Optimizing Complex OpenCL Code for FPGA: A Case Study on Finite Automata Traversal

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

***Abstract*— While FPGAs have been traditionally considered hard to program, recently there have been efforts aimed to allow the use of high-level programming models and libraries intended for multi-core CPUs and GPUs to program FPGAs. For example, both Intel and Xilinx are now providing toolchains to deploy OpenCL code onto FPGA. However, because the nature of the parallelism offered by GPU and FPGA devices is fundamentally different, OpenCL code optimized for GPU can prove very inefficient on FPGA, in terms of both performance and hardware resource utilization.**

This paper explores this problem on finite automata traversal. In particular, we consider an OpenCL NFA traversal kernel optimized for GPU but exhibiting FPGA-friendly characteristics, namely: limited memory requirements, lack of synchronization, and SIMD execution. We explore a set of structural code changes, custom and best-practice optimizations to retarget this code to FPGA. We showcase the effect of these optimizations on an Intel Stratix V FPGA board using various NFA topologies from different application domains. Our evaluation shows that, while the resource requirements of the original code exceed the capacity of the FPGA in use, our optimizations lead to significant resource savings and allow the transformed code to fit the FPGA for all considered NFA topologies. In addition, our optimizations lead to speedups up to 4x over an already optimized code-variant aimed to fit the NFA traversal kernel on FPGA. Some of the proposed optimizations can be generalized for other applications and introduced in OpenCL-to-FPGA compiler.

Keywords—OpenCL, FPGA, high-level synthesis, automata processing, NFA, performance optimization

I. 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. This shift from homogeneous to heterogeneous computing has progressively occurred not only in single-machine systems, but also in large-scale computing clusters. GPUs have been part of high-performance computing clusters and cloud computing platforms for several years now, and today, many supercomputers in the Top500 and Green500 [1, 2] lists are equipped with GPU and Intel Phi boards. More recently, there has been an increased interest in adding FPGAs to data centers and high-performance computing clusters. A popular example is Microsoft's Configurable Cloud [3], a cloud-scale FPGA-accelerated system originated from Microsoft's Project Catapult. Meanwhile, Amazon has started offering high-performance computing instances equipped with FPGAs [4].

The issue that has traditionally hampered the widespread adoption of FPGAs is that they require digital design expertise and specialized programming skills. As a consequence, along with the increase in popularity of FPGAs as part of data centers and high-performance computing clusters, 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 [5, 6]. However, it has been shown that the direct porting of OpenCL code designed for GPU often leads to poor results both in terms of performance and resource utilization [7-9].

In the past few years there has been an increasing interest in accelerating finite state automata. Almost all finite automata processing engines for GPU, implemented using OpenCL or CUDA, have a memory-intensive design. Since the off-chip memory bandwidth of OpenCL-enabled FPGA boards is significantly lower than that of similarly priced high-end GPUs, those designs are not a good fit for FPGA. Our recently proposed SIMD_NFA engine [20] has characteristics that make it a better candidate for FPGA deployment. Specifically, it limits the use of off-chip memory by encoding the NFA topology within the parallel kernel rather than in memory, has a SIMD-friendly design, and is synchronization free. Here, we investigate the deployment of the OpenCL version of SIMD_NFA, previously evaluated on GPU and Intel platforms, on FPGA. After having verified that, despite its more FPGA-friendly design, the naïve porting of SIMD_NFA to FPGA does not even fit the hardware resources of a reasonably equipped Intel Stratix V board, we propose optimization techniques to retarget SIMD_NFA to FPGA.

In this work, we make the following contribution.

- We study the deployment of SIMD_NFA, an existing automata processing engine designed for SIMD platforms, on FPGA.
- We propose a set of optimization techniques aimed to improve the resource utilization and the throughput of SIMD_NFA on OpenCL-enabled FPGA devices. These include: structural code changes, alternative memory layouts, adjustments to the degree of parallelism of the code, and best-practice optimizations. The SIMD_NFA code is automatically generated by a compiler toolchain given an input NFA topology, and the proposed optimizations are easily incorporated in the code generator.
- We study the applicability of some of our proposed optimizations to generic applications beyond SIMD_NFA.
- We perform an extensive experimental evaluation using various NFA topologies from an open-source benchmark suite [21]. From the resource utilization perspective, our evaluation shows that our optimizations not only allow

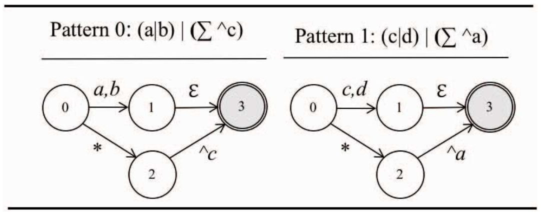


Figure 1. Fixed-topology NFAs accepting two regular expressions (pattern 0 and pattern 1) with the same structure but different symbols.

the code to fit the reference FPGA, but also enable to deploy up to 3 execution pipelines on it. From the performance perspective, our optimizations lead to speedups up to 4x over the first optimized code-variant that fits the NFA design on the considered FPGA.

II. Background on OpenCL for FPGA

OpenCL is an open standard that allows writing programs for a variety of parallel computing platforms, including multi-core CPUs and graphics processing units (GPUs) [22]. OpenCL allows programmers to write their applications in an architecture-independent fashion, and seamlessly deploy them on multiple OpenCL-enabled devices. OpenCL application code consists of two parts: host and kernel code. The host code handles data allocations, communication between host and device, device configuration, and kernel launch. The kernel code contains the core of the computation parallelized and executed on the device. OpenCL adopts a hierarchical multithreading model, whereby threads, called *work-items*, are grouped into *work-groups*. This multithreading model matches the hierarchical hardware organization of GPUs.

Recently, there has been interest in extending the use of OpenCL to FPGAs, thus freeing programmers from the need to write HDL code. Both Intel and Xilinx have developed their OpenCL-to-FPGA toolchains [5, 6]. These toolchains include a compiler that allows converting OpenCL kernels into FPGA bitstreams, and they support two execution models: *single work-item* and *NDRange* kernels. Single work-item kernels are essentially single-threaded functions, while NDRange kernels are executed in parallel by multiple work-items. In both cases, FPGAs provide parallelism through pipelining. In the NDRange case, pipeline replication provides an additional level of parallelism that can enable the parallel execution of multiple work-groups.

Similarly to GPUs, OpenCL-enabled FPGA boards have a three-level memory hierarchy. First, each work-item has a *private memory* that offers low latency but has limited size. Private memory is implemented on FPGA either through registers or Block RAM (BRAM). Second, work-items belonging to the same work-group share a *local memory*, which offers low latency and high bandwidth and resides in the BRAM. Third, all work-groups share a high-latency *global memory* that resides on the external DDRs of the FPGA board.

