Scaling Out Speculative Execution of Finite-State Machines with Parallel Merge

Yang Xia
The Ohio State University
Columbus, Ohio
xia.425@osu.edu

Peng Jiang
The University of Iowa
Iowa City, Iowa
peng-jiang@uiowa.edu

Gagan Agrawal
The Ohio State University
Columbus, Ohio
agrawal@cse.ohio-state.edu

Abstract

A finite-state machine (FSM) is a key component for many important applications, such as Huffman decoding, regular expression matching and HTML tokenization. Due to its inherent dependencies and unpredictable memory access pattern, FSM computations are considered to be extremely difficult to parallelize. As such, significant research efforts have been made to accelerate FSM computations. Although they achieve promising performance results on multi-core machines, these methods are not scalable for emerging many-core architectures such as the GPUs.

Based on our experiments, we point out that the bottleneck of achieving scalability on GPUs is the sequential merge inherent to these methods. However, unlike the case for simple reduction loops, parallel merge implementations for FSM computations typically require runtime checks and re-executions, which can also impede performance. Based on these observations, we develop parallel merge techniques that select efficient runtime check implementations and avoids unnecessary re-executions. Further, based on GPU architectural features, we develop optimization techniques to improve performance.

We evaluate our parallel merge implementations on a set of representative algorithms. Experimental results show that our parallel merge implementations are 2.02-6.74 times more efficient than corresponding sequential merge implementations and achieve better scalability on an Nvidia V100 GPU.

CCS Concepts • **Software and its engineering → Massively parallel systems**; • **Theory of computation → Parallel computing models**;

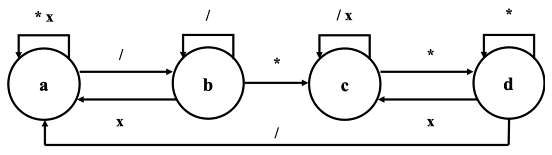
Keywords Finite-State Machines, Speculation, GPUs

PPoPP '20, February 22–26, 2020, San Diego, CA, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6818-6/20/02…$15.00
https://doi.org/10.1145/3332466.3374524

1 Introduction

FSM, as a classical computation model, has been used as a key component in several important applications such as those from data analytics and data mining [5, 7, 19, 20, 26], network security [6, 32], regular expression matching [1], data decoding [12, 25], and others. Figure 1a shows an example FSM, which can be used to identify C-style comments in source code delineated by /* and */. Here, x denotes any character other than / and *. Figure 1b illustrates the corresponding transition table, which determines the transitions from each combination of current state and the input symbol. Figure 1c shows a simple serial implementation of FSM computations.

In several applications, FSMs are executed on very large input datasets and yet require very rapid response. For example, Snort [24] is a network intrusion prevention and detection system based on regular expression matching. It needs to process large numbers of network packets to detect suspicious activities or attacks in a short amount of time. However, FSMs are known to be extremely challenging to parallelize and are also referred to as *embarrassingly sequential* computations [1]. This is because there is a clear dependence between successive loop-iterations. Acceleration of FSM computations is therefore challenging and has attracted significant attention [10, 18, 22, 23, 34, 35]. To parallelize FSMs, previous research efforts [23, 34, 35] have proposed *speculative execution* – this method speculates a state for each chunk and processes chunks in parallel. Once this process is finished, we check whether the speculated state for each chunk was correct. If not, the corresponding chunk will be reprocessed. A significant drawback of this method is that the speculation success rate might be low in many cases, and thus the cost of reprocessing might be high. An alternative is *enumerative execution* [18, 19], where each chunk is processed starting with all the states. Clearly, this approach introduces redundant computations for each chunk, and when the number of states is large, this overhead can be very large. A hybrid method *enumerative speculation* has been proposed recently [10], which processes each chunk with several (but not all) speculated states.



(a)

state \ input	/	*	x
a	b	a	a
b	b	c	a
c	c	d	c
d	a	d	c

(b)

```
state  = a;
for (i in input_data){
      state = T[i][state];
}
```

(c)

Figure 1. An FSM that accepts C style comments in source code delineated by /* and */ (Taken from [18])

Overall, the existing parallelization methods have achieved promising success on multi-core machines where the number of threads/cores is relatively modest. However, scaling FSMs on an architecture like GPUs is still a challenging problem. In fact, existing research work [23] has shown that *speculative execution* cannot scale with many-core CPUs such as a Xeon Phi. The proposed solution in this work [23] has been to only exploit a limited number of cores based on a systematic scalability analysis. Although this approach can achieve significant speedups and even reduce the wasteful use of hardware resources, the question remains whether FSMs can be made to continue to scale with an increasing number of cores.

In this work, we first generalize existing solutions to parallelize FSMs as a *spec-k* method – Here, k denotes the number of speculated states. When k is 1, the method becomes speculative execution, whereas when k is equal to the total number of states (N), it becomes the enumerative method. Other values of k denote ordinary enumerative speculation. We demonstrate that scaling general spec-k methods on GPUs remains challenging. Particularly, with both speculative and enumerative speculation methods, the dominant cost is to *merge* the states after parallel executions of chunks. The reason is that the default implementation of a merge, which is the *sequential merge*, can have a very large overhead when the number of cores is large. This leads to the choice of performing a *parallel* or *tree-like* merge. However, when speculations are involved, such a parallel merge is non-trivial. Thus, a major contribution of this paper is that we implement a code generator, which can generate efficient parallel merge implementations on GPUs. As far as we know, this is the first work that supports parallel merge with speculations on GPUs. In addition, our code generator explores several key design choices. Finally, we develop effective optimizations to further improve performance of FSM computations on GPUs. Specifically, as a major overhead of FSM computations is the unpredictable and random accesses to the transition table, we cache *hot* entries of the transition table into shared memory. Thus, we effectively exploit shared memory even when the transition table is large.

Table 1. Key terms of FSM computations and their meanings in this work

Terms	Meaning
num_states (N)	number of states for an FSM
num_guess (k)	number of speculated states for each chunk
num_inputs	number of different kinds of input symbols
num_item	number of data items for the input
num_thread (n)	number of parallel threads

We evaluate our methods on a set of representative applications: Huffman decoding, regular expression matching, HTML tokenization, and Div7. The main observations from our experiments are as follows. Our parallel merge implementations continues to scale out when we use all cores of the GPU while the performance typically drops under sequential merge implementations, with speedups over sequential CPU execution ranging from 60.44 to 208.69, 2.02-6.74x faster over sequential merge implementations. Also, our optimizations based on GPU architectures are effective: with layout transformations on the input data, our parallel merge implementations achieve an average speedup of 3.79, and caching of hot entries in the transition table leads to a nearly a 50% speedup for Hoffman decoding.

2 Background and Motivation

In this section, we first summarize previous research efforts on parallelizing FSM computations. Then we briefly discuss GPU architectural features that are important for our implementation. Finally, we point out the drawbacks of existing methods through an experimental study and motivate our work.

2.1 Parallelization Strategies for FSM Computations

FSMs can be *deterministic* or *non-deterministic* depending on if a condition can lead to a unique following state. In this paper, we focus on deterministic FSMs (DFA) because non-deterministic FSM (NFA) can be converted to deterministic ones through subset construction. In the following, we will use terms FSM and DFA interchangeably. A finite-state machine (FSM) can be represented as a tuple $(Q, \Sigma, q_0, \delta, F)$, in which Q is a finite set of states, Σ is a finite set of input symbols called the *alphabet*, δ is a transition function that maps a state and an input symbol to another state, $q_0 \in Q$ is the initial state and F is a set of accepting states. To make our discussions clear, Table 1 shows key terms related to FSM computations.

