ProfCounter: Line-Level Cycle Counter for Xilinx OpenCL High-Level Synthesis

André Bannwart Perina and Jürgen Becker
Institute for Information Processing Technologies
Karlsruhe Institute of Technology
Karlsruhe, Germany
Emails: abperina@usp.br, juergen.becker@kit.edu

Vanderlei Bonato
Institute of Mathematical Sciences and Computing
University of São Paulo
São Carlos, Brazil
Email: vbonato@usp.br

***Abstract*—Wide adoption of Field-Programmable Gate Arrays for compute-intensive problems has always been barred by the complex development flow, requiring hardware expert staff and extensive exploration effort. With the recent developments in High-Level Synthesis, it is possible to use high-level languages (e.g. C, C++, OpenCL) to produce hardware projects with reasonable results. However, several software development assist tools are still absent or provided with limited functionalities. In this paper ProfCounter is presented, consisting of a module capable of providing fine-grain profiling by measuring the latency of inner segments of OpenCL kernels. Thus, developers are able to perform analyses of specific parts of the code which can assist on code optimisations, by considering complex interactions between the kernel and other modules (e.g. DDR memory, cache) where cycle-accurate simulators are not able to properly model. Results show that ProfCounter is able to measure the cycle count of OpenCL kernels segments in the Xilinx environment and its overhead does not scale according to hardware complexity nor with the amount of measurements.**

I. Introduction

Improving energy efficiency of computing systems has become a widely investigated topic in the last years. The interest on architectures orthogonal to general purpose processors, such as Graphics Processing Units (GPU) and Field-Programmable Gate Arrays (FPGAs) increased. The latter, however, still suffers from a programmability barrier, requiring hardware expertise and a lot of effort on programming and debugging, coupled with a time-consuming compilation process (or synthesis). To ease the programmability burden, High-Level Synthesis (HLS) tools convert from high-level software codes (e.g. C/C++/OpenCL) to hardware descriptions. Examples of commercial HLS tools include the Xilinx Vivado HLS and Intel FPGA SDK for OpenCL.

OpenCL is a language designed to target several types of accelerators with a unified frontend and vendor-specific backends, including FPGAs with HLS. Several studies show that OpenCL HLS provides reasonable results in performance and/or energy efficiency when effort is given onto optimising the code for hardware generation [1][2][3]. An OpenCL system is divided in a host system (which manages the accelerators) and accelerator devices that executes kernels (which are functions written in a language similar to C).

Some profiling techniques widely available for software are still absent or implemented with limited functionalities in the HLS field, such as fine grain profiling. In HLS using OpenCL, profiling is only supported at a kernel level, thus it is not trivial to access performance details from specific intra-kernel regions. In these cases cycle-accurate simulators that avoid the time-consuming synthesis are of interest, however these simulators often do not consider external modules that can affect performance, such as off-chip memories or peripherals.

In this paper we present ProfCounter[1], which is a framework to provide fine-grain timestamping to OpenCL kernels targetting Xilinx platforms. With ProfCounter, developers are able to measure the latency of certain code segments by wrapping them with timestamping commands, in a similar fashion as performed in C with `gettimeofday()` calls.

The rest of the paper is structured as follows: Section II presents the ProfCounter framework, with validation analyses presented in Section III. Section IV presents and discusses the results, Section V the related works, with the concluding remarks on Section VI.

II. The ProfCounter Framework

The ProfCounter framework is presented in Fig. 1. It consists basically of an OpenCL kernel and a header file with profiling functions. It is currently compatible with the Xilinx HLS tools for OpenCL, namely SDx (SDSoC and SDAccel). When a timestamp command is sent from the Kernel Under Test (KUT) to `profCounter` via an OpenCL pipe, the current cycle count is saved to a buffer, which is flushed to the global memory readable by the host only after the KUT is finished (thus avoiding any competition for the global memory's bandwidth). ProfCounter enables accurate cycle count at line-level of OpenCL kernels with little modification in the KUT. Listing 1 presents an example of KUT (a vector-add) and the `profCounter` kernel logic.