By potentially lowering the barrier to the adoption of FPGAs by a broader audience, OpenCL-to-FPGA toolchains can have significant impacts. Previous work (e.g., Podobas et al. [23]) has shown that the Intel's OpenCL-to-FPGA toolchain can lead to more efficient FPGA codes that other existing high-level design strategies for FPGA. Yet, the automatic generation of efficient FPGA implementations from OpenCL code is not a trivial problem. First, OpenCL code tailored to a device might perform poorly on another device with different architectural features and offering different forms of parallelism. For example, while GPUs provide massive thread-level parallelism, FPGAs offer more limited pipeline parallelism. Second, the HDL code generated by these toolchains has often little readability, making it extremely challenging to add optimizations on top of it. To this end, there have been efforts aiming to understand the limitations of OpenCL codes on FPGA and exploring the effect of best-practice and custom optimizations on performance.

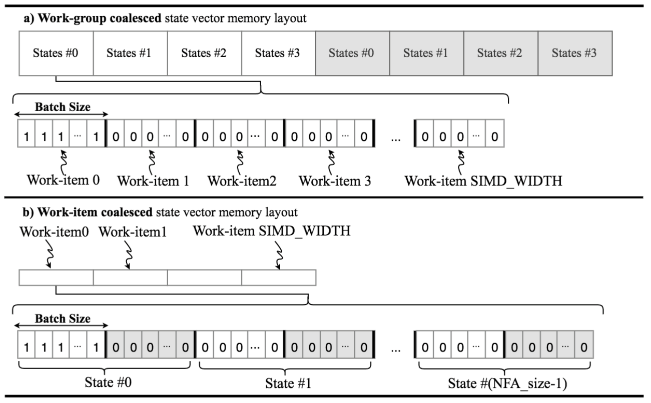


Figure 2. State vector layout for the NFA topology in Figure 1. For each state, each work-item has an associated 32-bit word storing the activation bits corresponding to that particular state in 32 distinct NFAs. Section (a) shows the original *work-group coalesced* layout. In this layout, *current*$_{sv}$ (in white) and *future*$_{sv}$ (in grey) are placed in memory one after the other. For each state vector, 32-bit words processed by subsequent work-items are placed contiguously in memory. Section (b) shows our alternative *work-item coalesced* layout, with alternating placement of *current*$_{sv}$ and *future*$_{sv}$. For each work-item, the two 32-bit words corresponding to *current*$_{sv}$ and *future*$_{sv}$ are stored contiguously in memory.

III. SIMD_NFA Processing Scheme

The application of reference (SIMD_NFA [20]) implements non-deterministic finite automata traversal and has been deployed on GPU, Intel Phi and Intel Skylake processors. Finite automata (FA) are a computational model that can be used to implement regular expression matching, whereby an input text (or stream) is searched for the occurrence of a given set of textual patterns. When using FA, patterns are represented as a set of states and state transitions (e.g., Figure 1). Pattern matching is performed by traversing the precomputed FA guided by the symbols in the input text. During traversal, a transition outgoing from an active state is followed if the input character matches a symbol on that transition; the activation of a *final* state triggers the match of the corresponding pattern [24]. FA can be in deterministic or non-deterministic form (DFA and NFA, respectively). Due to their compactness and intrinsic parallelism (they allow multiple concurrent state activations), NFA are at the core of many parallel pattern matching implementations.

Most NFA traversal engines for GPU [10-12] store the NFA states and transitions in global memory and parallelize

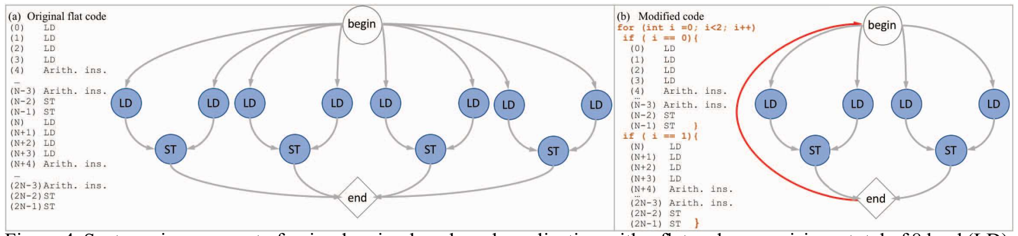
```
a) SIMD-NFA kernel
1: $current_{sv} \leftarrow initial_{sv}$
2: while (!input_stream.empty) do
3:     c ← input-streams[stream_id][counter++]
4:     $future_{sv} \leftarrow current_{sv}$ & $persistent_{sv}$
5:     topology-specific-traversal (c, $current_{sv}$, $future_{sv}$)
6:     $current_{sv} \leftarrow future_{sv}$
7:     $future_{sv} \leftarrow 0$
b) function topology-specific-traversal (c, $current_{sv}$, $future_{sv}$)
8: // source: state 0
9:      // destination: state 1
10:       mask ← match_check(c, tx_symbol_arr,0* pattern_count,2)
11:       sv_update(mask,0,1,1,0,0,0)  //positive tx handling
12:       sv_update (null,null,null,0,0,0,1)  //epsilon txs handling
13:       // destination: state 2
14:       sv_update(null,0,2,0,0,1,0)  //wildcard tx handling
15: // source: state 1      (no non-epsilon tx)
16: // source: state 2
17:       // destination: state 3
18:       mask ← match_check(c,tx_symbol_arr,2*pattern_count,1)
19:       sv_update(mask,2,3,0,1,0,0)  //negative tx handling
c) function match_check (c, tx_symbol_arr,offset, #symbols_per_tx)
20:   mask ← 0
21:   for symbol-id in [1, #symbols_per_tx] do
22:      partial_mask ← 0
23:      for batch_id in [1, batch_size] do
24:         symbol ← tx_symbol_arr [offset++]
25:         if (symbol==c)
26:               set batch_id bit of partial_mask
27:      mask|← mask | partial_mask
28:   return mask;
d) function sv_update (mask, src, dst,is-pos,is-neg,is-wildcard,is-epsilon)
29:   if (is-pos)               //transition with positive symbol
30:      $future_{sv}(dst)$ |← mask & $current_{sv}(src)$
31:  else if (is-neg)           //transition with negative symbol
32:      $future_{sv}(dst)$ |← ~mask & $current_{sv}(src)$
33:  else if (is-wildcard)   //wildcard transition
34:      $future_{sv}(dst)$ |← $current_{sv}(src)$
35:  else if (is-epsilon)    //epsilon transition
36:     $future_{sv}(dst)$ |← $future_{sv}(src)$
```

Figure 3. NFA traversal pseudocode. The *topology-specific-traversal* function refers to the NFA topology shown in Figure 1.

the traversal by distributing the set of active states and their transitions across the work-items (i.e., they leverage state-level parallelism). These implementations are typically irregular, and suffer from high memory bandwidth requirements, irregular memory accesses, control flow divergence, and synchronization overhead (they use atomic operations to update the active states information). SIMD_NFA addresses these issues for applications that rely on fixed-topology NFAs. Differently from existing GPU implementations, 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 uses NFA- and stream-level parallelism: it assigns to each work-item a distinct set of 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 its SIMD-like operation, its limited memory and control flow divergence, and its lack of synchronization, SIMD_NFA is a good candidate for FPGA. Thus, in this work we evaluate the deployment of an OpenCL version of this code developed for GPU on FPGA.