To overcome the inherent sequential characteristics of FSM computations, *speculative execution* was proposed to break the dependencies between state transitions. In this method, the input data is divided into chunks. While one thread is working through a chunk, another thread can start

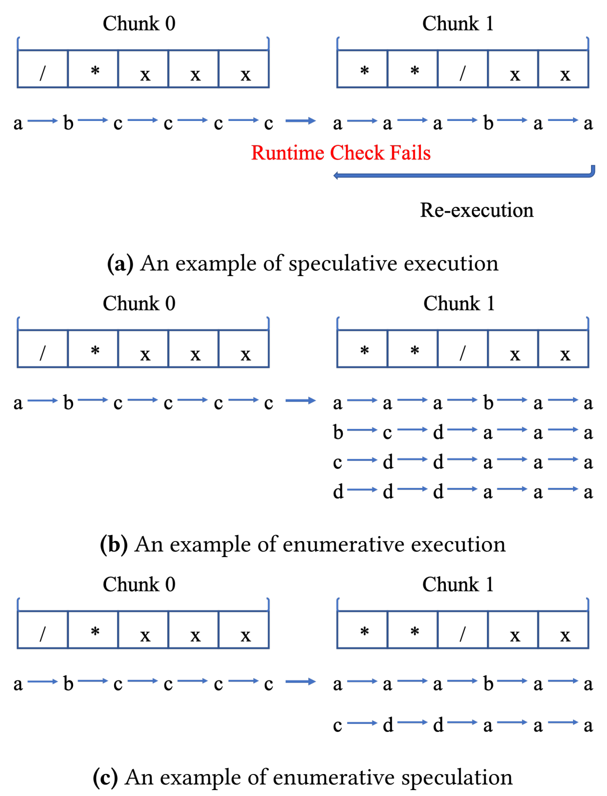

Chunk 0
Chunk 1
Runtime Check Fails
Re-execution
(a) An example of speculative execution
(b) An example of enumerative execution
(c) An example of enumerative speculation

Figure 2. Illustration of three different parallelization methods: speculation, enumerative and enumerative speculation

computing on the next chunk from what is referred to as a *speculated state*. Figure 2a shows an example of speculative computation (using the FSM in Figure 1). The input data is partitioned into two chunks and they are processed in parallel. Chunk 0 starts with the true initial state while chunk 1 starts with a speculated state. A typical strategy to improve the speculation success rate is called *look-back* [21, 34, 35], which means looking back characters at the end of the previous chunk. and speculating state based on this information. In this case, the last character of chunk 0 is *x*. By looking into the transition table, we know only *a* and *c* are possible states after a transition using |em x. Here, we use state *a* as the speculated state for chunk 1. After chunk 0 and chunk 1 finish the local processing, speculative method finds that state *a* was the wrong guess – as a result, it re-executes chunk 1 with the correct state *c*. If the speculation happened to be correct, speculation method could have actually achieved (close to) linear speedup. Through this simple example, we can observe that the performance of speculation method depends heavily on the speculation success rate. When the speculation success rate is low, this method will not be efficient.

An alternative method to parallelize FSM computations is *enumerative execution* [18]. In enumerative execution, instead of starting with a speculated state, we compute a chunk starting with all stats of a FSM. Figure 2b shows an example of enumerative computation of the FSM in Figure 1. In this case, chunk 1 starts computations with all states. Once the computation on chunk 0 has finished, we pick the version of the enumerative computation that started from the correct state, which in this case is the one that started with the state *c*. An obvious drawback is that it can cause a large amount of redundant work.

As illustrated above, both the speculative and enumerative execution have their limitations. This motivated the design of an *enumerative speculation* method [10]. Instead of speculating only a single state like speculative method or enumerating all states as in enumerative execution, they proposed to speculate several possible states. Figure 2c shows the example of enumerative speculation with the FSM in Figure 1. By looking back at the last character, enumerative speculation uses both state *a* and state *c* as two speculated states for chunk 1. After chunk 0 finishes the local processing, enumerative execution method finds that state *c* is the correct initial state and gets the final result. Compared with speculation method, enumerative execution may achieve a higher speculation success rate with more speculated states and thus a lower reprocessing overhead. On the other hand, enumerative speculation reduces the redundant computation overhead since the number of speculated states is typically much smaller than the total number of states. As a result, [10] demonstrated that enumerative speculation can outperform both speculative execution and enumerative execution for a variety of applications.

In this work, we generalize all existing parallelization strategies as *spec-k* method. Here, k is num_guess as also shown in Table 1. Now, speculative execution is a special case in which *k* is equal to 1 and enumerative execution is when k is equal to the number of states, *N*.

2.2 GPU Architectures

From the viewpoint of software, there are three parallelism levels on GPUs. [1] The first level is called *warps*, which are consecutive 32 threads that execute instructions in a *lockstep*. Threads within a warp can exchange register variables using efficient shuffle instructions. The second level is *thread blocks*. Each thread block can hold up to 1024 threads and is assigned to one of the GPU's multiprocessor units, which are called the SMs. Threads within the same thread block are allowed to access shared memory, which is much more efficient than accessing the global memory of the GPU, but still slower than shuffle instructions that threads within a warp can use. Besides, CUDA provides efficient instructions to implement barriers among all threads within a thread block. Finally, the third level of parallelism is the *grid*. Threads from different thread blocks within a grid can only communicate through global memory, which is relatively slow. If all threads within

[1]In this paper, we use terms specific to the CUDA programming model.

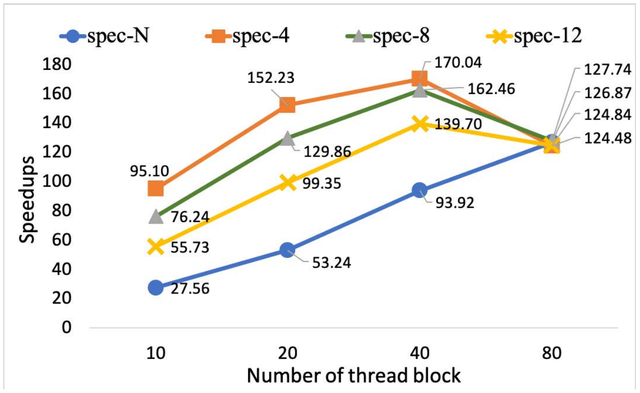


Figure 3. Speedups with different numbers of thread blocks on an Nvidia Tesla V100 for regular expression 2. The baseline is a single-thread CPU program.

a warp simultaneously access global memory locations that lie in the same aligned 128-byte segment, the hardware *coalesces* them into one memory transaction. Otherwise, threads within a warp access multiple 128-byte segments and global memory bandwidth is not fully utilized. In addition, there is no grid-wide barrier among threads of the same grid, so programmers need to implement global barriers by themselves [8, 30].

2.3 Motivation and Opportunities

As also mentioned in the previous section, [23] pointed out the scalability problem of speculative execution (spec-1 method) on Xeon Phi. In this section, we demonstrate that this scalability issue is general for spec-k methods, i.e, exists for other values of k also. We performed an experiment with a regular expression matching FSM on an Nvidia Tesla V100. The regular expression used is "$(.+, . + \backslash.)^4|(.+,)^4|(. + \backslash.)^4$", which is also shown as *Regular Expression 2* in Table 3.

