Noname manuscript No.
(will be inserted by the editor)

Massively Parallel Rule-Based Interpreter Execution on GPUs using Thread Compaction

M. Köster · J. Groß · A. Krüger

PREPRINT / Received: 04/24/19 / Accepted: 15/06/2020

Abstract Interpreters are well researched in the field of compiler construction and program generation. They are typically used to realize program execution of different programming languages without a compilation step. However, they can also be used to model complex rule-based simulations: The interpreter applies all rules one after another. These can be iteratively applied on a globally updated state in order to get the final simulation result. Many simulations for domain-specific problems already leverage the parallel processing capabilities of Graphics Processing Units (GPUs). They use hardware-specific tuned rule implementations to achieve maximum performance. However, every interpreter-based system requires a high-level algorithm that detects active rules and determines when they are evaluated. A common approach in this context is the use of different interpreter routines for every problem domain. Executing such functions in an efficient way mainly involves dealing with hardware peculiarities like thread divergences, ALU computations and memory operations. Furthermore, the interpreter is often executed on multiple states in parallel these days. This is particularly important for heuristic search or what-if analyses, for instance. In this paper, we present a novel and easy-to-implement method based on thread compaction to realize generic rule-based interpreters in an efficient way on GPUs. It is optimized for many states using a specially designed memory layout. Benchmarks on our evaluation scenarios show that the performance can be significantly increased in comparison to existing commonly-used implementations.

Keywords GPU · interpreter execution · memory layout · thread compaction

M. Köster
Saarland Informatics Campus, Campus D3.2, Saabrücken, Saarland, Germany

J. Groß
Saarland Informatics Campus, Campus D3.2, Saabrücken, Saarland, Germany

A. Krüger
Saarland Informatics Campus, Campus D3.2, Saabrücken, Saarland, Germany

1 Introduction

Interpreters are omnipresent in the field of computer science. They are typically defined in the form of inference rules that are modeled with the help of imperative programs. Furthermore, they are commonly used to execute programs in certain languages directly in order to avoid a previous compilation step. However, the general interpreter concept is also very useful to model (time-dependent) processes with rule-based simulation logic. In order to execute the simulation logic, the rules can be iteratively applied to the current state that contains state-dependent information. After a desired number of steps the resulting state contains the desired simulation information that was updated during all iterations. A slightly different use case applies all rules iteratively until a fixed-point is reached and the simulation has reached its final state. Well known examples are real-time position-based simulations of cloth or fluids, for instance [15].

Regardless of the exact use case, a high-level interpreter is always required. Its purpose is to find rules that can be applied. It further acts as a driver that performs the iterative application of all rules and specifies the execution order. A common problem that always arises in this context is to determine all active rules that can be applied to a certain state. Active in this scope means that the precondition of a rule evaluates to true with respect to the current state. After distinguishing active from inactive rules, the active ones will be executed afterwards while the inactive ones will be skipped. However, in general it is not possible to decide when a rule becomes active beforehand. Consequently, it is necessary to perform this step every time during an interpreter execution, which is critical for the overall run-time performance.

From a theoretical point of view, this is a well researched task that appears to be straight forward. However, applying theory to practice reveals that a high-performance implementation allowing for an efficient execution of such rules can become very sophisticated. Especially massively parallel architectures like Graphics Proccesing Units (GPUs, or accelerators) consisting of *Single Instruction Multiple Thread* units (*SIMT*, often referred to as *warp* [14,18] or *wavefront* [1]) require specific methods that take the detailed hardware specifications into account. Hence, the usual approach is modeling of domain-specific interpreter algorithms that are problem- and hardware-specific. Such methods are typically hand crafted and manually tuned for a specific target platform. An alternative approach is the use of compilers that transform the underlying interpreter logic to improve performance [13].

In many use cases, it is sufficient to apply the rules to one global context (also referred to as a *state*). A state contains all required environmental variable bindings that are necessary for interpretation. In the field of compiler construction, a state contains all variable→value bindings in the current context. However, in other domains (e.g. in the field of heuristic search or optimization), it is not sufficient to investigate a single state at a time [10,11]. These states can be tracked in parallel and the interpreter has to be adjusted

in a way that it can be applied to several states simultaneously. This leads to additional challenges with respect to interpreter development in general.

In this paper, we present a new generic rule-based interpreter execution algorithm that is specially designed for GPUs. It is optimized for tracking multiple states in parallel per thread group while leveraging a specially designed memory layout to improve coalescing. Furthermore, it uses *thread compaction* to enhance the overall occupancy. Our proposed approach performs significantly faster on our evaluation scenarios than existing generic state-of-the-art interpreters for rule-based execution on GPUs.