Here, we illustrate the operation of SIMD_NFA with an example. Figure 1 shows two NFAs with the same topology but different transition symbols corresponding to two regular expressions (e.g., patterns 0 and 1) with the same structure. As mentioned above, the SIMD_NFA engine embeds the topology of the NFA within the traversal code, and stores the transitions' characters and the input streams in global memory. Like other GPU implementations of NFA traversal, 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, and their layout is shown in Figure 2 (a). As can be seen, 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 32 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. We call this memory layout *work-group coalesced* state vector layout.

Figure 3 shows the pseudocode of the NFA traversal kernel. 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. The *topology-specific-traversal* function shown in Figure 3(b) corresponds to the example in Figure 1. The underlined bitmap operations on state vectors are performed by all work-items in parallel; the iterations of all *for* and *while* loops in the code are performed sequentially by all concurrent work-items. First, the input characters are sequentially fetched from global memory based on the input stream identifier (lines 2 and 3). Line 4 supports the activation of persistent states (i.e., states that, once activated, will remain active) efficiently. The *topology-specific-traversal* function 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 topology in Figure 1 contains one transition of each kind: (0,1), (2,3), (0,2) and (1,3) are a positive, a negative, a wildcard and an epsilon transition, respectively. 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* (lines 29-32). Since wildcard and epsilon transitions are input independent, they do not require the mask for updating the $future_{sv}$ (lines 33-36).



(a) Original flat code
```
(0)    LD
(1)    LD
(2)    LD
(3)    LD
(4)    Arith. ins.
…
(N-3)  Arith. ins.
(N-2)  ST
(N-1)  ST
(N)    LD
(N+1)  LD
(N+2)  LD
(N+3)  LD
(N+4)  Arith. ins.
…
(2N-3) Arith. ins.
(2N-2) ST
(2N-1) ST
```

(b) Modified code
```
for (int i =0; i<2; i++)
 if ( i == 0){
  (0)    LD
  (1)    LD
  (2)    LD
  (3)    LD
  (4)    Arith. ins.
  …
  (N-3)  Arith. ins.
  (N-2)  ST
  (N-1)  ST    }
 if ( i == 1){
  (N)     LD
  (N+1)   LD
  (N+2)   LD
  (N+3)   LD
  (N+4)   Arith. ins.
  …
  (2N-3)  Arith. ins.
  (2N-2)  ST
  (2N-1)  ST   }
```

Figure 4. System viewer report of a simple microbenchmark application with a flat code comprising a total of 8 load (LD), 4 store (ST) and a number of arithmetic instructions (a) before and (b) after inserting a control-flow instruction. The system viewer report presents memory and control flow of each block in the design.

Note that all bitmap operations are performed in local memory, and the only global memory accesses are the ones at lines 3 (input streams reads) and 24 (transition symbols reads).

IV. Deploying SIMD_NFA to FPGA

Our goal is to explore and evaluate the deployment of SIMD_NFA on FPGA. The first step is to simply port the original OpenCL code to FPGA using the NDRange and single work-item execution models. The original code is in NDRange form and consists of fully independent work-items. We recall that, in SIMD_NFA, different work-items process distinct sets of NFAs and access disjoint portions of the state vectors and of the array containing the transition characters. We construct the single work-item kernel by embedding the body of the NDRange kernel within a nested loop, with the outer and inner loops iterating over work-groups and work-items, respectively. Since the work-items of the NDRange kernel are independent, there won't be any loop-carried dependencies between the iterations of the single work-item kernel. Thus, both kernels are good candidates for pipelining on FPGA.

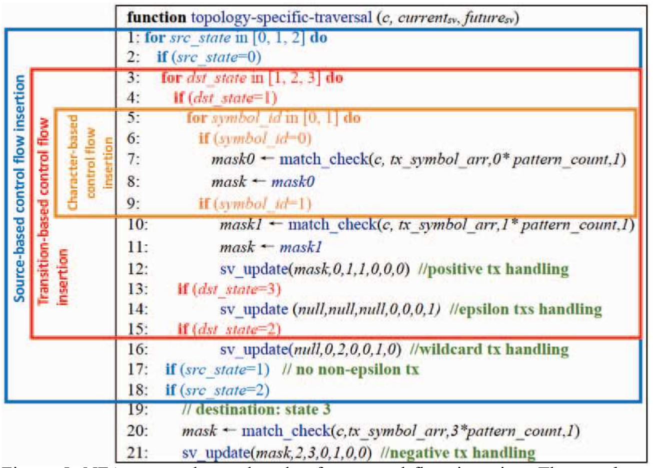
As discussed in Section II, simply porting OpenCL codes optimized for GPU to FPGA can result in low performance and high resource utilization [7-9, 25]. This is mainly due to differences in the underlying hardware and execution model. First, GPUs have a SIMD-like architecture that provides massive multithreading and allows the parallel execution of tens of work-groups and thousands of work-items. FPGAs provide a different type of parallelism. They leverage pipelining to allow parallel execution of work-items and require pipeline replication to allow parallel execution of work-groups. Second, GPU codes often rely on local memory for sharing data and allowing efficient communication among work-items. Barrier synchronization is often required to guarantee the correctness of local memory updates and memory consistency. However, on FPGA such barriers result in pipeline flushes, negatively affecting performance. Therefore, designs with no synchronization requirements are preferred on FPGA. Third, since the memory bandwidth on FPGA is significantly lower than on GPU, FPGA performance is sensitive to the nature of the memory access patterns.

Recall that the SIMD_NFA design does not require synchronization primitives, uses local memory for all state vectors updates, and has a memory layout designed to provide efficient local and global memory accesses. Although based on these characteristics we expect SIMD_NFA to be a good fit for FPGA, naively porting its OpenCL implementation to FPGA results in two sources of inefficiency: excessive logic resource utilization and global memory access stalls. This motivates us to explore a set of custom and best practice optimizations to retarget the code optimized for GPU to FPGA, and to improve resource utilization and traversal throughput. Specifically, we explore four kinds of optimizations: (1) structural changes to the code, (2) alternative memory layouts, (3) adjustment to the NFA-level degree of parallelism, and (4) best practice guide optimizations. In addition, we evaluate the general applicability of the first two optimizations to generic OpenCL code.

A. *Structural code changes*

First, we observe that the resource utilization of the original OpenCL code is so high that it prevents the deployment of the code on a reasonably sized FPGA board, and this holds for all considered NFA topologies. Table I (columns 9 and 10) reports the *estimated* logic utilization of the original SIMD_NFA code on a Stratix V FPGA and seven NFA topologies with various sizes (see Section V for more details). These estimates are provided by Intel's OpenCL-to-FPGA toolchain (area analysis reports) before place-and-route. Some of these estimates (e.g., *Fermi* and *LD_k8_d1* in NDRange form) are less than 100%. However, even in these cases the final design could not be placed and routed on the considered FPGA. In addition, due to the size of the ER kernel code, the toolchain failed to generate the corresponding Verilog files, not even allowing resource utilization estimates for this NFA topology. This led us to investigate possible structural code changes aimed to reduce resource utilization.