Figure 3 compares the speedups with different numbers of thread blocks on an Nvidia Tesla V100 over a sequential implementation on a CPU. Better performance is seen with a smaller value of k (such as 4 in this case) since the overhead of redundant work is lower. However, the scalability is limited irrespective of the value of k. The limit of scalability is the *merge* step, where results from different threads are combined. This merge step processes results from each thread sequentially, and with a large number of threads, easily becomes the bottleneck. Unlike the previous work [23] where the conclusion was to only exploit a limited number of cores, we propose to solve the scalability problem by design of an optimized (and correct) *parallel merge*.

3 Achieving Scalability with Parallel Merge

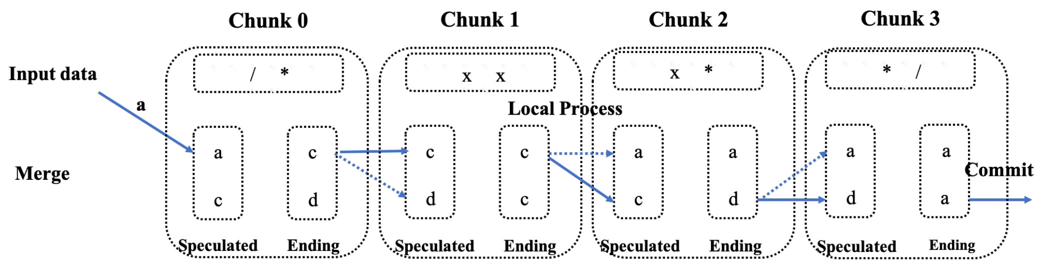
We have shown that sequential merge can be a performance bottleneck for the parallelization of FSM computations. To overcome this bottleneck, a solution is to perform a *parallel tree-like* merge, which is routinely used for reduction-like parallel algorithms [3, 29]. However, unlike simple reductions, it is challenging to support parallel merge when speculation is involved. In this section, we first explain the correctness and performance issues with parallel merge implementation for FSM computations. Then, based on our observations, we present key design ideas that enable scalability with parallel merge.

3.1 Overview of Parallel Merge for FSM Computations

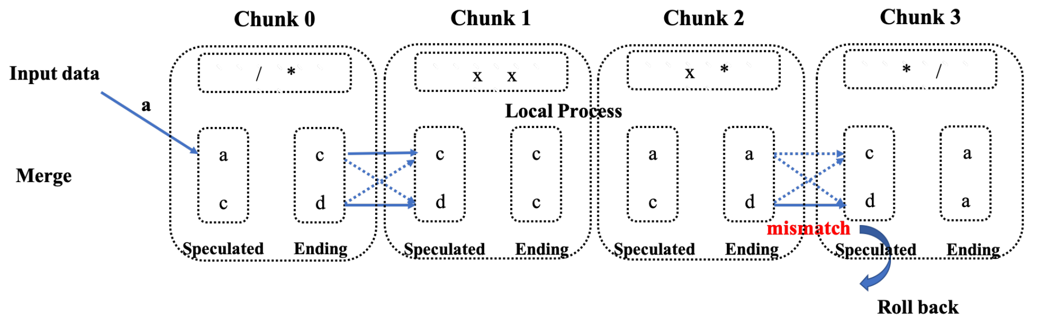
Consider the execution of the FSM in Figure 1. As indicated in the figure, the input to the FSM computations is $/{*}xxx{*}{*}/$. We assume that the input data is divided into four chunks and assigned to four threads. For each chunk, it speculates two states, i.e, it adopts the spec-2 method. After the local parallel processing stage, each chunk gets two corresponding *ending states*. In Figure 4, *speculated states* and *ending states* are shown as two columns under each chunk. Figure 4a illustrates the procedure of sequential merge. As also shown in the figure, the initial state is state *a*. At each step, a single true state from the previous chunk is compared against each speculated state of the next chunk. If there is a match, we update the true state as the corresponding ending state of the next chunk. Otherwise, we re-execute the next chunk with the true state.

Next, we describe the parallel merge procedure for the same example. In the first step, it merges results for the combination of chunk 0 and chunk 1, at the same time, it merges the combination of chunk 2 and chunk 3. Then, it merges these two results to get a final result for all four chunks. This clearly saves time as compared to sequential merge, especially as the number of threads increases. Figure 4b shows the procedure of the first step under a parallel merge. Since we do not always know the true initial state in this step, for each ending state we try to find a match with a speculated state of the next chunk. To be more specific, when we merge results of chunk 0 and chunk 1, two ending states are state *c* and state *d*. For each ending state, we find a match from the next chunk. Thus, the merge result is that for speculated state *a* and speculated state c, the ending state for the combination of chunk 0 and chunk 1 is the state *c*. However, to merge results of chunk 2 and chunk 3, for both state *a* and state *d* from chunk 2, we seek a matching speculated state of the chunk 3. Unfortunately, we are not able to find a match for the state *a*. As a result, to get merge results of the combination of chunk 2 and chunk 3, thread 3 needs to re-execute chunk 3 with state *a*.

From the above example, we can summarize the following challenges when supporting the parallel merge with speculations of FSM computations. First, under the sequential merge, the runtime checks only involve a single true state that needs to be compared against k *speculated states* for each step of



(a) Procedure of Sequential merge for spec-2 method.



(b) Procedure of parallel merge for spec-2 method.

Figure 4. Illustration of issues with parallel merge on FSM computations

the merge. However, under the parallel merge with spec-k, k *ending states* are involved – each of them needs to be compared against each *speculated state* from the next chunk. Therefore, the complexity of runtime checks increases from $O(k)$ to $O(k^2)$. Second and more importantly, under the sequential merge, each chunk gets a true initial state from the previous one and then performs runtime checks (and possible re-executions). If a re-execution is performed, it is indeed necessary. On the other hand, under the parallel merge, since we start to merge without knowing the true initial state, we try to match each ending state against a speculated state from the next chunk. If its match is not found, there may be a need to re-execute. However, since these re-executions are not based on the true initial state, they might be unnecessary. For example, in Figure 4, re-executions on chunk 3 is unnecessary because state a is not the true initial state for chunk 2. Such unnecessary re-executions can significantly degrade performance, especially as both k and the number of threads increases.

To address the above two challenges, we propose two implementation methods in Section 3.2, which are the nested loop and hash implementations. In addition, we propose our re-execution strategy that avoids unnecessary re-executions.