Currently SDx does not support non-blocking `read_pipe()` calls, thus the `cycle` variable in Listing 1 does not increment accordingly with the clock cycling since it is blocked by the pipe. As a solution, the real `profCounter` kernel follows the same logic, but implemented as a Verilog HDL module with non-blocking pipes and wrapped as an

[1]ProfCounter is available at https://github.com/comododragon/sdx-ocl-profcounter

©2019 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

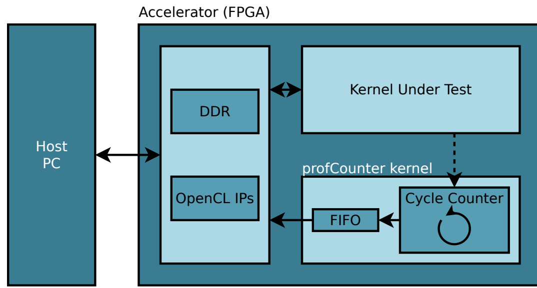

Fig. 1. The OpenCL framework with ProfCounter. The dotted line represent an OpenCL pipe.

```
kernel void vadd(...) {
  PROFCOUNTER_INIT();
  PROFCOUNTER_STAMP(); // write_pipe()
  for(int i = 0; i < size; i++) C[i] = A[i] + B[i];
  PROFCOUNTER_STAMP(); // write_pipe()
  PROFCOUNTER_FINISH();
}

kernel void profCounter(global long *log) {
  long offset = 0, cycle = 0; unsigned comm = COMM_NOP;
  for(cycle = 0; comm != COMM_FINISH; cycle++) {
    read_pipe(p0, &comm);
    if(COMM_STAMP == comm) log[offset++] = cycle;
  }
}
```

Listing 1. Example of ProfCounter usage: a KUT with timestamping commands and a simplified version of the profCounter kernel (the timestamp buffer and global memory flush were omitted for simplicity).

OpenCL kernel using the "RTL kernel" capability of Xilinx SDx.

The timestamp commands can be placed in several places of the KUT, including loops and conditionals. ProfCounter also provides identified timestamps (`PROFCOUNTER_CHECKPOINT_x()`, `x` is a numerical ID) that can be used when the KUT has non-trivial flow behaviour that makes it difficult to pinpoint which stamp command generated which timestamp.

III. Validation Analyses

To validate ProfCounter as a measuring tool, we analysed non-intrusiveness, accuracy and overhead. We collected several OpenCL benchmarks with the purpose of validating ProfCounter.

Nine OpenCL kernels were adapted from C benchmarks from [4] which are based on the PolyBench benchmark [5]: `atax`, `bicg`, `conv2d`, `conv3d`, `gemm`, `gesummv`, `mvt`, `syr2k` and `syrk`. These are simple C functions, adapted to OpenCL simply by wrapping the functions as OpenCL kernels. These benchmarks have a static predictable flow, thus Vivado can accurately estimate the cycle count. We also adapted the Breadth-First Search (BFS) algorithm from Rodinia [6], transforming it into a task kernel (i.e. only one processing element, though parallelism is still explored by the HLS).

A. Non-intrusiveness

As any measuring tool, the measurement overhead must be kept to a minimum and the tool inclusion must not alter the latency of the generated hardware. For this purpose, we compared the cycle counts reported by Vivado with and without the presence of ProfCounter.

For the `bicg` kernel it was detected that the presence of stamping commands caused alterations in the Control-Flow Graph (CFG) of the intermediate representation used as base for HLS, altering the cycle count. More specifically, the presence of `write_pipe()` of the stamping commands triggered CFG alterations following (after) the command. To overcome this issue, the stamping commands were modified to generate a placeholder code instead of the actual OpenCL pipe calls. The placeholder code is then replaced by the actual pipe write after CFG generation/optimisation and before HLS scheduling/binding. To avoid the OpenCL pipe to be optimised away by the lack of pipe calls during optimisation, only the finish command uses `write_pipe()`. Since this should be the last instruction in the KUT, no CFG alteration was detected with this approach.

We also noticed that for `bfs`, Vivado automatically pipelined loops in the KUT when the ProfCounter was present. Without ProfCounter, even when explicit commands to pipeline these loops was passed to Vivado, the HLS was not able to perform pipelining. As a current limitation, we only consider KUTs where loop pipelining is explicitly disabled.