To get some insight into the compilation and mapping of OpenCL code onto FPGA, we study the intermediate reports generated by Intel's OpenCL-to-FPGA toolchain. From our observations of the *system viewer* reports, it appears that the compiler breaks the code into blocks and generates a hardware


Source-based control flow insertion
Transition-based control flow insertion
Character-based control flow insertion

```
function topology-specific-traversal (c, current_sv, future_sv)
1: for src_state in [0, 1, 2] do
2:   if (src_state=0)
3:     for dst_state in [1, 2, 3] do
4:       if (dst_state=1)
5:         for symbol_id in [0, 1] do
6:           if (symbol_id=0)
7:             mask0 ← match_check(c, tx_symbol_arr,0* pattern_count,1)
8:             mask ← mask0
9:           if (symbol_id=1)
10:              mask1 ← match_check(c, tx_symbol_arr,1* pattern_count,1)
11:              mask ← mask1
12:              sv_update(mask,0,1,1,0,0,0) //positive tx handling
13:      if (dst_state=3)
14:              sv_update (null,null,null,0,0,0,1) //epsilon txs handling
15:      if (dst_state=2)
16:              sv_update(null,0,2,0,0,1,0) //wildcard tx handling
17:   if (src_state=1)  // no non-epsilon tx
18:   if (src_state=2)
19:      // destination: state 3
20:      mask ← match_check(c,tx_symbol_arr,3*pattern_count,1)
21:      sv_update(mask,2,3,0,1,0,0) //negative tx handling
```

Figure 5. NFA traversal pseudocode after control-flow insertion. The *topology-specific-traversal* function refers to the NFA topology shown in Figure 1.

TABLE I. LOGIC UTILIZATION AND LINES OF CODE (LOC) OF THE ORIGINAL OPENCL CODE AND THE *SRC-CFI* CODE VARIANT FOR VARIOUS NFA TOPOLOGIES (DATASETS IN SECTION V). THE DATA ARE COLLECTED ON AN INTEL STRATIX V FPGA. GREY CELLS INDICATE DESIGNS THAT DO NOT FIT THE CONSIDERED FPGA. IN THESE CASES, SINCE PLACE&ROUTE WAS NOT POSSIBLE, THE REPORTED LOGIC UTILIZATION IS AN *ESTIMATE* BY THE OPENCL-TO-FPGA TOOLCHAIN.

Dataset	*Topology Characteristics*							*Original-flat-code*		*src-cfi code-variant*					
								NDRange	*Single Work-item*	*NDRange*			*Single Work-item*		
	states #	*tx #*	*tx char#*	*positive tx #*	*negative tx #*	*wildcard tx #*	*epsilon tx #*	*Logic util. (%)*	*Logic util. (%)*	*LOC*		*Logic util. (%)*	*LOC*		*Logic util. (%)*
										OpenCL	*Verilog*		*OpenCL*	*Verilog*	
HD_k8_d1	17	24	23	15	8	0	0	112	328	217	~118k	31	227	~648k	70
Fermi	18	26	24	11	0	13	0	69	245	210	~195k	43	220	~155k	36
LD_k8_d1	18	42	16	16	0	17	8	88	264	231	~166k	28	241	~1,425k	58
SPM	21	47	69	51	17	1	0	195	1256	267	~448k	124	277	~432k	246
HD_k20_d1	41	59	59	39	20	0	0	266	829	314	~224k	37	324	~1,751k	172
LD_k20_d1	42	101	40	40	0	41	20	198	698	340	~401k	33	350	~4,998k	132
ER	72	196	195	148	47	0	0	-	-	618	~1,244k	146	628	~17,383k	488

module for each of these code 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 (i.e., we observe instances where code blocks including branch instructions are mapped to a single module), control flow instructions are used to break the code into code blocks for module generation. In addition, we observe that the compiler tends to unroll small loops with known loop trip counts, effectively limiting the number of control flow operations that can lead to separate code blocks. In summary, long straight-line code can result in large modules, decreasing the opportunities for hardware reuse.

The original OpenCL code is automatically generated given an input NFA topology. In particular, the code generator processes the specified NFA topology transition-by-transition. For each positive and negative transition, it inserts a call to *match_check* and *sv_update*; for each wildcard and epsilon transition, it inserts a call to *sv_update* (Figure 3(b)). The resulting *topology-specific-traversal* code has a "flat" structure. Our observation on module generation suggests that the large resource utilization of the "flat" code 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. Based on these observations, we introduce structural changes in the code without modifying its functionality. Specifically, we incrementally add control-flow statements to facilitate the generation of smaller modules.

As an example, Figure 4 (a) presents the system viewer report for a simple microbenchmark that includes 8 load, 4 store and a number of arithmetic instructions. As can be seen, this code leads to a single module that includes 8 load and 4 store units. However, dividing this flat code into two sections by adding control flow statements (as shown in Figure 4 (b)) results in a smaller block with only 4 load and 2 store units. The corresponding module is instantiated once. However, due to the presence of the loop, the module is now reused by the two code blocks corresponding to the two if-statements. Consequently, adding *redundant* control flow statements (while preserving the original code functionality) can significantly reduce the resource utilization of the OpenCL code. For the SIMD_NFA code, we add control flow statements in three stages: source-based, transition-based, and character-based insertion. The resulting code variants could be automatically generated by modifying the SIMD_NFA code generator.

Source-based control flow insertion (*src-cfi*) – We first insert a loop that iterates over the states of the NFA (line 1 of Figure 5). Then, for each NFA state, we introduce an if-block that encapsulates the logic related to the transitions outgoing from that state (i.e., the required calls to the *match_check* and *sv_update* functions). These added control flow statements (line 2, 17 and 18 of Figure 5) allow the compiler to split the code into code blocks and, consequently, to generate smaller modules. Recall that, when a code block is executed in multiple iterations of a loop, its logic is instantiated only once. This allows logic reuse, not possible in the flat code. In addition, this motivates the insertion of the loop at line 1 of Figure 5, which is not required from a functional perspective. Table I (columns 13 and 16) reports the resource utilization of this code variant. We note that the implementation of the module corresponding to each source state depends on several factors. First, the outgoing transition types, leading to the instantiation of the appropriate bitmap logic from the *sv_update* function within the module. Second, the number of characters per transition, which defines the logic and the number of load units required by the *match_check* function. Third, the number of outgoing transitions, which indicates the number of instantiations of the logic associated to *match_check* and *sv_update* within the same module. If two source states have the same attributes, they can lead to the instantiation of the same module. To allow for module reuse, we aim to generate modules with similar logic. To this end, we gradually increase the number of control flow statements so as to allow breaking the *topology-specific-traversal* function into finer-grain code blocks.

Transition-based control flow insertion (*tx-cfi*) – 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 (line 3, 4, 13 and 15 of Figure 5). Here, the design of the module corresponding to a transition depends only on the transition type and the number of characters per transition. Therefore, this code change increases module granularity and the likelihood of having code blocks with similar logic requirements. This, in turn, facilitates code reuse (note that, as in the previous code change, the modules are instantiated within a loop).

Character-based control flow insertion (*char-cfi*) –The *match_check* function iterates over all the characters triggering a transition (Figure 3, line 21). We aim to simplify the logic of the *match_check* function. For this purpose, we let this function check only one character by removing the loop at line 21 from *match_check* and let this function check only one character. The *topology-specific-traversal* function will then invoke *match_check* multiple times, once for each character of the transition. We then insert a loop in *topology-specific-traversal* to iterate over these function calls, and if-statements to split the iterations into different modules (line 5,6 and 9 of Figure 5).