3.2 Implementations of Runtime Checks

As illustrated above, with spec-k method, each chunk ends up with k ending states. To merge results among different chunks, we need to compare these ending states against k speculated states of the next chunk. These runtime checks are similar to *semi-join* operations [2] in databases. Similar to a semi-join implementation, we use a nested loop for these runtime checks, as shown as Algorithm 1. Here, array *states* stores ending states of the current chunk, whereas the arrays *init_states* and *next_states* record speculated and ending states, respectively, from the next chunk. First, we load each ending state of the current chunk as *target_state*. Then we compare *target_state* with each speculated state from the

Algorithm 1 Procedure of nested loop implementation for runtime checks

```
▷ states are ending states of the current chunk
▷ init_states are speculated states from next chunk
▷ next_states are ending states from next chunk
for s = 0; s < num_guess; s++ do
    target_state = states[s]
    found = 0
    for i = 0; i < num_guess;i++ do
        sus_state = init_states[i]
        if sus_state == target_state then
            found = 1
            break
        end if
    end for
    if found == 0 then
        Re-execute the next chunk with states[s]
    else
        state[s] = next_states[i]
    end if
end for
```

Algorithm 2 Procedure of hash implementation for runtime checks

```
1: ▷ states are ending states of the current chunk
2: ▷ init_states are speculated states from next chunk
3: ▷ next_states are ending states from next chunk
4: ▷ bucket_size records the size of each bucket
5: ▷ HASH_SIZE is a predefined parameter
6: ▷ Step 1: Establish the hash table
7: for s = 0; s < num_guess; s++ do
8:     hash_index = init_states[s] % HASH_SIZE
9:     hash_init_states[hash_index][bucket_size[hash_index]] = init_states[s]
10:    hash_end_states[hash_index][bucket_size[hash_index]] = next_states[s]
11:    bucket_size[hash_index]++
12: end for
13: ▷ Step 2: Probe the established hash table
14: for s = 0; s < num_guess; s++ do
15:    found = 0
16:    target_state = states[s]
17:    hash_index = target_state % HASH_SIZE
18:    item_cnt = bucket_size[hash_index]
19:    for i = 0; i < item_cnt;i++ do
20:        if hash_init_states[hash_index][i] == target_state then
21:            found = 1
22:            break
23:        end if
24:    end for
25:    if $found$ == 0 then
26:        Re-execute the next chunk with states[s]
27:    else
28:        states[s] = hash_end_states[hash_index][i]
29:    end if
30: end for
```

next chunk, which is loaded as sus_state. Once we find that there is a match between $target\_state$ and sus_state , we can update an ending state $states[s]$ as $next\_states[i]$ for these two chunks in line 17. Otherwise, we need to re-execute the next chunk with $states[s]$.

Since the complexity of this algorithm is high (O(k^2)), we also implemented a hash version, shown as Algorithm 2. The procedure can be divided into two steps: The first step establishes a hash_table on the ending states from the next chunk, which is shown in lines 7-12. In the second step, for each ending state $states[s]$ of the current chunk, we probe the previously established hash table and try to find a match (lines 14-28). Although hash implementation reduces the time complexity to O(k), the access pattern of these arrays is dynamic. As a result, accessing these arrays will be spilled into local memory, which might hinder performance. Thus, our code generator adopts hash implementation only when num_guess (k) is large. [2]

[2]Based on empirical observations, we use hash implementation when num_guess is larger than 12.

3.3 Re-execution Strategy

To avoid unnecessary re-executions, we propose to delay re-executions with the following method. If we are not able to find a match for an ending state of the current chunk, we mark that the corresponding speculated state of the next chunk is $invalid$ rather than re-execute the next chunk. To be more specific, consider the example in Figure 4. At the end of the first step of merge, we are unable to find a match for state a from chunk 3. In this case, instead of re-executing chunk 3 with state a, we just mark that speculated state a of chunk 2 is invalid. Therefore, the merged results of chunk 0 and chunk 1 are: For initial state a, the ending state is state c; for state c, the corresponding ending state is state c. On the other hand, the merged results of chunk 2 and chunk 3 are: For speculated state c , the ending state is state d. However, for speculated state a, the merge is invalid. Then, in the second step of the merge, we try to merge the results for all these four chunks. For both speculated state a and state c, its ending state is state c, the ending state for the entire set of four chunks is state a. As it is shown, the unnecessary re-execution is avoided and the merged results are still correct.

4 Implementation and Optimizations

In this section, we discuss the implementation details of parallel merge on GPUs and emphasize some of our key design choices. We also describe a key optimization, which is to cache hot parts of the transition table with the goal of reducing memory access overheads.

4.1 Base implementation on GPUs

We first divide input data into chunks and assign one thread to each chunk. Then, each thread processes a local chunk. The procedure of local processing under the enumerative speculation method is shown in Algorithm 3. Each thread declares an array of speculated states as $states[num\_guess]$ and initializes this array based on a simple *look-back* strategy [21]. Then, a thread starts local processing in a loop: at each iteration, a thread reads an input item *in* and operates state transitions for each speculated state. The ending states of each chunk are recorded in $states$ array. It should be noted that our code generator determines the value of *num_guess* before it generates the kernel code. As such, *num_guess* is known at compilation time and access pattern of array $states$ is static, so array $states$ can be loaded in the registers as long as *num_guess* is not large. After the local parallel process is finished, the merge stage is executed with three sub-stages: The first one is *warp stage*. We merge ending states of each thread within a warp using a set of shuffle instructions. After *warp stage* is finished, the last thread of a warp gets the merge results of the entire warp and stores them in shared memory. In the second step, which is *block merge*, we merge results of a single thread block as follows. First, after a barrier synchronization, each lane of the first warp loads merge

Algorithm 3 Procedure of the local processing stage under the enumerative speculation method

```
▷ Initialize speculated states
states[num_guess]
Set up values of array states based on simple look-back strategy
▷ Local process
for index = 0; index < chunk_size; index ++s do
    in = input[index]
    #pragma unroll
    for s = 0; s < num_guess; s++ do
        states[s] = T[in][states[s]]
    end for
end for
```

results of a warp into registers. As such, results of different warps in this thread block are loaded into the first warp. Then, the first warp merges results of different warps using the same procedure as the *warp stage*. Finally, we get the final merge result among different chunks through the third step, which is *global merge*. After another barrier synchronization, a single thread of a thread block writes results of a thread block into global memory. We adopt persistent threads [8]to synchronize merge results between different thread blocks to reduce synchronization overhead. Since the number of thread blocks is typically small, we merge the results in a sequential manner during this step.

Next, we briefly introduce two key implementation choices. **Input data transformation:** In a naive implementation, each thread sequentially accesses its own chunk, which is stored in a continuous section of memory. Since the chunk size is typically large, consecutive threads will access distant memory locations. As a result, memory accesses across threads will not be coalesced. To overcome this problem, we change the layout of the input data so that consecutive threads access consecutive memory locations. Obviously, such data layout transformation can have overheads. However, in many cases, algorithms will be executed over the same data set many times. For example, network intrusion detection systems (NIDS) typical check many different regular expressions for a single packet. In such scenarios, one can justify offline input data transformation since the overhead of data transformation is easily amortized by the benefits of coalesced memory accesses.

Persistent-thread Model: On GPUs, programmers can decide how many thread blocks to launch in a grid. However, it is possible that only a subset of these blocks can be executed concurrently on a GPU because of limitations of resources such as the number of registers and the size of the shared memory. When a thread block finishes, the hardware scheduler will choose another one to run until the entire grid finishes. This causes several problems, for example, different thread blocks cannot synchronize and exchange data with each other. To avoid this, we adopt *persistent-thread* model [8]. In persistent-thread model, we only launch as many thread blocks as can be simultaneously active, and assign multiple work items to each thread.

4.2 Reducing Overhead of Transition Table Memory Accesses

A significant difficulty in optimizing FSM executions is that the memory accesses on the transition table are data-dependent and thus unpredictable. For GPU architectures, this indicates that if we store the transition table in global memory, memory accesses to the transition table are non-coalesced and can become a performance bottleneck. To reduce the overhead of random global memory accesses, we can load a copy of the transition table into shared memory for each thread block, i.e., the shared memory can be used as user-managed cache for the transition table. However, the size of shared memory per SM is very limited on GPUs. In Nvidia Tesla V100, the available shared memory is no more than 96 KB per SM, much smaller than the global memory. On the other side, the number of entries in the transition table is $num\_states \times num\_inputs$. If num_states and/or num_inputs is large, shared memory is incapable of holding the entire transition table. As a consequence, the non-coalesced global memory access can significantly degrade performance.

