Porting Finite State Automata Traversal from GPU to FPGA: Exploring the Implementation Space

Marziyeh Nourian, Mostafa Eghbali Zarch and Michela Becchi
North Carolina State University
{mnouria, meghbal, mbecchi}@ncsu.edu

1 Introduction

In order to achieve better performance and power efficiency, computing systems are increasingly becoming heterogeneous and leveraging many-core processors and reconfigurable accelerators along with general-purpose CPUs. There has been an increased interest in adding FPGAs to data centers and high-performance computing clusters. FPGAs are minimally adopted since they are hard to program and require digital design expertise. Thus, there has been a push towards increasing the programmability of these devices through the use of programming models – like OpenCL – intended for multi- and many-core architectures. For example, both Xilinx and Intel are providing their own OpenCL-to-FPGA development toolchain and runtime system [1, 2]. However, the direct porting of OpenCL code designed for GPU often leads to poor results both in terms of performance and logic utilization [3-5]. This is mainly due to differences in the underlying hardware and the nature of the offered parallelism (i.e., massive multithreading in GPU vs. pipelining in FPGA).

Meanwhile, there has been an increasing interest in accelerating finite state automata (FA), the core computational engine of pattern matching, used in variant fields such as biology and networking. This has led to many FA processing implementations targeting a variety of computing platforms [6-10]. Almost all FA processing engines for GPU have a memory-intensive design and require barrier synchronization. Since the off-chip memory bandwidth of OpenCL-enabled FPGA boards is significantly lower than that of similarly priced high-end GPUs, and since on FPGA synchronization barriers result in pipeline flushes, negatively affecting performance, those designs are not a good fit for FPGA. Our recently proposed SIMD_NFA engine [11] has characteristics that make it a better candidate for FPGA deployment. Here, we investigate the deployment of the OpenCL version of SIMD_NFA, on FPGA and propose optimization techniques to retarget SIMD_NFA to FPGA. We believe that some of these optimizations can be generalized to benefit other applications.

2 SIMD_NFA Processing Scheme

Most non-deterministic finite automata (NFA) traversal engines for GPU [6-8] store the NFA states and transitions in global memory and parallelize the traversal by distributing the set of active states and their transitions across the work-items. These implementations are typically irregular, and suffer from high memory bandwidth requirements, irregular memory accesses, control flow divergence, and synchronization overhead. SIMD_NFA [11] is a memory-efficient NFA processing scheme that addresses these issues for applications that rely on fixed-topology NFAs (i.e., NFAs with same topology and different symbols triggering the transitions). To this end, SIMD_NFA encodes the NFA topology in the traversal code, and stores in memory only the characters associated to the state transitions (in addition to the input streams). SIMD-NFA assigns to each work-item a distinct *batch* of 32 NFAs (i.e., patterns) and distributes the input streams across the work-groups. The work distribution and the memory layout are designed so as to minimize the control flow and memory divergence and avoid the need for invoking synchronization primitives.

Due to the lack of space, we represent the SIMD_NFA pseudo code in the poster draft. Here, we present an overview of SIMD_NFA operation. SIMD_NFA uses two bitmap arrays to record the states' activations before and after processing each input character, and stores them in local memory for fast access. These arrays are called *current* and *future state vectors* ($current_{sv}$ and $future_{sv}$), respectively. In these arrays states that have the same identifier (i.e., the same location in the NFA topology) from different NFAs are placed in contiguous regions of memory. By assigning to each work-item a distinct batch of NFAs, SIMD_NFA avoids the need for barrier synchronization and atomic operations when updating the state vectors. To allow for coalesced memory accesses, work-items of each work-group access subsequent memory words.

As a part of the main kernel, the input characters are sequentially fetched from global memory based on the input stream identifier and the activation of persistent states (i.e., states that, once activated, will remain active) is supports efficiently by a bitmap operation on the state vectors. The main kernel invokes *topology-specific-traversal* function that contains the operations necessary for updating $future_{sv}$ based on each transition of the topology. There are four transition types, each handled by a distinct bitmap operation, namely: positive, negative, wildcard and epsilon transitions. The *sv_update* function updates $future_{sv}$ through simple bitmap operations based on the transition type, the current value of the state vectors and a mask generated by the *match_check* function. Positive and negative transitions are triggered based on match and mismatch between the transition symbol(s) and the input character, respectively. The 32-bit mask (32 being the batch size) is used to indicate whether the input character matches the symbol(s) of the transition being processed for each NFA in the batch. For positive and negative transitions, the $future_{sv}$ is updated based on the value of the $current_{sv}$ and the mask. Since wildcard and epsilon transitions are input independent, they do not require the mask for updating the $future_{sv}$. Note that, the topology-specific-traversal function invoked by the main kernel is the only topology specific section of the code; the remaining code (including the match_check and sv_update functions) is common across NFA topologies.