It is important to note that nested loops can increase the complexity and the logic utilization of the kernel (this holds especially for single work-item kernels) [26]. To address this problem, we manually coalesce the loops of these three structural changes into one (placed at line 1 in Figure 5).

General applicability of control-flow insertion technique – In order to study the general applicability of this method to other applications, we used synthetic microbenchmarks similar to the one of Figure 4. Specifically, we first generated flat codes with different instruction mixes, and then we inserted (functionally unnecessary) control flow statements so as to allow the compiler to identify similar code blocks and enable module reuse. As in Figure 4(b), control flow insertion was done automatically by breaking the code into *CFS* equally sized code blocks, and inserting an external for-loop with *CFS* iterations and an if-statement around each of the *CFS* code blocks. We first constructed microbenchmarks with the same number of load and store operations but different arithmetic intensities (namely, 1, 2 and 4), and we inserted 5, 10 and 20 control flow statements into the flat code. As shown in Figure 6(a), control flow insertion results in up to 3.7x logic utilization reduction (with arithmetic intensity 4). We then studied the effect on resource utilization of the similarity between the code blocks generated by control flow insertion. The results of these experiments are shown in Figure 6(b), where the arithmetic intensity is set to 1. The percentage of similarity refers to the number of arithmetic operations that are shared by the different code blocks. As expected, the best results are achieved when all the code blocks have the same structure. However, even if the arithmetic operations differ significantly across code blocks, the compiler leverages the number of load and store operations per block to generate smaller, reusable modules. We also verified that this decrease in logic utilization does not come at a performance cost even when the flat design fits the FPGA used. These results suggest the general applicability of our proposed control-flow-insertion technique beyond SIMD-NFA.

B. *Changes to the memory layout*

Figure 2 (a) shows the layout of the state vectors for the original OpenCL code. This layout, called *work-group coalesced*, allows subsequent work-items within the same work-group to access contiguous memory chunks. As a consequence, it is suitable for GPU, where work-items belonging to the same work-group are executed in parallel and contiguous memory accesses avoid bank conflicts for local memory. The work-group coalesced layout, however, might not be optimal for FPGA, where the work-items within a work-group are executed in a pipelined fashion, rather than a SIMD-like fashion. Specifically, we note that single work-item kernels are executed sequentially and potentially present more locality per each loop iteration. This considered, on FPGA we also test the layouts of Figure 2(b). This layout, called *work-item coalesced*, stores the state vectors work-item by work-item. In other words, it groups together all the data processed by the same work-item.

General applicability of work-item coalesced layout – In order to evaluate the generality of the work-item coalesced memory layout and its effectiveness, we created a microbenchmark that uses this layout for local memory. Specifically, the kernel code consists of three loops: (1) transfer of input array from global to local memory, (2) computation in local memory, (3) transfer of output array from local to global memory. Each iteration of loop (2) processes a variable number of adjacent elements (*AdjAcc*), affecting data locality. Figure 6(c) shows the speedup achieved using work-item coalesced over work-group coalesced layout. We observe slight speedups up to 1.23x, 1.14x and 1.08x for arithmetic intensities 0.5, 1 and 2, respectively, and 32 adjacent accesses per iteration. The speedup plateaus beyond *AdjAcc* of 32. However, we did not observe performance benefits when the workitem-coalesced layout is used for global memory or with NDRange kernels.

C. *Changing the degree of NFA-level parallelism*

As mentioned in Section III, in SIMD_NFA each work-item traverses a *batch* of NFAs concurrently. In the original GPU code the batch size is set to 32, corresponding to the size of a *warp* (i.e., a SIMD unit on GPU). On FPGA, there is flexibility in the selection of the batch size. The batch size (B_{size}) 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

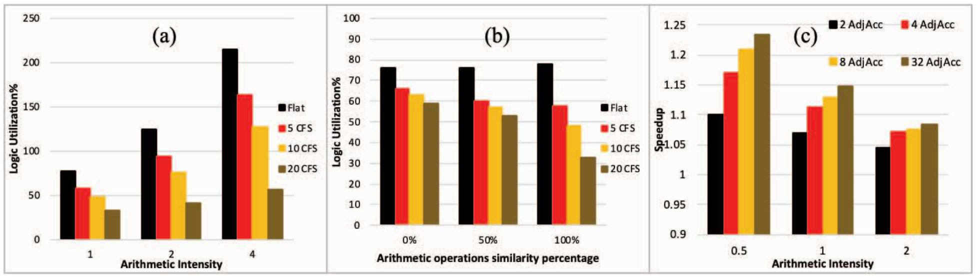


Figure 6. (a),(b) Effect of control-flow insertion on logic utilization for microbenchmarks with different arithmetic intensities in (a), and different degrees of code similarity across blocks in (b). (c) Speedup of single work-item code using work-item coalesced layout over work-group coalesced layout. AdjAcc is the number of adjacent memory accesses per iteration. The size of the array processed is the same across experiments (8194 floats).

the *match_check* function consists of B_{size} bits. As a consequence, the loop that fills that mask (line 23 in Figure 3) has B_{size} iterations. Because it has a loop-carried dependency, that loop cannot be pipelined, and is automatically unrolled by the compiler. Thus, the number of instructions of the *match_check* function depends on the batch size. Reducing the number of instructions can simplify the resulting logic and enhance the clock rate. Lastly, decreasing the batch size increases the number of work-items required to support the same number of NFAs (and vice versa). In our experiments, we test a batch size of 8 (in addition to 32).

D. *Best practice guide optimizations*

We used several techniques from Intel's SDK for FPGA Best Practice Guide [26] to reduce the logic utilization and improve the performance of the code. These include: explicitly indicating the number of work-items and the size of the local memory required (to prevent the compiler from using the 256 work-item and 16KB local memory default settings), using constant 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 when the resource utilization of a single execution pipeline permits it.

V. HARDWARE CONSIDERATIONS

In this section, we discuss hardware characteristics of our design derived from the analysis of the *report.html* file generated by Intel's OpenCL-to-FPGA SDK. This file provides memory and area utilization data and information on loop structure and pipelining. After having identified hardware-specific bottlenecks of our design, we discuss the effect of various optimizations on those bottlenecks.

The *match_check* function (section c of Figure 3) is the bottleneck of the SIMD_NFA design in terms of both resource utilization and throughput. We note that this is the portion of the code that accesses the transition symbol array (*tx_symbol_arr*) located in global memory, thus making the most use of global load units. According to the compiler report, even after applying the *src-cfi* structural changes, this block of code uses from 30% to 89% of the ALUT and FF resources allocated to the kernel (HDK8D1 single work-item and ER NDRange version). Moreover, profiling data collected during our experiments show that the load units allocated to this section of code stall the pipeline from 45% to 95% of the time across NFA topologies.

