[201], ..., for the second iteration. Such different memory access patterns contribute to the different execution times (as revealed in the timestamps in Figure 2) of the two kernels.

4 FRAMEWORK

As shown in Figure 1, our proposed iBuffer contains logic functions and a trace buffer. The logic functions control how the incoming data are processed while the trace buffer stores the pertinent information. The command channel configures the state of the iBuffer to process the incoming data on a data channel. The output channel is used to forward the data stored in the iBuffer to the user. An iBuffer can be in one of the following states, `reset, sample, stop`, and `read`. The transitions among the states are presented in Figure 3. A state transition occurs either when there is control information coming through the command channel, or when an event completes in the state machine. Data are written into the trace buffer during the `sample` state, using one of the two ways, linear or cyclic. In the linear scheme, writes to the trace buffer stop when it is full, while in a cyclic scheme, writes continue until a stop command is issued through command channel. A read command through the command channel moves the state to `read`, during which the data are sent on the output channel. The state moves to `stop` when all the data in the trace buffer are read. Sampling is restarted by resetting the state machine.

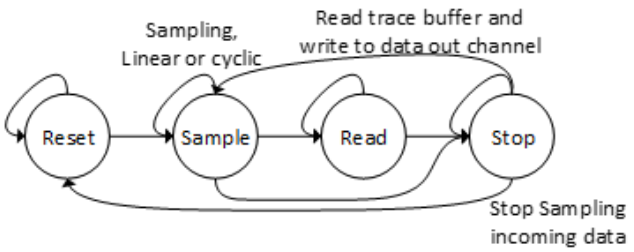

Figure 3: The state machine for an iBuffer. States can be changed by writing commands into the command channel, or by completion of tasks in the iBuffer.

As an iBuffer is used for dynamic debugging or profiling, it must not affect the designs under test. This imposes the following challenges. First, writes to the input data channel of the iBuffer should not block the calling site. Second, reads from and writes to the iBuffer should not affect global memory accesses latency of the kernel under test, i.e., debugging or profiling shall not alter the memory behavior of the design under test. The first challenge is addressed with a stall-free pipeline design as shown in Listing 8. The outer loop is an infinite loop and one iteration is launched every cycle. Single-cycle launch of each iteration of the outer loop ensures that incoming data is read from the input data channel each cycle and processed without incurring any data loss or stalls in the caller-site pipeline. We ensure single-cycle launch by avoiding any dependence on the outer loop variable and by either unrolling all the inner loops or avoiding inner loops. The launching schedule of the outer loop is confirmed with the compiler-generated log that provides details about compiled kernels. The second challenge is addressed by having a trace-buffer in local memory, hence writes to this memory do not affect global memory accesses. Users, however, need to ensure that the trace buffer is read out and written to global memory when the kernel under test is not running.

```
__attribute__((num_compute_units(N,1)))
__kernel void state_machine(void) {
  bool r, rvalid;
  state_e curr_state, next_state;
  uchar id  = get_compute_id(0);
  #pragma acc kernels loop independent
  for(ulong i = 0; i < ULONG_MAX ; i++) {
    take_stamp =
      read_channel_nb_altera(data_in[id], &r);
    next_state =
      read_channel_nb_altera(cmd_c[id], &rvalid);
    if(rvalid) {
      switch(next_state) {
        case RESET: ...
        case STOP: ...
        case SAMPLE: ...
      }   }   }
```

Listing 8: Autorun iBuffer persistent kernel.

An iBuffer can store information from a specific calling site where it writes data into the input data channel of the iBuffer. Users may want to probe into multiple kernels or have multiple calling sites inside a kernel. This requires multiple iBuffer instances, which are accomplished using the Altera attribute `num_compute_units (x,y,z)`. This attribute replicates the kernel in up to three dimensions, where the number of units in each dimension is specified by the variable x, y, and z. The depth (or size) of an iBuffer can be controlled by changing the define *DEPTH* as in Listing 10. This makes iBuffer scalable, for both the depth of the trace buffer and the number of instances, while each instance can be controlled by a separate command channel.

5 USE CASES

In this section, we present two use cases of our proposed iBuffer. The first collects the latency to show how a pipeline is stalled. The second provides smart watchpoints, which can perform address bound checking and/or value invariance checking on the fly for specified memory locations.

```
void take_snapshot(uint id, int in) {
    (void) write_channel_nb_altera(data_in[id],
                                   (int) in);
    mem_fence(CLK_CHANNEL_MEM_FENCE);}
.......
for(int k = 0; k < col_a; k++) {
  take_snapshot(0,k); //snapshot site 1
  int a = data_a[i*col_a + k];
  take_snapshot(1,a); //snapshot site 2
  int b = data_b[k*col_b + j];
  int acc  += a*b;
  ..... }
```

Listing 9: Measuring load latency using stall monitor.