2 Related Work

There are many papers about thread compaction and parallel simulations that follow the principle of rule-based interpreters and their applications to a huge variety of different domains. We present a selection of related papers from different research fields that use similar principles to improve performance.

Fung et al. [4] use the concept of thread compaction to avoid overhead in divergent control flow on GPUs. They propose an algorithm that automatically compacts threads at the hardware level and show that this approach improves the overall run time of a program. Rhu et al. [20] use permutations to reorder active threads to a new position in order to maximize utilization of GPU computing resources. In contrast to the previous approach, they use a software-based concept that allows for applications to different domains.

The same principle of thread compaction can also be used to perform parallel stream compaction of data streams. For instance, Billeter et al. [2] proposed a stream-compaction algorithm based on prefix sums and hardware-specific operations in CUDA that uses the same concept. They use a prefix sum to compute the target indices for all elements that remain in the stream. The major conceptual difference with respect to thread compaction is the fact that data elements are compacted instead of logical execution threads. Hoberock et al. [6] use a similar method for deferred shading that also relies on prefix sums in global memory to reduce divergences during rendering. Hughes et al. [7] also leverage stream compaction to enhance performance in the field of data visualization. However, they do not require multiple kernel launches to perform the compaction step. Instead, they perform compaction inside every thread group while incrementing a counter in global memory via atomic operations in order to resolve memory locations for active elements and move them to the final position. In the field of computer graphics, Wald [22] uses tiled compaction in shared memory on GPUs to improve performance of path tracing. He compares the implementation to naive kernels without this strategy and shows performance improvements using thread compaction.

Moving on to simulations, Mueller et al. [16] describe the modeling of particle-based (position-based) simulations using parallel domain-specific constraint solving on GPUs. They use several high-level routines to apply different constraint-solving logic for every constraint consecutively. The same simula-

tion principle is also used in the field of SPH (smoothed particle hydrodynamics) based iterative solvers like the ones from Macklin et al. [15], Kelager [9] and Köster et al. [12]. However, none of them uses thread compaction during processing since they typically do not face a significant amount of thread divergences in their domain.

Parallel constraint propagation is similar to the constraint-solving approaches above. Although this is not the primary topic of this paper, it is highly related since it often follows similar principles. From a high-level point of view, constraint solvers try to narrow the search space during the propagation step by iteratively updating the constraint network/problem, which can be seen as an interpreter execution based on several steps. Ruiz-Andino et al. [21] and Granvilliers et al. [5] propose parallel propagation algorithms for several constraint types. A GPU capable algorithm is the one by Campeotto et al. [3]. It uses specialized CUDA kernels to realize constraint propagation in the scope of multiple kernels. To best of our knowledge, none of the currently available approaches combines constraint propagation with thread compaction or specialized memory layouts.

3 Rule-Based Interpreter Execution in Theory and Practice

From a theoretical point of view, a common technique to model interpreters is the definition of operational semantics in the form of inference rules. Thereby, a rule R_i is defined via

$$R_i = \frac{\Gamma \vdash \mathrm{Pre}_1 \ldots \Gamma \vdash \mathrm{Pre}_n}{\Gamma \vdash \mathrm{Con}},$$

where Pre_j ($j \in [1, \ldots, n]$) are the different preconditions that have to be fulfilled in the scope of the current context (or state) Γ. Con refers to the consequence after applying this rule. For more information about inference rules and operational semantics we refer to [19].

Applying these rules iteratively using a context Γ results in a derivation tree that represents the program execution and an updated context Γ'. As previously mentioned, a whole program execution can be derived by applying all rules iteratively until there is no additional rule to apply: once a rule becomes active (its preconditions were not fulfilled before), a change has been detected. The same holds true for the opposite direction: A rule was active before and does not hold any more. This process is repeated as long as there is a change regarding the active/inactive state of any rule R_i:

$$\{R_1, \ldots, R_n\}_\Gamma \Rightarrow \{R_1, \ldots, R_n\}_{\Gamma^1} \Rightarrow \ldots \Rightarrow * = \{R_1, \ldots, R_n\}_{\Gamma^k}.$$

Otherwise, the context does not change anymore, the program has terminated (or the fixed-point is reached) and the final context Γ^k is available. From a practical point of view, these rules can be modeled via imperative code fragments that check all preconditions via an *if-statement*. Furthermore, this precondition is typically checked against many object instances within a single

context. Thereby, an instance usually refers to a variable binding in a single state. The instances, for which the general precondition evaluates to true, will then be modified by the logic of the conclusion statements. We refer to the number of possible instances to iterate over by the *value range* of R_i. In other words, the value range can be seen as the number of variables to iterate over. We apply the rule R_i to every variable and check whether the preconditions are met. In real-world implementations, we iterate over all integer values $\in [0, \ldots, \text{value range} - 1]$ and map the current index to memory locations of the target object instance. Note that a rule is considered to work on its current object instance, which gives the opportunity to parallelize rule applications to different objects. This domain knowledge makes a compiler parallelization analysis unnecessary.