3 Deploying SIMD_NFA to FPGA

OpenCL-to-FPGA support two execution models: *single work-item* and *NDRange* kernels. While SIMD_NFA can be deployed through both NDRange and single work-item execution models, due to the scarce space, we focus on NDRange form. The original SIMD_NFA code is in NDRange form and consists of fully independent work-items. By profiling the native SIMD_NFA code on FPGA, we have found two main sources of inefficiency: global memory access stalls and excessive logic utilization. This motivates us to explore a set of custom and best practice optimizations to retarget the code optimized for GPU to FPGA.

Structural code changes (struct) - By analyzing OpenCL-to-FPGA toolchain reports, it appears that the compiler breaks the code into blocks. Each module contains the logic necessary to implement the functionality of the corresponding code block, as well as the load units required by its memory operations. Although modules are not generated at the granularity of basic blocks, it looks like control flow instructions are used to break the code into code blocks for module generation. The topology-dependent section of OpenCL code contains a combination of function calls to *match_check* and *sv_update* to process specified NFA topology transition-by-transition. This "flat" structure has no additional control flow that leads to very large designs, none of them fitting the FPGA we used. This is likely due to the lack of branching instructions in the code, which results in a large module implementing all the state transitions, and in limited hardware reuse. Thus, we have introduced structural changes in the code without modifying its functionality. Specifically, we have incrementally added control-flow statements to facilitate the generation of smaller modules. First, we insert a loop that iterates over the states of the NFA. Then, for each NFA state, we introduce an if-block that encapsulates the logic related to the transitions outgoing from that state. The baseline code incorporates this change. Note that, when a code block is executed in multiple iterations of a loop, its logic is instantiated only once that allows logic reuse. We aim to generate modules with similar logic to allow for module reuse. Thus, we gradually increase the number of control flow statements so as to allow breaking the topology-specific code section into finer-grain code blocks. Therefore, we further increase the amount of branching by adding to the code an if-block for each outgoing transition from a given state, and a for-loop encapsulating these if-blocks. Finally, we do the same for characters triggering each transition.

Stream-level pipelining (SLP) - In the original implementation, the input streams are distributed over the work-groups, allowing coarse-grained stream-level parallelism. NDRange kernels allow work-group parallelism solely through pipeline replication, which is possible only if the resource utilization of a single execution pipeline allows it. Consequently, in most cases FPGAs are left to process input streams sequentially. Stream-level pipelining can be achieved by assigning multiple input streams to each work-group. Within each work-group, the input streams will be then distributed over the work-items, such that each work-item will process a given input stream against a batch of NFAs. By pipelining the execution of the work-items, the compiler can enable stream-level pipelining.

Changing degree of NFA-level parallelism (batch8) - In SIMD_NFA each work-item traverses a *batch* of 32 (corresponding to the size of a *warp*) NFAs, concurrently. On FPGA, batch size (B_{size}) selection is flexible and affects several aspects of the implementation. First, the chunks of the state vectors updated by each work-item are of B_{size} bits. Second, the transition characters' array in global memory is accessed in chunks of B_{size} bytes. Third, the mask generated by the *match_check* function consists of B_{size} bits. The loop that fills that mask has B_{size} iterations and is automatically unrolled by the compiler. Thus, the number of instructions of the *match_check* function depends on the B_{size}. Reducing the number of instructions can simplify the resulting logic and enhance the clock rate. Lastly, decreasing the B_{size} increases the number of work-items required to support the same number of NFAs. In experiments, we further test a batch size of 8.

Best practice guide optimizations - We used several techniques from Intel's SDK for FPGA Best Practice Guide [12] to reduce the logic utilization and improve the performance of the code. These include using constant and local memory to allow faster accesses to transition characters, using the *vector* data type to increase the memory bandwidth efficiency, enabling automatic SIMD vectorization of loops, and using pipeline replication (PR) when the resource utilization permits it.