The other function used by the traversal code is *sv_update* (section d of Figure 3). This function performs solely bitwise logic operations on vectors stored on chip, and thus it uses mainly ALM and BRAM blocks. Similarly, the bitwise operations on persistent and future state vectors (lines 6 and 7 of Figure 3) require only ALM and BRAM blocks. Since they do not access off-chip memory, the hardware modules corresponding to these portions of code do not cause major pipeline stalls. The only additional use of global memory is by the code at line 3 of Figure 3, which loads input characters from the input stream sequentially, requiring only one load unit. Since the resulting memory accesses are contiguous (and infrequent), they do not lead to pipeline stalls.

Memory and area utilization – The hardware design generated from the flat code for the considered NFA topologies includes up to 185 burst-coalesced global load units and one burst-coalesced global store unit. Burst-coalesced load/store units provide efficient contiguous global memory accesses by buffering requests until the largest possible burst of data can be transferred from/to global memory. This, however, comes at the cost of hardware complexity and additional FPGA resources [26]. Applying all the structural code changes, namely *src-cfi*, *tx-cfi* and *char-cfi,* reduces the number of load units to 9 on all considered NFA topologies for both single work-item and NDRange kernels. Using the *vector* data type further reduces the number of load units to 2. Finally, when using constant memory to store the transition characters, the OpenCL-to-FPGA offline compiler instantiates pipelined load units for the *topology-specific-traversal* function. Since pipelined load units are less expensive than burst-coalesced ones in terms of area, the use of constant memory allows an additional (even if slight) improvement on resource utilization.

Loop structure and pipelining characteristics – The *report.html* file includes critical information on the hardware generated for each loop in the body of single work-item kernels. This information includes pipelining status, initiation interval, and scheduled maximum frequency. For each pipelined loop, the initiation interval indicates the number of clock cycles between the launch of one loop iteration and the next, and is one of the main factors affecting performance. In the optimal case the initiation interval is equal to one, implying that one loop iteration is processed every clock cycle. We observe that, after applying the first structural code change, namely *src-cfi*, there are three loops inside the kernel that are performance bottlenecks: (i) the loop at line 1 of the pseudo-code of Figure 5, (ii and iii) the loops required to implement the statements at lines 6 and 7 of the pseudo-code of Figure 3. For different NFA topologies, loop (i) results to be either unpipelined, or pipelined with a large initiation interval (between 104 to 177). Applying the remaining structural code changes (lines 3 and 5 of Figure 5) allows a reduction in the initiation interval. Recall that we coalesce the loops of all structural changes into one. Implementing *src-cfi*, *tx-cfi* and *char-cfi* enables the pipelined implementation of the final coalesced loop and reduces the initiation interval to 65 and 66 across NFA topologies. It is worth noting that our experimental results show no changes in the value of the initiation intervals resulting from optimizations other than our proposed structural code changes. Loops (ii) and (iii) are used to update the state vectors stored in local memory after processing each input character, and they have initiation intervals equal to 98 and 95, respectively. Applying the *wiCL* optimization reduces the initiation intervals of these loops to 1 and 64 for all considered NFA topologies. These initiation interval reductions lead to considerable performance improvements.

VI. EVALUATION

A. *Experimental Setup*

Hardware – We use an Intel DE5-Net board, which includes a Stratix V FPGA (5SGXA7 family). The off-chip memory consists of 4 GB of DDR3 SDRAM, structured in two banks, with peak bandwidth of 25.6 GB/s. The on-chip memory consists of 2,560 M20K memory blocks, each 20 Kbits in size. The FPGA includes 234,720 Adaptive Logic Modules (ALMs), each containing a LUT and 4 registers, and 256 DSP blocks. The host side program runs on an Intel processor with a 64-bit Red Hat OS. We use Intel FPGA SDK for OpenCL Standard Edition v. 18.1.0 to compile and synthesize our OpenCL code.

Datasets – We use five fixed-topology NFA datasets from ANMLZoo benchmark suite [21], namely: Fermi, Sequential Pattern Mining (SPM), Entity Resolution (ER), Hamming Distance (HD) and Levenshtein Distance (LD) NFAs. HD and LD NFAs aim to identify substrings of length k in an input text within Hamming and Levenshtein distance d from a given pattern, respectively. For HD and LD, we use a small dataset ($k=8$, $d=1$) and a large dataset ($k=20$, $d=1$). The considered NFAs topologies differ in terms of numbers of states, number and types of transitions (Table I, columns 2-8). The NFA topology affects both performance and resource utilization of the resulting FPGA design. Each dataset has 2048 NFAs (patterns) and we use two 15KB input streams, either generated using an open-source trace generator [27] (with probability of match set to 50%) or taken from real gene sequences (for HD and LD only).

Performance metrics – Our evaluation covers traversal throughput and resource utilization. The traversal throughput is defined as $\frac{Input\_size \times stream\_count}{execution\_time}$, where the execution time is the running time of the kernel and does not include data transfers between host and device. As measure of resource utilization, we report logic and BRAM utilization data. Logic utilization data indicate the percentage utilization of the ALMs available on the device. Quartus provides these reports *after* place&route.

B. Experimental results

Figure 7 illustrates throughput, logic and BRAM utilization resulting from applying the Section IV optimizations to the original OpenCL kernel. Missing data in the charts correspond to designs that do not fit the FPGA (due to high logic and BRAM requirements). From left to right, we sort NFA datasets based on their sizes (i.e., the number of states) and we show results from incrementally performing the following optimizations to the flat code: source-based control flow insertion (*src-cfi*), transition- and char-based control flow insertion (*tx-cfi* + *char-cfi*), *work-item coalesced* local memory layout (*wiCL*) for single work-item kernels, *vector* data type usage, *constant* memory usage to store frequently accessed transitions, batch size of 8 (*batch-8*) usage. Since batch-8 significantly reduces throughput, we implement the remaining optimizations to the *constant* version of the code (which encapsulates all previous code changes – from *src-cfi* to *vector*). These remaining optimizations are: 2 levels of automatic SIMD parallelization (*SIMD*), and pipeline replication (*PR*). Note that, except for the *batch-8* optimization, the batch size is set to 32.

Throughput – High-level observations include: First, single work-item kernels in most cases yield lower throughput than NDRange ones. We recall that, on FPGA, the presence of synchronization in the code can lead to inefficiencies due to pipeline flushes. Thus, by avoiding the need for synchronization, the single work-item model is a good fit for codes that would require synchronization if executed in a thread-parallel fashion. Since SIMD_NFA is synchronization-free by design, an NDRange implementation can provide higher parallelism and lead to better performance. Second, for almost all NDRange kernels, the best throughput is achieved when applying all optimization up to *SIMD*, while *PR* does not improve the results. For single work-item kernels, the use of *constant* memory leads to the best throughputs. During compilation we set the target clock frequency to 240MHz, and the reports show a resulting effective clock frequency varying between 150MHz and 250MHz across datasets and code variants. Below, we discuss in more details how the considered optimizations affect performance.

vector – Execution profiling results show that the use of vector variables allows full memory bandwidth efficiency and removes global memory stalls (i.e. one of our main inefficiency causes) completely, allowing for significant speedups (up to 2.4x for NDRange and 4.3x for single work-item kernels).

constant – Constant memory operates as an on-chip cache and offers higher load/store bandwidth than global memory at