The key observation to relieve the above problem is that a small number of states typically occur with a relatively high frequency in the computation of FSMs. To illustrate this, we consider input *I = print("hello world") /*test print*/* for the FSM shown in Figure 1. As we can see, in this example, execution will stay in state *a* most of the time. To verify this observation, we measured state frequencies of the FSM computation for regular expression 1 as one example. The setups of this application are illustrated in Section 5. Figure 5 shows the cumulative distribution function (CDF) when states are ordered in decreasing order of frequency. As shown in Figure 5, there are 18 states in the DFA of regular expression 1. However, the first 8 states account for around 95% of all transitions.

Based on this observation, we implemented a strategy to cache the *hot* parts of the transition table in GPU shared memory. For simplicity, we use a static scheme where the allocation of states in shared memory does not change during the execution. We count the frequency of each state in the transition table – for example, in the transition table shown in Figure 1b, the frequency of each of states *a* and *c* is 4 and the frequency of each of states *b* and *d* is 2. Thus, we assume that state *a* and state *c* are *hot* states. The intuition here is that with more cases where these states are accessed, it is likely that the frequency of such states is high during state transitions also. When the size of shared memory is limited, we store entries starting with the highest frequency states, till there is no more space in the shared memory. In the case of Figure 1b, there are 12 entries in the transition table. Assuming that the size of share memory is only 6, we would store the first and the third row of the transition table

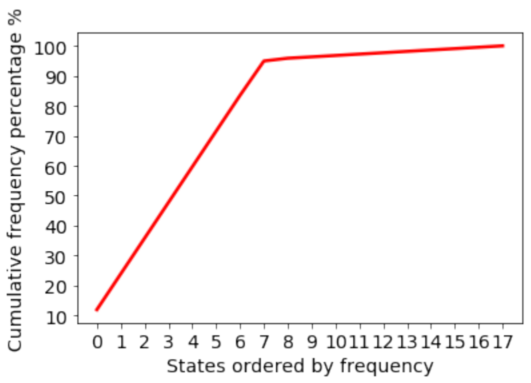


Figure 5. State frequency distribution *cdf* for regular expression 1. The most frequent 8 states account for nearly 95% of all transitions.

into shared memory since that state *a* and state *c* are hot states.

During computations of FSMs, we check whether the row starting with current state is stored in shared memory by accessing a hash table *Hot_States*. We also use a hash function to determine the index in shared memory for such a state – specifically, $Hot\_States[hash(state)] = state$. Here, *Hot_States* is a hash table used to judge whether the row for a given *state* is stored in shared memory. In this work, $hash(state) = (state * SCALE)\%HASH\_SIZE$. Here, *HASH_SIZE* is the size of the hash table and *SCALE* is a parameter used to distribute states more evenly in the hash table. If there are any hash conflicts, i.e. two states have the same hash value, we just keep the one with higher frequency. If a state is in shared memory, we access the corresponding row in the shared memory to transit to the next state and avoid a global memory access. Although we have extra accesses (We need to access the hash table) and extra computation overhead for the hash function, this strategy can still achieve performance improvement since the cost of accessing shared memory is much lower than the cost of a random access in the global memory.

5 Experimental Results

In this section, we present experimental results on a set of representative algorithms to demonstrate the effectiveness of our methods. In the following, we first introduce the setup of our experiments. We proceed to evaluate the effectiveness of our parallel merge implementations – specifically, we compare the speedups of sequential merge and parallel merge. We also vary the number of speculated states and assess its impact. Finally, we demonstrate the effectiveness of our optimizations, which are transforming input data and caching hot states.

5.1 Experimental Setup

Our experiments were performed on an Nvidia Tesla V100, which was released in 2018. Major specifications of this GPU are shown in Table 2. The GPU is attached to an Intel(R) Xeon(R) CPU E5-2680 (2013 Ivy Bridge) running at 2.4 GHz. This machine is also used for sequential CPU runs. The host operating system for our experiments is CentOS Linux release 7.4.1708 (Core). We used Clang libtooling to generate CUDA kernels automatically. The CUDA programs are based on CUDA 10.1 toolkit and NVCC V10.1.168 is used to compile our programs. All sequential programs are compiled using gcc 4.8.5 with "-O3" optimization flag.

Table 2. Major Specifications of a Nvidia Tesla V100

GPU Name	Tesla V100
Architecture	Volta
#SM	80
FP32 CUDA Cores/GPU	5120
Memory Interface	4096-bit HBM2
Register File Size / SM (KB)	65536
Max Registers / Thread	255
Shared Memory Size / SM (KB)	Configurable up to 96 KB
Max Thread Block Size	1024

Table 3. Applications used in experiments and sequential execution times

Application Names	num_states	num_inputs	seq. execution time (us)
Huffman Decoding	205	2	2765070
Regular Expression 1	18	7	2188510
Regular Expression 2	29	3	2185900
HTML Tokenization	38	128	2399090
Div 7	7	2	2394750

Note that the main focus of our work (and thus evaluation) is to show that 1) we can continue to scale with an increasing number of threads with parallel merge and 2) our parallel-merge is more efficient than the sequential-merge. Thus, we use execution times of a simple, hand-coded, one core (single thread) program rather than a parallel CPU implementation as the baseline. We report these baseline execution times in Table 3. The GPU speedups reported do not include the cost of transferring data from CPU to GPU. These costs range between 1.44 to 1.59 seconds, or close to 50% of the sequential execution times. However, when the same input is applied to several FSMs, this cost can be amortized. Furthermore, with technologies like NVLink and use of unified memory between the CPU and GPU, the data transfer costs can be much lower. We also repeated each experiment three times and took the average execution time as the final result. Because the variance in execution times was very small, i.e around 1%, the range is not reported in our figures.

5.2 Scalability Studies

In this subsection, we evaluate the effectiveness of our parallel merge method with a set of real applications. We choose four popular FSM-based applications: Huffman decoding (Huffman), regular expression matching (RegExp), HTML tokenization (HTML), and Div7. These algorithms are collected from previous research efforts in this area [10, 18, 22]. Table 3 shows the details of these applications. Since we experimented on spec-k method, we also show the speculation

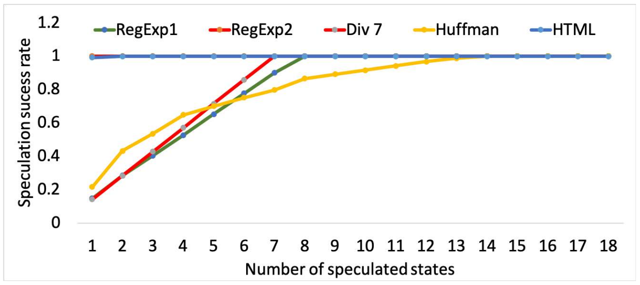


Figure 6. Speculation success rates of different applications

success rates of these applications with different numbers of speculated states in Figure 6.