B. Accuracy and Overhead

It is important also to analyse the accuracy of the measurements, so as the overhead created by inserting the measurement tool. We analysed these points by synthesising the aforementioned kernels for a Xilinx Zynq UltraScale+ ZCU102 development kit using Xilinx SDSoC version 2018.2 toolset. The platform features a Zynq SoC composed of a Processor Subsystem (PS) and a Programmable Logic (PL). The PS is used as the OpenCL host controller, while the PL is the FPGA accelerator.

To validate the accuracy, we executed the kernels and compared the values reported by ProfCounter with the cycle reports from Vivado. The data movement between the host (PS) and accelerator (PL) occurs via global memory (DDR), thus to compare the latency of the computation part only, the data must first be transferred to local buffers (BRAM), processed and then brought back to the global memory. For the PolyBench-based kernels, we performed such data movement and created two profiling regions: one considering only the processing part and another also considering the global/local data transfers.

Since `bfs` has irregular accesses to global memory and loops with dynamic bounds, Vivado could only estimate the latency for the inner loop in the form of an interval. In this case we used ProfCounter to measure the latency of the inner loop and also the whole computation part. We did not adapt `bfs` to use local buffers, thus the computation part makes direct accesses to global memory.

To analyse the overhead, we synthesised all kernels with and without ProfCounter and calculated the final resource difference. We also synthesised all kernels with loop unrolling

TABLE I
COMPUTATION CYCLE COUNT FOR THE BENCHMARKS

KUT	Unroll disabled Vivado	Unroll disabled ProfCounter	Unroll enabled Vivado	Unroll enabled Profcounter
atax	147712	147713	66304	66369
bicg	656384	656386	263936	264066
conv2d	603540	603541	38052	38053
conv3d	1675860	1675861	156660	156661
gemm	25280768	25280769	8495744	8495745
gesummv	164608	164609	66816	66881
mvt	655872	655873	263680	263809
syr2k	29393152	29393153	16883968	16883969
syrk	27377920	27377921	8495744	8495745
bfs	—	49707181	—	49230850

enabled to increase resource usage and to evaluate if ProfCounter's overhead scales with the complexity of the kernel.

IV. EXPERIMENTAL RESULTS

In this section, we present the profiling results for the benchmarks, exposing ProfCounter's accuracy and its resource overhead in the Zynq UltraScale+ platform. All tests were executed 100 times as other processes in the PS may slightly affect the host software total execution time. In all cases, the KUT latencies reported by Vivado were the same regardless of the inclusion of ProfCounter, guaranteeing its non-intrusiveness.

A. Accuracy Analysis

Table I presents the cycle count for the computation part of the OpenCL kernels from ProfCounter and from Vivado latency analysis, ignoring data transfers when applicable.

With loop unrolling disabled, the cycle count given by Vivado deviated by 1 cycle for almost all kernels and by 2 cycles for `bicg`. We found out by analysing the scheduling reports that Vivado does not consider the exit test condition in their loop latency calculation (1 cycle for each loop) while ProfCounter does. In the `bicg` case the computation part is composed by two consecutive loops, which explains the deviation of 2 cycles.

When loop unrolling was enabled, the same deviation occured for all kernels where Vivado was able to infer the cycle count. However, a bigger deviation was detected for some kernels as pointed in the table. By analysing the generated FSM of those cases, we noticed that Vivado added some states before the loop header to load the variables used in the unrolled loop, as exemplified in Fig. 2. Vivado does not consider these load states in their estimations, while ProfCounter successfully account these states as part of the loop, as the amount of loads is proportional to the loop size and unroll factor.

For the PolyBench-based benchmarks where Vivado could report the total kernel cycle count, the difference reported by Vivado and ProfCounter was a constant 132 cycles. We believe that this deviation is related to OpenCL kernel initialisation.