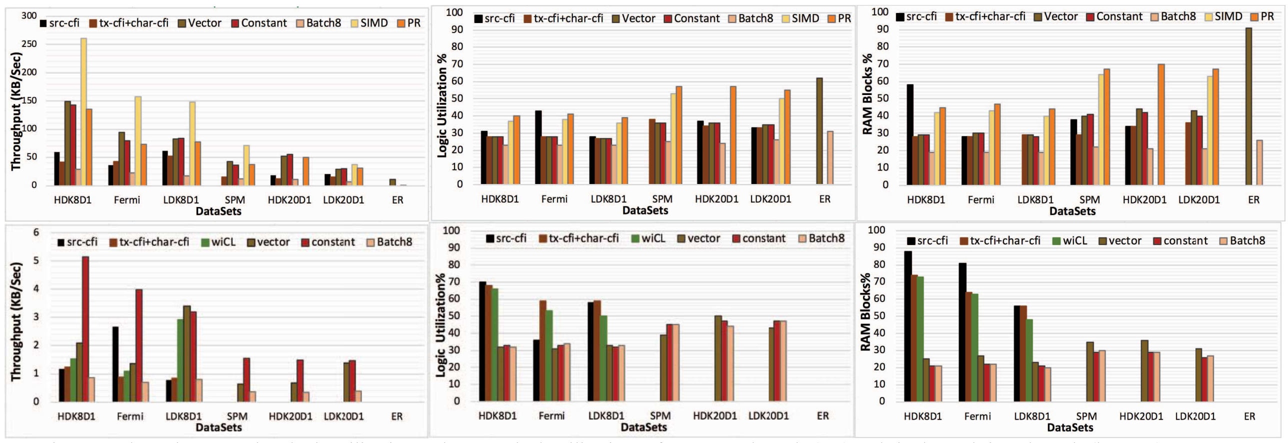


Figure 7: Throughput speedup, logic utilization and RAM Block utilizations of NDRange kernels (top) and single work-item kernels (bottom.)

a significant miss latency cost. Single work-item kernels benefit from constant memory (leading to an average 121% speedup over the *vector* code variant), except for LDK8D1. Due to the limited number of loads required by LDK8D1, the cost of cold cache misses is not amortized by the cache hit speedup. In case of NDRange kernels, we observe an increase in miss rate, likely due to changes in load order due to pipelining. We note that changing the size of constant cache does not significantly change the performance.

We also tested placing the transition characters array in local memory. This is possible only for small datasets that fit the limited local memory capacity. For these datasets, the local memory results are similar to the constant memory ones. Therefore, in Figure 7 we only show the results reported using constant memory.

SIMD and *PR* – We recall that *SIMD* and *PR* are best practice optimizations specific to NDRange kernels, and they allow work-item and work-group level parallelism, respectively. *PR* results in a consistent throughput decrease. The profiling information suggests that the higher degree of parallelism comes at the cost of an increased constant memory miss rate, leading to the observed performance degradation. Yet, *SIMD* parallelization increases memory coalescing and offers the best throughput.

Resource utilization – High-level observations include: First, single work-item kernels require more resources than NDRange ones. Second, using *vector* variables for single work-item kernels is the most effective optimization on resource utilization. For NDRange kernels, after structural code changes allowing the code to fit FPGA, *batch-8* is the most impactful in reducing resource utilization. More details are discussed below.

Structural code changes – We recall that the original flat code does not fit the considered FPGA for any of the datasets. While applying the *src-cfi* optimization allows us to fit most of the datasets on FPGA, applying *tx-cfi* and *char-cfi* on top of it does not lead to a significant reduction in logic and BRAM utilization. For NDRange kernels, Fermi benefits the most from *tx-cfi*+*char-cfi* with a 34% and a 51% logic and BRAM usage reduction. Among our datasets, Fermi has the most irregular topology with the number of transitions' characters per state varying between one to four. Using finer-grained structural changes, therefore, increases the opportunities of code reuse, leading to lower resource utilization.

vector – Vector variables improve resource utilization by reducing the number of load units by a factor of 8. Subsequently, according to the compiler report, a considerable amount of logic is dedicated to the board interface and the global interconnect.

batch-8 – Using a reduced batch size has a clear benefit on the resource utilization of NDRange kernels. However, this comes at the cost of reduced throughput. This is because the reduction of the batch size by a factor of four calls for an increase in the number of work-items by the same factor.

Finally, as expected, logic utilization grows in the same direction as the number of LOC (compare LOC in Table I with the logic utilization in Figure 7). Single work-item kernels tend to be larger than NDRange ones, and they show a slightly looser relationship between LOC and logic utilization. We attribute this to the more complex control logic required by single work-item kernels to implement loop-level pipelining.

Power consumption – We used the *quartus_pow* utility to gather an estimate of the power consumption on the FPGA board. Since this command provides only the estimated power consumption of the FPGA, we add 2.34W to account for the two memory modules on the board [31]. The resulting estimated power consumption of the of *src-cfi* code variant averaged over the considered NFA topologies is 17.9W and 20.7W for NDRange and single work-item kernels, respectively. The power consumption of the other code variants differs from that of *src-cfi* by up to 5W.

Comparison with GPU – In [20] we show that, on a Nvidia Titan Xp GPU, SIMD_NFA achieves single-stream throughputs between 1.3 KB/sec (ER) and 26.4 KB/sec (Fermi). However, GPUs support inter-stream parallelism, and running enough input streams to fully utilize the device results in throughputs from 459.9 KB/sec (ER) to 8744.4 KB/sec (Fermi). While GPUs can achieve high processing throughputs, they have higher power consumption compared to FPGA solutions. For example, the power consumption of the *src-cfi* code variant of SIMD_NFA on an Nvidia TitanXp GPU is on average 142.1W. We measured the GPU power by executing *nvidia-smi* with 1ms intervals, which accounts for the power of the whole board.

Comparison with custom FPGA designs – Several of the most efficient NFA traversal designs for FPGA rely on the one-hot encoding scheme [14][30] and encode the NFA structure uniquely in sequential and combinational logic. While these custom implementations typically yield high processing throughputs, they are constrained by resource utilization, they suffer from long configuration time, and they are not scalable to multiple input streams due to the need for logic replication. In our previous work [30], we have evaluated state-of-the-art FPGA implementations on some of the datasets considered here using a Xilinx Virtex-VI device. Our results show that these custom implementations can achieve peak throughputs up to 0.3 and 0.2 GB/sec on HD datasets with (k=8, d=2) and (k=20, d=2), respectively, and an estimated power consumption of 2.07W on average. These datasets consist of 4030 NFAs and require 1 and 2 FPGA devices, respectively. The main drawback of these logic-based designs is that any changes in the NFA, either in the topology or transition characters, require a full synthesis and FPGA reconfiguration, which takes about 37 minutes. While its throughput is lower than that of custom FPGA designs, for fixed-topology NFA SIMD_NFA offers better configurability and preprocessing cost since it stores transition characters in memory instead of logic. We recall, however, that our goal here is not to propose an optimized NFA engine, but rather to evaluate the limitations of a SIMD-friendly OpenCL code on FPGA and explore compiler optimizations that can be generalized to other applications.