5.2.1 Huffman Decoding

Huffman coding is widely used in text compression. This algorithm takes advantage of non-uniform frequencies of characters found in texts and encodes the characters to variable-length bit strings. The mapping from characters to the variable-length bit strings is obtained by the Huffman encoding procedure, which is a greedy algorithm that builds a Huffman tree. Huffman decoding, which is the reverse process of Huffman encoding, interprets the bit strings back to the characters based on the Huffman tree. This process is very similar to the process of executing state transitions in an FSM. Thus, we have experimented with the decoding part of this method. For input data, we randomly combine texts from the downloaded books from the Project Gutenberg [3]. The specifications of the four input texts are shown in Table 4. The first column shows the file names of input texts. The number in the filename is the index of that book on the website. The input we generated is called *combined* in this table. The second column shows the number of states in the FSM built by Huffman encoding. After Huffman coding, the number of bits for the input data is 1243106627.

Figure 7 compares speedups of sequential merge implementations and parallel merge implementations on *combined* inputs. For each number of thread blocks, we report the maximal speedup under a particular value of k. As indicated in the figure, with a sequential merge, it achieves maximal speedup 60.44 when the number of thread blocks is 20. As the number of thread blocks increasing, the speedup reduces. However, speedup would keep increasing and reach 407.29 with 80 thread blocks for parallel merge. We also compared with the spec-N method. As indicated in Figure 7, although it is scalable, the maximal speedup is only 15.06, which is much smaller than the spec-k method. The reason that spec-N is much less efficient is that the number of states in this FSM is large (205). As a result, register file is not able to hold such a large array and data is spilled into local memory, which causes a large memory access overhead.

Table 4. Specifications of the input texts used for Huffman decoding.

File Name	Number of States in FSM
76.txt.utf-8	179
50247.txt.utf-8	203
98.txt.utf-8	177
74.txt.utf-8	179
combined	205

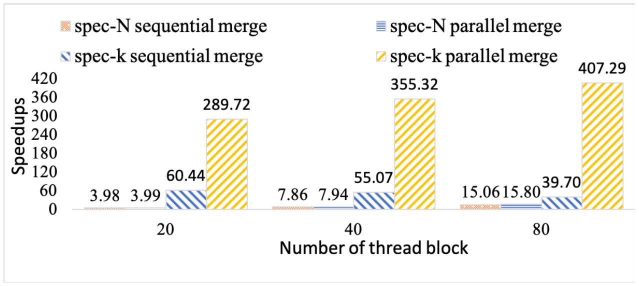


Figure 7. Comparison of speedups of parallel merge implementation and sequential merge implementation of Huffman decoding

Table 5. Regular expressions used in experiments

Regular expressions	Reference Name
$(.^{*}\ l\ .\ ^{*}i\ .^{*}k\ .^{*}\ e)\vert(.^{*}a\ .^{*}p\ .^{*}p\ .^{*}l\ .^{*}e)$	regular expression 1
$(.+,.+\backslash.)^{4}\vert(.+,)^{4}\vert(.+\backslash.)^{4}$	regular expression 2

5.2.2 Regular Expression Matching

Another important application is regular expression matching. We verify effectiveness of our method with two regular expressions [10, 35] as shown in Table 5. Here, the . in the patterns represents any character, \. represents the period, and the superscripts indicates repetitions.

For regular expression matching problem, we search for all the strings that matches a given pattern. Once a match is found, the program outputs the position of the match in the text file. In this experiment, we generated a text file with 1073741824 random low-case characters. Figure 8 and Figure 9 show experimental results with regular expression 1 and regular expression 2, respectively. As indicated in Figure 8, it achieves maximal speedups 72.31 when we launch 40 thread blocks with sequential merge. On the contrary, it achieves speedup 353.99 when there are 80 thread blocks with a parallel merge implementation. As for spec-N method, the maximal speedup is just 164.68. For regular expression 2, the results are similar.

5.2.3 HTML Tokenization

Tokenizing text is another important application of FSM computations. Specifically, a tokenizer reads an input text and outputs a corresponding sequence of tokens. In this work, we performed experiments on HTML tokenization, which is widely used in web browsers and web crawlers.

[3]http://www.gutenberg.org/

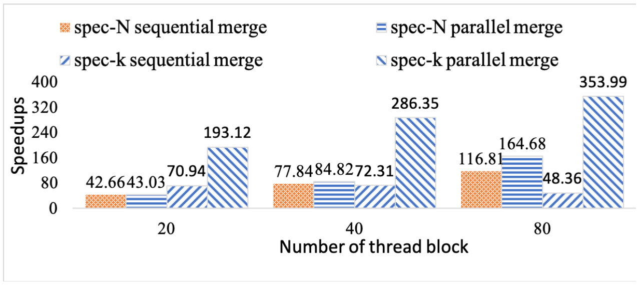


Figure 8. Comparison of speedups of parallel merge implementations and sequential merge implementations for regular expression 1.

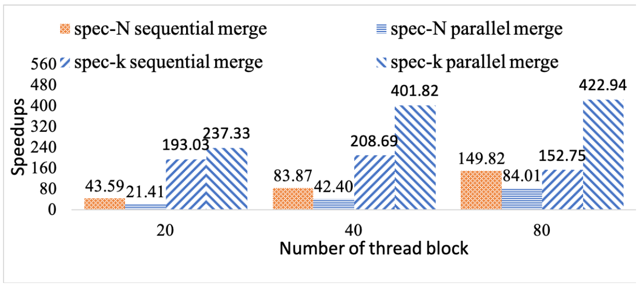


Figure 9. Comparison of speedups of parallel merge implementations and sequential merge implementations for regular expression 2.

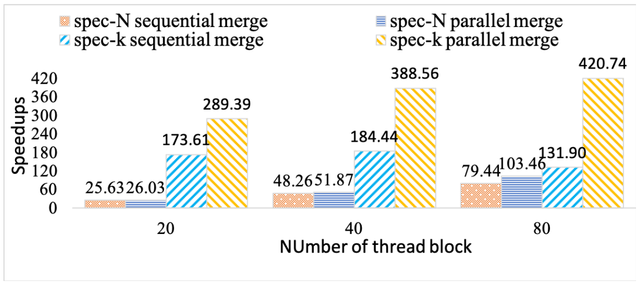


Figure 10. Comparison of speedups of parallel merge implementations and sequential merge implementations for HTML tokenization.

Our implementation uses an FSM with 38 states, and we generated input data by randomly combining web pages from New York Times website, similar to Huffman decoding. The total number of characters of the input data is 1060900492. As indicated in Figure 6, the speculation success rate for this application is high. Thus, it achieves the best performance when the number of speculated states is just one. Figure 10 compares the speedups of sequential merge and parallel merge implementations with different numbers of thread blocks. As shown in the figure, with a sequential merge, it achieves maximal speedup 184.44 when the number of thread block is 40, whereas, with a parallel merge implementation, it achieves the best performance 420.74 when the number of thread blocks is 80. We also compared performance with spec-N method. Although its scalability is also good, however, the maximal speedup it achieves is just 103.46, much lower than thhe spec-k method.

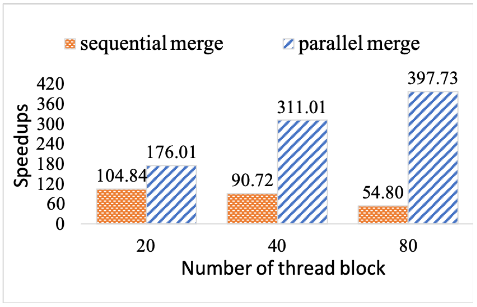


Figure 11. Comparison of speedups of parallel merge implementations and sequential merge implementations for Div 7.