5.1 Pipeline Stall Monitors

Pipeline stalls may occur because of loads or stores accessing global memory, or a throughput difference between a producer and a consumer connected through a channel. A pipeline stall monitor is useful to profile the kernel in such scenarios. By using the HDL-based timestamps and iBuffer framework, we develop a stall monitor as shown in Figure 4 that stores timestamps at points of interest in a kernel. A timestamp is taken inside the iBuffer when there is data available to be read at the data input channel. Such information is then written into the trace buffer in either a cyclic or linear fashion. Listing 9 shows an example of measuring the load latency of $a = data\_a[i * col\_a + k]$ in a matrix multiply kernel using the function `take_snapshot`. This example is very similar to Listing 4 while the function `take_snapshot` sends in the variable `in` through the data input channel and the timestamp is taken implicitly inside the iBuffer when the data input channel is read. As the iBuffer is stall free, the latency of the load can be computed as the difference between the two snapshots and the processed trace contains the latency of the load in an execution window determined by the trace buffer depth.

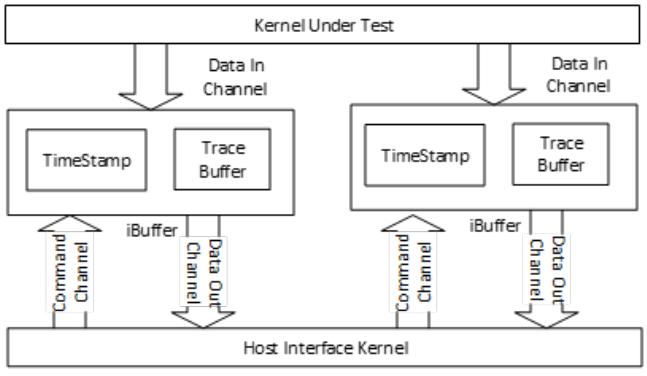

Figure 4: Monitoring pipeline-stalls using timestamps and the iBuffer.

To facilitate the host to communicate with our proposed iBuffer so as to initiate monitoring and collect the monitored results, a host interface kernel is introduced as shown in Figure 4. The implementation of this host interface kernel is shown in Listing 10. It works as an agent to forward the command from the host to the iBuffer through the command channel. When the command is a `read`, it then reads the data out channel until all the elements in the trace

```
#define N 10 //iBuffer Count
#define DEPTH 1024 //Trace buffer depth
channel cmd_e cmd_c[N]; //Command channels
channel out_e out_c[N]; //Data channels
kernel
void read_host (cmd_e cmd, int id, out_e out) {
  #pragma unroll
  for(int i = 0; i < N; i++) {
    if(i == id)
        write_channel_altera(cmd_c[i], cmd);}
 if(cmd == READ) {
   for(int k = 0; k < DEPTH; k++) {
     #pragma unroll
     for(int i = 0; i < N; i++) {
       if(i == id) {
         output[k] =
         read_channel_altera(out_c[id]);
       }  }  }  }  }
```

Listing 10: Host interface kernel to forward control commands from host to iBuffer and to read data from iBuffer.

buffer are read. This data is written to global memory, which can be accessed by the host for further post processing.

5.2 Smart Watchpoints

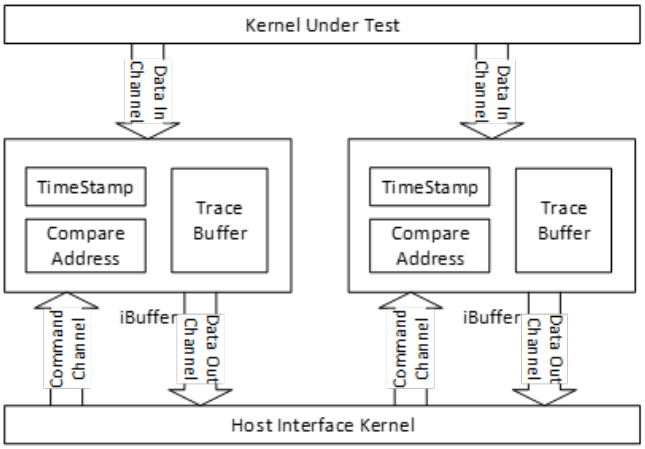

Figure 5: Smart watchpoints using timestamp and the iBuffer.

A watchpoint monitors how the value at a user-specified location in memory changes over time. As proposed in a prior work [11], additional functionality such as invariance checking or address bound checking can be included to make watchpoints more intelligent. Our design supporting such intelligent watchpoints uses the timestamp function and iBuffer framework, with augmented logic for address monitoring and/or value checking as shown in Figure 5. In our design, after setting up the watchpoints, we monitor the memory accesses of interest explicitly. In other words, a user needs to explicitly insert a `monitor_address` function for every possible memory operation that may access the location under watch. Such an example is shown in Listing 11. The address to be watched is provided using an additional channel `addr_in_c`. The watchpoint logic compares this address against the addresses sent on the data input

```
void add_watch(uint id, size_t address) {
  write_channel_nb_altera(addr_in_c[id], address);}
void monitor_address(uint id, size_t addr,
                     ushort tag) {
  in.addr = addr; in.tag = tag;
  write_channel_nb_altera(data_in[id], in);}
.....
add_watch(0,(size_t) &data_a[0]); //Add watch point
for(int k = 0; k < M; k++) {
  b = ...; ...
  a     = addr_a[k];
  //Monitor the read address for bound checking
  monitor_address(0,(size_t) &addr_a[k], a);
  *a    = b;
  //Monitor the write address for
  //      bound checking and value updates
  monitor_address(1,(size_t)a, b);
  ...} ...
```

Listing 11: Adding a watchpoint for a specified address and monitoring memory operations.

channel through the `monitor_address` functions. On a match, the tag specifying the value at the memory location and the timestamp are written into the trace buffer, if the iBuffer is in the `sample` state. The additional checking functions such as address bound checking or value invariance are supported by simply changing the code of iBuffer, where the information is read from the data input `data_in` channels. The user can control the iBuffer through a host interface kernel similar to Listing 10.