VII. Related Work

Zohouri et al. [7] evaluated the performance and power consumption of six applications from Rodinia benchmark suite [28] on two Altera FPGA devices. They studied several best practice optimization techniques including: pipeline replication, loop unrolling, etc. Their analysis shows that direct ports of GPU-optimized code do not perform well on FPGA. However, FPGA-specific best practice optimizations can significantly improve the performance of the OpenCL code and, while not necessarily allowing the same performance as on modern GPUs, they can lead to more power-efficient implementations. Additionally, they tested CFD application that similar to SIMD_NFA has no synchronization overhead and they report similar results between NDRange and single work-item kernels

Krommydas et al. [8] performed a similar analysis on several OpenCL kernels from the OpenDwarfs benchmark suite [29]. Their study investigates the following aspects: pipeline parallelism on single work-item kernels, manual and compiler vectorization, static coalescing, pipeline replication, and inter-kernel channels. Some of their findings that relate to our observations are as follows. First, single work-item kernels often result in better performance than NDRange ones (this is coherent with the findings in [7]). Second, manual optimizations result in more substantial performance gains compared to automatic ones. while on many memory-bound kernels compiler vectorization and pipeline replication are not effective, manual vectorization can allow for efficient memory coalescing and improve performance. In general, automatic OpenCL compiler optimizations often bring little benefit on FPGA, and manual optimizations are required to achieve more substantial performance gains. Luo et al. [25] investigated the effect of three manual code optimizations on another OpenCL application with irregular memory access patterns. Their analysis shows that these optimizations provide better energy efficiency on FPGA while not affecting the behavior of the code on CPU.

Hassan et al. [9] explored FPGA-specific optimizations for irregular OpenCL applications suffering from unpredictable control flows, irregular memory accesses and work imbalance among work-items. Their analysis covers three directions: exploiting parallelism at different levels, optimizing floating-point operations and minimizing data movement across the memory hierarchy. SIMD_NFA, however, does not present their target code patterns. Thus, their optimizations are orthogonal to the ones considered here.

VIII. Conclusion

In this work we have explored the FPGA deployment of a finite automata. After having verified that the naïve porting of its OpenCL code using the Intel OpenCL-to-FPGA toolchain leads to implementations that do not even fit a reasonably equipped FPGA board, we have proposed optimizations at the level of the OpenCL code to effectively retarget the code to FPGA. Our results show that our proposed techniques, along with best-practice optimization mechanisms, lead to significant improvements in terms of throughput and resource utilization. Some of the proposed techniques (i.e., control-flow insertion and changes to the memory layout) can be generalized to other applications.

Acknowledgment

This work has been supported by National Science Foundation award CNS-1812727.

References

[1] "Top 500 list," https://www.top500.org.
[2] "Green500 list," https://www.top500.org/green500/.
[3] A. M. Caulfield, et al., “A cloud-scale acceleration architecture,” in Proc. of MICRO 2016, pp. 1-13.
[4] "Amazon EC2 F1," https://aws.amazon.com/ec2/instance-types/f1/.
[5] "Intel FPGA SDK for OpenCL Software Technology," https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html.
[6] "SDAccel: Enabling Hardware-Accelerated Software," https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.
[7] H. R. Zohouri, et al., “Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs,” in Proc. of the SC 2016.
[8] K. Krommydas, et al., “Bridging the performance-programmability gap for fpgas via opencl: A case study with opendwarfs,” in Proc. of FCCM 2016.
[9] M. W. Hassan, et al., “Exploring FPGA-specific Optimizations for Irregular OpenCL Applications,” in Proc. of ReConFig 2018, pp. 1-8.
[10] N. Cascarano, et al., “iNFAnt: NFA pattern matching on GPGPU devices,” *SIGCOMM Comput. Commun. Rev.*, pp. 20-26, 2010.
[11] Y. Zu, et al., “GPU-based NFA implementation for memory efficient high speed regular expression matching,” in Proc. of PPoPP 2012.
[12] X. Yu, et al., “GPU acceleration of regular expression matching for large datasets: exploring the implementation space,” in Proc. of CF 2013.
[13] Y. Fang, et al., “Fast support for unstructured data processing: the unified automata processor,” in Proc. of MICRO 2015, pp. 533-545.
[14] R. Sidhu, and V. K. Prasanna, “Fast Regular Expression Matching Using FPGAs,” in Proc. of FCCM 2001, pp. 227-238.
[15] Y.-H. E. Yang, et al., “Compact architecture for high-throughput regular expression matching on FPGA,” in Proc. of ANCS 2008.
[16] M. Becchi, and P. Crowley, “Efficient regular expression evaluation: theory to practice,” in Proc. of ANCS 2008, pp. 50-59.
[17] A. Mitra, W. Najjar, and L. Bhuyan, “Compiling PCRE to FPGA for accelerating SNORT IDS,” in Proc. of ANCS 2007, pp. 127-136.
[18] P. Dlugosch,et al., “An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing,” *Parallel and Distributed Systems, IEEE Transactions on,* pp. 1-1, 2014.
[19] R. Karakchi, et al., “A Dynamically Reconfigurable Automata Processor Overlay,” in Proc. of ReConFig 2017.
[20] M. Nourian, H. Wu, et al., “A Compiler Framework for Fixed-topology Non-deterministic Finite Automata on SIMD Platforms,” in Proc. of ICPADS 2018.
[21] J. Wadden, et al., “ANMLzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures,” in Proc. IISWC 2016.
[22] J. E. Stone, et al., “OpenCL: A parallel programming standard for heterogeneous computing systems,” *Computing in science & engineering,* vol. 12, no. 3, pp. 66, 2010.
[23] A. Podobas, et al., “Evaluating high-level design strategies on FPGAs for high-performance computing,” in Proc.FPL 2017, pp. 1-4.
[24] J. E. Hopcroft, et al., *Introduction to automata theory, languages, and computation*: Addison-Wesley, Reading, Massachusetts, 1979.
[25] Y. Luo, et al., “Evaluating irregular memory access on opencl fpga platforms: A case study with xsbench,” in Proc. of FPL 2017, pp. 1-4.
[26] Intel. "Introduction to Intel FPGA SDK for OpenCL Std. Ed. Best Practices Guide," https://www.intel.com/content/www/us/en/programmable/documentation/rqk1517250959424.html.
[27] M. Becchi, et al., “A workload for evaluating deep packet inspection architectures,” in Proc. of IISWC 2008, pp. 79-89.
[28] S. Che, et al., “Rodinia: A benchmark suite for heterogeneous computing,” in Proc. of IISWC 2009, pp. 44-54.
[29] K. Krommydas, et al., “OpenDwarfs: Characterization of Dwarf-Based Benchmarks on Fixed and Reconfigurable Architectures,” *J. Signal Process. Syst.,* vol. 85, no. 3, pp. 373-392, 2016.
[30] M. Nourian, et al., “Demystifying automata processing: GPUs, FPGAs or Micron's AP?,” in Proc. of ICS 2017, Chicago, Illinois, 2017.
[31] Kingston Technology, "Kingston KVR16S11S6/2 Memory Module Specification," 11 Dec. 2013. [Online]. Available: https://www.kingston.com/dataSheets/KVR16S11S6_2. pdf.