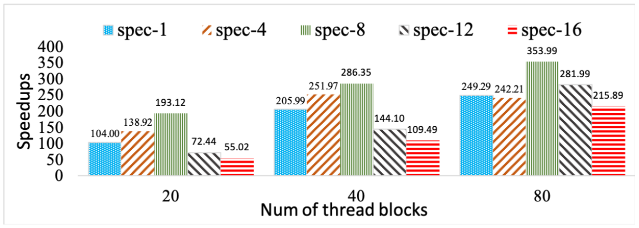


Figure 12. Comparison of speedups with different values of k for regular expression 1 .

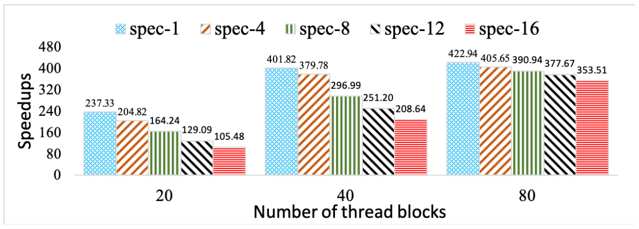


Figure 13. Comparison of speedups with different values of k for regular expression 2.

5.2.4 Div 7

Div 7 is a classic FSM which is illustrated in previous work [22, 34, 35]. This application is to test if a binary sequence is divisible by seven. The input data we tested is a random binary sequence whose length is also 1073741824. In this case, for any input symbol, states would transit to seven different states. Thus, for a random binary sequence, the speculation success rate is linear with the number of guesses, which is also indicated in Figure 6. However, since the number of states is small and no pair of states converge for any input, we adopt spec-N method for Div 7 so that speculation success rate is 100%. Figure 11 compares the speedups of sequential merge and parallel merge implementations with different numbers of thread blocks. As we can see, with sequential merge, the maximal speedup is 104.84, when the number of thread blocks is 20. On the other hand, with parallel merge, it achieves maximal speedups, which is 397.93, when we launch 80 thread blocks.

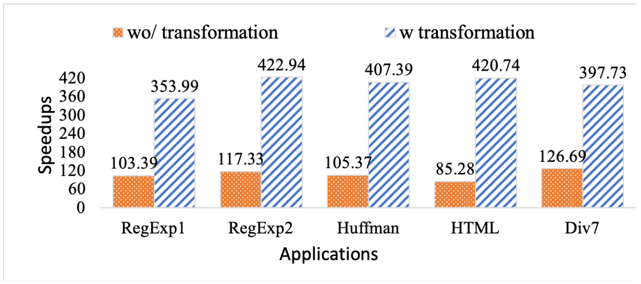


Figure 14. Comparison of speedups with and without data transformation

5.3 Discussion of the impact of k

In the applications that we have studied, the value of k has a significant performance impact on parallel merge implementations. To have a better understanding of it, we study the case of regular expression matching. Specifically, we compare the speedups with different values of k for both regular expression 1 and regular expression 2. The experimental results are shown in Figure 12 and Figure 13. As we can see, for regular expression 1, it always achieves the best performance when we set k as 8 while for regular expression 2, it achieves maximal speedups when we set k as 1. For regular expression 1, as we increase the value of k, the speculation success rate increases first, reaching a value close to 1 when k is 8. Thus, it gets maximal speedup when k is 8. On the other side, for regular expression 2, the speculation success rate is around 1 even when k is 1. When we continue to increase the value of k, the amount of redundant work also increases. As a result, it achieves the best performance with k is equal to 1.

5.4 Effect of Data Transformation

Figure 14 compares the speedups with and without data transformations for parallel merge implementations of these applications. For each case, we only choose the maximal speedups across different numbers of thread blocks and different values of k. As indicated in the figure, the performance is several times better with data transformations because of coalesced memory accesses.

5.5 Effect of Caching Rows Starting with Hot States

To demonstrate the effect of caching rows starting with hot states, we compare the speedups with and without caching for Huffman decoding in Figure 15. We use this application to do a case study because it has the largest transition table among all our applications. As shown in the figure, it achieves nearly a 50% speedup with our caching strategy.

6 Related Work

In this section, we discuss related efforts on parallelizing FSM computations and some of the other closed related work on optimizing irregular applications on GPUs.

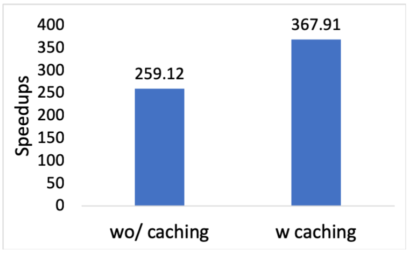


Figure 15. Effect of caching rows starting with hot rows for Huffman decoding.

Parallelization Work on FSMs: Extensive research work has been done on parallelizing FSM computations [4, 9, 11–15, 21, 27, 33–36]. Among them, speculation is a classical method. Prabhu *et al.* [21] proposed a *look-back* strategy to predict the initial state and described two language constructs to apply speculation strategy easily. Several other efforts were specific to a single application: Klein *et al.* [12] apply speculative parallelization for Huffman decoding, Luchaup *et al.* [14, 15] on regular expression matching, and Jones *et al.* [11] for browser front-end. Zhao *et al.* [35] proposed a principled speculation method that pre-analyzes FSMs to get the application-specific information and speculates initial states based on a probabilistic model. They further propose an *on-the-fly* principled speculation technique [34] to remove the overheads of offline training. However, these works assume that the states in an FSM are likely to converge. Another solution to parallelize FSM computations is to apply the parallel prefix sum algorithm [3, 9, 13]. Mytkowicz *et al.* [18] proposed to reduce the redundancy based on state convergence property because they observe that states in most FSMs converge to no more than 16 distinct states after a certain length of input data. Finally, Jiang *et al.* proposed enumerative speculation that we have discussed extensively.
GPU Acceleration of Irregular Applications: There has been a significant amount of research work on accelerating irregular applications on GPUs – we only introduce research works that are closely related to our work. Wang *et al.* [28] conducted a quantitative analysis of the performance impact of shuffle instructions on GPUs and use shuffle instructions to optimize two sequence alignment algorithms. Xiao *et al.* [30] proposed a lock-free implementation of global synchronization, where the kernel-launch overhead is significantly reduced. Based on this, Gupta *et al.* [8] proposed the concept of *persistent kernel execution* and discussed when and why this approach is beneficial. Yan *et al.* [31] implement a matrix-based intra-block scan approach, which is communication efficient. They also proposed to optimize the implementation of scan on both AMD and NVIDIA GPUs automatically. Merrill *et al.* proposed a variable look-back strategy to minimize the waiting for carries. Instead of waiting for the global carry values from the prior chunk, they proposed to use the most recent available global carries and the local carries from chunks that follow that chunk. Maleki

et al. [17] proposed an efficient prefix sum implementation, which is named SAM. In addition, [17] supports two generalized prefix sums, higher-order prefix sums and tuple-based prefix sums. Further, they generalize simple prefix sum implementations to linear recurrences and propose an efficient implementation of any linear recurrence in [16].

7 Conclusion and Future Work

This paper builds on two observations. First, the enumerative speculation or spec-k method offers a continuum in approaching parallel FSM computations. Second, scaling the merge process is challenging when speculations are involved. We have focused on scaling spec-k approach for FSM computations on many-cores. We have further optimized the implementation by focusing on nature of processing and memory hierarchy on GPUs. Our results show that the parallel merge implementation allows scaling of FSM computations. We also show significant benefits of the optimization methods we have developed.