For `bfs`, Vivado was only able to estimate the innermost loop latency (from 274 to 406 cycles). With ProfCounter, we were able to count the cycles for the outermost loop, which varied depending on the input data. For the innermost

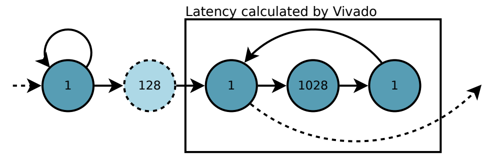

Fig. 2. Simplified FSM for the computation segment of `bicg`. The node numbers represent the latency of each node. The stamping commands are performed in the dotted edges, while the latency reported by Vivado does not consider the loads performed prior to the loop (represented as the dotted node) nor the exit condition tests for each loop.

TABLE II
RESOURCE OVERHEAD CAUSED BY PROFCOUNTER (IN %)

KUT	Unroll disabled LUTs	Unroll disabled FFs	Unroll disabled BRAMs	Unroll enabled LUTs	Unroll enabled FFs	Unroll enabled BRAMs
atax	57.53	78.74	36.59	36.54	43.51	35.71
bicg	49.42	72.00	11.54	12.65	14.55	11.36
conv2d	57.26	79.90	22.06	8.54	18.37	22.06
conv3d	54.09	79.16	11.90	22.91	42.94	11.90
gemm	55.51	74.09	15.46	14.88	18.15	15.46
gesummv	50.81	71.83	21.13	34.99	42.24	20.83
mvt	55.10	75.07	11.72	26.36	30.03	11.63
syr2k	54.57	73.27	15.46	28.65	30.39	15.46
syrk	56.78	77.92	22.06	17.38	20.56	22.06
bfs	55.92	78.41	150.00	29.16	45.26	150.00
Average	54.70	76.04	31.79	23.21	30.60	31.65

loop, ProfCounter reported 1 and 407 cycles for two different iterations considering a given input data. Thus, in one iteration the loop did not execute (1 cycle for loop exit test), whilst in the other case the loop executed only one iteration with the worst-case latency informed by Vivado. The iteration latency is in accordance with the estimation from Vivado, however ProfCounter was able to detect more detailed information about the execution of `bfs`.

B. Scalability Analysis

According to the reports before place and route, the ProfCounter module (isolated from any logic automatically added by SDx) uses 325 LUTs, 493 flip-flops and 1 BRAM unit.

Table II presents the total overhead caused by the insertion of ProfCounter in a project, calculated with the resource count after place and route.

Even though the overhead seems expressive, the benchmarks here presented are very simple. In all cases, the overhead caused by ProfCounter accounts for less than 1% of the ZCU102 resources. Considering all kernels, adding ProfCounter caused an average increase of 2625.85 ± 71.46 LUTs, 4711.20 ± 49.44 flip-flops and 7.50 ± 0.00 BRAM units.

As an example, the final count for the non-unrolled `atax` increased from 4535, 5983 and 20.5 to 7144, 10694 and 28 for LUTs, flip-flops and BRAMs respectively when ProfCounter was added. However, the average overhead decreased when the complexity of the kernels increased. The average overhead dropped from 54.70% and 76.04% to 23.21% and 30.60% for LUTs and flip-flops when unrolling was enabled. In the BRAM case, loop unrolling did not expressively increase BRAM usage.

We also varied the amount of stamping commands in the `bfs` kernel. There was no significant increase in the resources (increasing from 4 to 6 stamping commands increased the resource count in 11 LUTs and 1 flip-flop), indicating that ProfCounter does not scale according to hardware complexity and amount of stamping commands.

V. Related Works

A source-level debugger for LegUp is presented in [7]. It instruments the source code based on the HLS process and uses in-hardware probing to provide debug features. They use a custom probing module that uses internal information from the HLS process to reduce the resource overhead. On average, the in-hardware probe adds an overhead of 2500 logic elements on the Intel FPGA Cyclone II and does not scale with larger circuits.

HLScope [8] provides in-hardware probing through a module developed as a high-level code which can be integrated to the function under test. Their in-hardware probe uses approximately 170 LUTs and 1 BRAM unit, scaling if more signals are probed. However, it is not portable to Xilinx SDx due to current limitations (e.g. non-blocking read of pipes are needed).