5.3 Evaluation

In our experiments, we implemented our proposed profiling and debugging support on different kernels. Table 1 lists logic and memory usage for generated designs upon a matrix multiplication kernel. The base shows the utilization for the kernel without any debugging/profiling support. When the same kernel is profiled using a stall monitor (SM), it results in small memory overhead and similar logic utilization while the clock frequency is reduced by 20.5%. We observed the similar results with a single watchpoint (WP) and a combination of a watchpoint and a stall-monitor. On a different kernel which performs pointer-chasing operations, however, the frequency overhead is less than 3%, as discussed in Section 3.1. Therefore, it seems that the overhead is kernel dependent. If the kernel is simple as matrix multiplication which can reach high baseline frequency, the overhead is more evident. Also, as we can see from Table 1, the design with a stall monitor has lower logic utilization than the baseline, which implies that the baseline matrix multiplication kernel may benefit from some synthesis optimizations, which trade logic for higher frequency, while the kernels with debugging/profiling support do not.

6 RELATED WORK

A wide-variety of work has been done on debugging circuits and interfaces. Bart et al. [8] use trace-buffers for creating an on-chip debugging infrastructure. Efficient trace-buffer structures and ways to improve the information storing in the buffers are proposed by Goeders and Wilson[4][5]. Hung [6] develop mechanisms to predict what signals will be useful during debugging. Similar to the work by Calagar et al. [3], these mainly utilize logic analyzers such as SignalTap to collect signals.

Table 1: The Logic and Memory Usage and Frequency Results.

Type	Clock Freq. (Mhz)	Logic Utilization	Memory Bit	Memory Blocks
Base	245.32	42.02K	2.97M	396
SM	194.96	41.92K	4.16M	414
WP	198.53	42.57K	4.03M	407
SM + WP	198.09	45.84K	4.16M	416

To verify the design generated by high-level synthesis tools, Yang et al. [10] develop a framework that uses just-in-time traces and inserts debugging logic into tool-generated RTL code. To address integration of custom logic with HLS tool-generated designs, Monson et al. [7] develop tools to map the user instructions and memory addresses to signals in designing and creating a "software-like" debugging interface.

None of these works address the OpenCL for FPGA design paradigm. Altera [2] provides profiling support for OpenCL for FPGA designs, which is inserted into the generated logic during synthesis and provides information on accumulated bandwidth and channel stalls. In comparison, our proposed framework provides detailed insight into synthesized designs and supports smart debugging functions.

7 CONCLUSIONS

In this work, we present a source-level scalable framework for debugging and profiling OpenCL for FPGA designs. It includes efficient timestamps and sequencer primitives and an intelligent trace buffer that can process data besides recording them on the fly. Our experiments show that the proposed framework has a small overhead on area and frequency, and provides a software-like interface to enable users to profile and debug the kernel functions effectively. The source code of the work is available at github.com/huiyzhou/dac-17-ocl-fpga-profiling

ACKNOWLEDGEMENT

The research is a result of a faculty visit to Cisco Systems and is also partially supported by the National Science Foundation (NSF) (under Grants No. 1216569, 1618509).

REFERENCES

[1] Altera. *Design Debugging Using the SignalTap II Logic Analyzer*, 2013.
[2] Altera. *Altera SDK for OpenCL: Best Practices Guide*, 2015.
[3] Calagar, N., et al. Source-level debugging for fpga high-level syntheis. In *FPL'14*.
[4] Goeders, J., and Wilton, S. J. Effective fpga debug for high-level synthesis generated circuits. In *FPL'2014*.
[5] Goeders, J., and Wilton, S. J. Using dynamic signal-tracing to debug compiler-optimized hls circuits on fpgas. In *FCCM'2015*.
[6] Hung, E., and Wilton, S. Speculative debug insertion for fpgas. In *FPL'2011*.
[7] Monson, J., and Hutchings, B. Using source-level transformations to improve high-level synthesis debug and validation on fpgas. In *FPGA 2015*.
[8] Vermeulen, B., and Goel, S. K. Design for debug: Catching design errors in digital chips. *IEEE Design & Test 19*, 3 (2002), 37–45.
[9] Xilinx. *ChipScope Pro Software and Cores: User Guide*, Apr 2012.
[10] Yang, L., et al. Jit trace-based verification for high-level synthesis.
[11] Zhou, P., et al. iwatcher: efficient architectural support for software debugging. In *ISCA'2004*.