Our evaluation also demonstrates that the value of k has a significant impact on the performance and we also discuss this impact briefly with the example of regular expression matching. However, how to choose the optimal value of k remains a challenging problem. In our future work, we will develop a cost model, which considers the properties of the FSMs, the architecture of GPUs and property of the input data so that we can decide the optimal value of k based on the model.

References

[1] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, et al. 2009. A view of the parallel computing landscape. *Commun. ACM* 52, 10 (2009), 56–67.

[2] Philip A Bernstein and W Chiu Dah-ming. 1981. Using semi-joins to solve relational queries. In *Journal of the ACM*. Citeseer.

[3] Guy E Blelloch. 2004. Prefix sums and their applications. (2004).

[4] Niccolo' Cascarano, Pierluigi Rolando, Fulvio Risso, and Riccardo Sisto. 2010. iNFAnt: NFA pattern matching on GPGPU devices. *ACM SIGCOMM Computer Communication Review* 40, 5 (2010), 20–26.

[5] Cristiana Chitic and Daniela Rosu. 2004. On validation of XML streams using finite state machines. In *Proceedings of the 7th International Workshop on the Web and Databases: colocated with ACM SIGMOD/PODS 2004*. ACM, 85–90.

[6] Domenico Ficara, Stefano Giordano, Gregorio Procissi, Fabio Vitucci, Gianni Antichi, and Andrea Di Pietro. 2008. An improved DFA for fast regular expression matching. *ACM SIGCOMM Computer Communication Review* 38, 5 (2008), 29–40.

[7] Todd J Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. 2003. Processing XML streams with deterministic automata. In *International Conference on Database Theory*. Springer, 173–189.

[8] Kshitij Gupta, Jeff A Stuart, and John D Owens. 2012. A study of persistent threads style GPU programming for GPGPU workloads. In *Innovative Parallel Computing-Foundations & Applications of GPU, Manycore, and Heterogeneous Systems (INPAR 2012)*. IEEE, 1–14.

[9] W Daniel Hillis and Guy L Steele Jr. 1986. Data parallel algorithms. *Commun. ACM* 29, 12 (1986), 1170–1183.

[10] Peng Jiang and Gagan Agrawal. 2017. Combining SIMD and Many/Multi-core parallelism for finite state machines with enumerative speculation. *ACM SIGPLAN Notices* 52, 8 (2017), 179–191.

[11] Christopher Grant Jones, Rose Liu, Leo Meyerovich, Krste Asanovic, and Rastislav Bodik. 2009. Parallelizing the web browser. In *Proceedings of the First USENIX Workshop on Hot Topics in Parallelism.*

[12] Shmuel Tomi Klein and Yair Wiseman. 2003. Parallel Huffman decoding with applications to JPEG files. *Comput. J.* 46, 5 (2003), 487–497.

[13] Richard E Ladner and Michael J Fischer. 1980. Parallel prefix computation. *Journal of the ACM (JACM)* 27, 4 (1980), 831–838.

[14] Daniel Luchaup, Randy Smith, Cristian Estan, and Somesh Jha. 2009. Multi-byte regular expression matching with speculation. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 284–303.

[15] Daniel Luchaup, Randy Smith, Cristian Estan, and Somesh Jha. 2011. Speculative parallel pattern matching. *IEEE Transactions on Information Forensics and Security* 6, 2 (2011), 438–451.

[16] Sepideh Maleki and Martin Burtscher. 2018. Automatic Hierarchical Parallelization of Linear Recurrences. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 128–138.

[17] Sepideh Maleki, Annie Yang, and Martin Burtscher. 2016. *Higher-order and tuple-based massively-parallel prefix sums*. Vol. 51. ACM.

[18] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. 2014. Data-parallel finite-state machines. *ACM SIGARCH Computer Architecture News* 42, 1 (2014), 529–542.

[19] Yinfei Pan, Ying Zhang, and Kenneth Chiu. 2008. Simultaneous transducers for data-parallel XML parsing. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 1–12.

[20] Yinfei Pan, Ying Zhang, Kenneth Chiu, and Wei Lu. 2007. Parallel xml parsing using meta-dfas. In *Third IEEE International Conference on e-Science and Grid Computing (e-Science 2007)*. IEEE, 237–244.

[21] Prakash Prabhu, Ganesan Ramalingam, and Kapil Vaswani. 2010. Safe programmable speculative parallelism. In *ACM Sigplan Notices*, Vol. 45. ACM, 50–61.

[22] Junqiao Qiu, Zhijia Zhao, and Bin Ren. 2016. Microspec: Speculation-centric fine-grained parallelization for fsm computations. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. ACM, 221–233.

[23] Junqiao Qiu, Zhijia Zhao, Bo Wu, Abhinav Vishnu, and Shuaiwen Leon Song. 2017. Enabling scalability-sensitive speculative parallelization for fsm computations. In *Proceedings of the International Conference on Supercomputing*. ACM, 2.

[24] Martin Roesch. 1999. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th USENIX Conference on System Administration (LISA '99)*. USENIX Association, Berkeley, CA, USA, 229–238. http://dl.acm.org/citation.cfm?id=1039834.1039864

[25] Priti Shankar, Amitava Dasgupta, Kaustubh Deshmukh, and B Sundar Rajan. 2003. On viewing block codes as finite automata. *Theoretical Computer Science* 290, 3 (2003), 1775–1797.

[26] Robert A Van Engelen. 2004. Constructing finite state automata for high performance web services. In *IEEE International Conference on Web Services*. Citeseer.

[27] Giorgos Vasiliadis, Michalis Polychronakis, Spiros Antonatos, Evangelos P Markatos, and Sotiris Ioannidis. 2009. Regular expression matching on graphics hardware for intrusion detection. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 265–283.

[28] Jie Wang, Xinfeng Xie, and Jason Cong. 2017. Communication Optimization on GPU: A Case Study of Sequence Alignment Algorithms. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 72–81.

[29] Yang Xia, Peng Jiang, and Gagan Agrawal. 2019. Enabling prefix sum parallelism pattern for recurrences with principled function reconstruction. In *Proceedings of the 28th International Conference on Compiler Construction*. ACM, 17–28.

[30] Shucai Xiao and Wu-chun Feng. 2010. Inter-block GPU communication via fast barrier synchronization. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 1–12.

[31] Shengen Yan, Guoping Long, and Yunquan Zhang. 2013. StreamScan: fast scan algorithms for GPUs without global barrier synchronization. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 229–238.

[32] Fang Yu, Zhifeng Chen, Yanlei Diao, TV Lakshman, and Randy H Katz. 2006. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*. ACM, 93–102.

[33] Xiaodong Yu and Michela Becchi. 2013. GPU acceleration of regular expression matching for large datasets: exploring the implementation space. In *Proceedings of the ACM International Conference on Computing Frontiers*. ACM, 18.

[34] Zhijia Zhao and Xipeng Shen. 2015. On-the-fly principled speculation for FSM parallelization. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 619–630.

[35] Zhijia Zhao, Bo Wu, and Xipeng Shen. 2014. Challenging the embarrassingly sequential: parallelizing finite state machine-based computations through principled speculation. In *ACM SIGARCH Computer Architecture News*, Vol. 42. ACM, 543–558.

[36] Yuan Zu, Ming Yang, Zhonghu Xu, Lin Wang, Xin Tian, Kunyang Peng, and Qunfeng Dong. 2012. GPU-based NFA implementation for memory efficient high speed regular expression matching. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 129–140.