The work of [9] provides an in-hardware profiling module for OpenCL HLS targetting Intel FPGA platforms. They can provide fine-grain profiling of OpenCL kernels, along with watchpoints for variables and addresses. This work is very close to ProfCounter, however they target Intel FPGA OpenCL. The logic utilization overhead ranges from 550 to 3820 logic elements and from 18 to 20 memory blocks depending on the type of profiling. It is not portable to Xilinx, as the Verilog version of cycle counter is implemented as a helper function callable inside OpenCL kernels, which is not currently supported. With the pure-OpenCL kernel, each timestamp point would require a separate kernel, and there is no trivial mechanism to start all at the same time.

Similar to [9], the work of [10] presents a monitoring hardware for OpenCL targetting Intel FPGA, where an RTL monitor is attached to the KUT using a streaming bus, providing similar timestamping capability to our work. However it is not directly portable to Xilinx, since they do not make use of OpenCL pipes and SDx does not support the inclusion of RTL as callable functions inside the KUT. The amount of ALUTs and flip-flop shows greater variation relative to the amount of stamping commands when compared to our work (e.g. the overhead for flip-flop increases from 21% to 31% for 2 and 8 stamping commands respectively). Adding two monitors to their test case increased the resource count in 281 ALUTs, 1103 flip-flops and 9 BRAM units.

None of these works can be used to profile OpenCL kernels for the Xilinx SDx as provided by ProfCounter. The in-hardware probes are not portable do SDx due to aforementioned limitations. Up to our knowledge, this is the first line-level profiler available for the OpenCL in Xilinx platforms.

It is not completely fair to compare resource overhead between different FPGA technologies, however results indicate that ProfCounter does not scale with more complex circuits nor with the amount of stamping commands.

VI. Concluding Remarks

This paper presented ProfCounter, a line-level profiler for OpenCL in the Xilinx HLS (SDx) environment. Although synthesis is required, ProfCounter is able to provide actual cycle latency of segments inside OpenCL kernels as defined by the developer with little code modification. According to experimental results, the amount of resources used by ProfCounter remains practically constant regardless of circuit complexity and amount of stamping commands. We believe that ProfCounter can be quite useful in cases where the developer wants actual cycle latencies of certain parts of a kernel where estimations may not accurately reflect the actual execution, such as when considering access to external memories. As future work, tests with NDRange (SIMD) kernels should be performed, along with the inclusion of pipelined kernels.

Acknowledgment

The authors would like to thank FAPESP (São Paulo Research Foundation, grants no. 2018/22289-6 and 2016/18937-7) for the financial support given to this research project.

References

[1] S. Tatsumi, M. Hariyama, M. Miura, K. Ito, and T. Aoki, "OpenCL-based design of an FPGA accelerator for phase-based correspondence matching," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2015, p. 90.

[2] F. B. Muslim, L. Ma, M. Roozmeh, and L. Lavagno, "Efficient FPGA implementation of OpenCL high-performance computing applications via high-level synthesis," *IEEE Access*, vol. 5, pp. 2747–2762, 2017.

[3] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka, "Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2016, p. 35.

[4] G. Zhong, A. Prakash, Y. Liang, T. Mitra, and S. Niar, "Lin-analyzer: a high-level performance analysis tool for FPGA-based accelerators," in *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 2016, p. 136.

[5] L.-N. Pouchet, "PolyBench: the polyhedral benchmark suite," 2012, available at https://www.cs.ucla.edu/pouchet/software/polybench, accessed 9th apr. 2019.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE, 2009, pp. 44–54.

[7] J. Goeders and S. J. Wilton, "Using dynamic signal-tracing to debug compiler-optimized HLS circuits on FPGAs," in *2015 IEEE 23rd annual international symposium on field-programmable custom computing machines*. IEEE, 2015, pp. 127–134.

[8] Y.-k. Choi and J. Cong, "HLScope: high-Level performance debugging for FPGA designs," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2017, pp. 125–128.

[9] A. Verma, H. Zhou, S. Booth, R. King, J. Coole, A. Keep, J. Marshall, and W.-c. Feng, "Developing dynamic profiling and debugging support in OpenCL for FPGAs," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2017, pp. 1–6.

[10] H. Bensalem, Y. Blaquière, and Y. Savaria, "Toward in-system monitoring of OpenCL-based designs on FPGA," in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2019, pp. 1–5.