4 Experimental Evaluation

We use an Intel DE5-Net board, which includes a Stratix V FPGA. We use Intel FPGA SDK for OpenCL Standard Edition v. 18.1 to compile and synthesize our OpenCL code. We perform our experiments using various NFA topologies from an open-source benchmark suite [13]. Our experimental evaluation covers traversal throughput and logic utilization (Fig. 1). In the graphs of Fig. 1, from left to right datasets are sorted in an ascending order based on NFA size. The optimizations are incrementally performed on the original flat code, from left to right, until constant memory usage. Since batch-8 significantly reduces the throughput, we implement each remaining optimization to the constant memory code-variant. Experiments are performed on 2 input streams. The data for designs that do not fit on the device are missing. From the resource utilization perspective, our evaluation shows that our optimizations not only allow the code to fit the reference FPGA, but also enable us to deploy up to 3 execution pipelines on it. From the performance perspective, our optimizations result in up to 4x speedups over the first optimized code-variant that fits each NFA topology on the FPGA device.

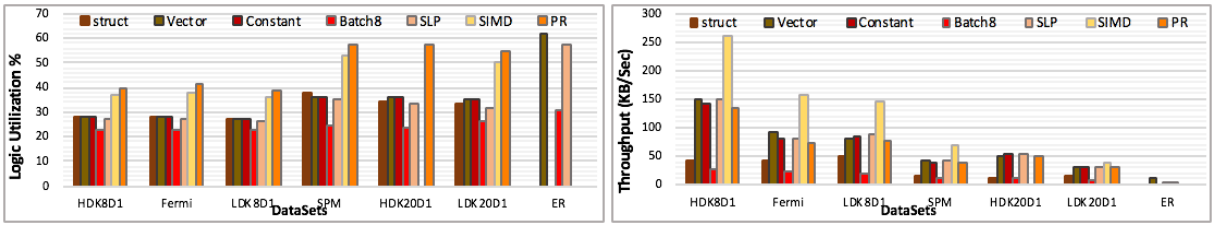

Fig. 1. Experimental results in terms of logic utilization (left) and traversal throughput (right).

REFERENCES

[1] "Intel FPGA SDK for OpenCL Software Technology," https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html.

[2] "SDAccel: Enabling Hardware-Accelerated Software," https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.

[3] H. R. Zohouri, N. Maruyama et al., "Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs," in Proc. of SC 2016.

[4] K. Krommydas, A. E. Helal et al., "Bridging the performance-programmability gap for fpgas via opencl: A case study with opendwarfs," in Proc. of FCCM 2016.

[5] M. W. Hassan, A. E. Helal et al., "Exploring FPGA-specific Optimizations for Irregular OpenCL Applications," in Proc. of ReConFig 2018.

[6] N. Cascarano, P. Rolando, F. Risso, and R. Sisto, "iNFAnt: NFA pattern matching on GPGPU devices," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 5, pp. 20-26, 2010.

[7] Y. Zu, M. Yang et al., "GPU-based NFA implementation for memory efficient high speed regular expression matching," in Proc. of PPoPP 2012.

[8] X. Yu, and M. Becchi, "GPU acceleration of regular expression matching for large datasets: exploring the implementation space," in Proc. of CF 2013.

[9] Y. Fang, T. T. Hoang et al., "Fast support for unstructured data processing: the unified automata processor," in Proc. of MICRO 2015.

[10] Y.-H. E. Yang, W. Jiang, and V. K. Prasanna, "Compact architecture for high-throughput regular expression matching on FPGA," in Proc. of ANCS 2008.

[11] M. Nourian, H. Wu, and M. Becchi, "A Compiler Framework for Fixed-topology Non-deterministic Finite Automata on SIMD Platforms," in Proc. of ICPADS 2018.

[12] Intel. "Introduction to Intel FPGA SDK for OpenCL Standard Edition Best Practices Guide," https://www.intel.com/content/www/us/en/programmable/documentation/rqk1517250959424.html.

[13] J. Wadden *et al.*, "ANMLzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures," presented at the Workload Characterization (IISWC), 2016 IEEE International Symposium on, 2016.