Common implementations that leverage the processing power of GPUs apply one rule after another in the scope of distinct kernels. This allows us to generate specialized kernels that are instantiated for every rule. Different rules are applied consecutively. This establishes device-wide kernel synchronization and avoids race conditions between different rules, which may work on the same object instances. Algorithm 1 applies a rule in the context of multiple states. Every thread group handles a single state and performs a group-stride blocked loop over the value range of the first rule R_1 in the state s. Within the loop, every index i is checked against the preconditions of R_1. If the conditions evaluate to true, the consequence will be executed and applied to the state. In many current systems that rely on a single state, this algorithm is slightly modified: Instead of multiple thread groups that work on distinct states, they use a grid-stride loop that iterates over the whole value range.

Algorithm 1: Simple execution algorithm for multiple states

```
  Input: state s
  /* Perform a blocked loop over the value range of R_1            */
1 for i := group index; i < value range of R_1; i += group size do
      /* Check rule precondition                                    */
2     if Condition(s, i) then
          /* Evaluate rule                                          */
3         Evaluate(s, i);
4     end
5 end
```

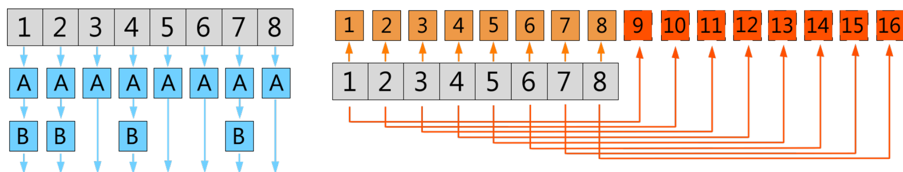
In the presence of multiple states we can execute multiple rules one after another in the scope of a single kernel (Algorithm 2). We can use group-wide synchronization primitives on all currently available major accelerators [1,18]: After execution of the first rule R_1 we have to wait for all other threads to reach this point (Line 6). This ensures that all changes that were made on the state (that resides in global memory) are visible to all other threads in the group. We can now execute the next rule R_2. Note that we can still generate specialized loops as we can inline all required functionality of all involved rules.

Algorithm 2: Simple execution algorithm for multiple states & rules

```
   Input: state information
   /* Perform a blocked loop over the value range of R_1                */
 1 for i := group index; i < value range of R_1; i += group size do
      /* Check rule precondition                                        */
 2    if R_1.Condition(i) then
         /* Evaluate rule                                               */
 3       R_1.Evaluate(i);
 4    end
 5 end
   /* Wait for all changes of the previous rule type R_1                */
 6 group barrier;
   /* Perform a blocked loop over the value range of R_2                */
 7 for i := group index; i < value range of R_2; i += group size do
      /* Check rule precondition                                        */
 8    if R_2.Condition(i) then
         /* Evaluate rule                                               */
 9       R_2.Evaluate(i);
10    end
11 end
   /* Evaluate additional rules...                                      */
12 ...
```

Figure 1 visualizes possible thread divergences and the memory access pattern of both algorithms. The *if-statement* inside the loop can lead to severe control-flow divergences if the preconditions lead to different comparison results on distinct object instances. Consider the simplified control-flow graph of the *if-statement* consisting of the precondition block A and the consequence block B. All threads start by executing the first block A; however, some of them pass the precondition tests and jump to block B. If these threads are executed in the same warp, the other threads have to wait until block B has been executed. This results in a loss of processing resources since these lanes (threads within a warp) cannot perform other operations as long as a lane is executing block B. Afterwards, all threads continue with the outer-most loop. This can also lead to thread divergences with respect to the whole group: some threads have already finished the iteration over the value range, whereas others need to perform additional rule application steps (see Section 4).

Fig. 1 General thread divergences (left) and memory accesses (right) of Algorithm 1 and Algorithm 2. Gray boxes represent threads inside the group. Blue arrows and boxes indicate control flow. Orange arrows and boxes represent global memory accesses during the first iteration of a single value-range loop. The red counterparts represent the accesses in the second iteration of a loop.



Regarding memory operations, the data layout can be chosen in such a way that all threads perform coalesced memory accesses, which is highly important for performance reasons. For example, the different values within R_i's value range are mapped to coalesced memory addresses in global memory. Furthermore, every state has its own contiguous memory region where all required information is stored. This also holds true for further iterations of the outer-most loop, which ensures efficient memory operations across all threads in a group. Since multiple groups are launched that process different states in parallel, the memory latency can be easily hidden by scheduling other groups in favor of the currently active one on a multiprocessor.

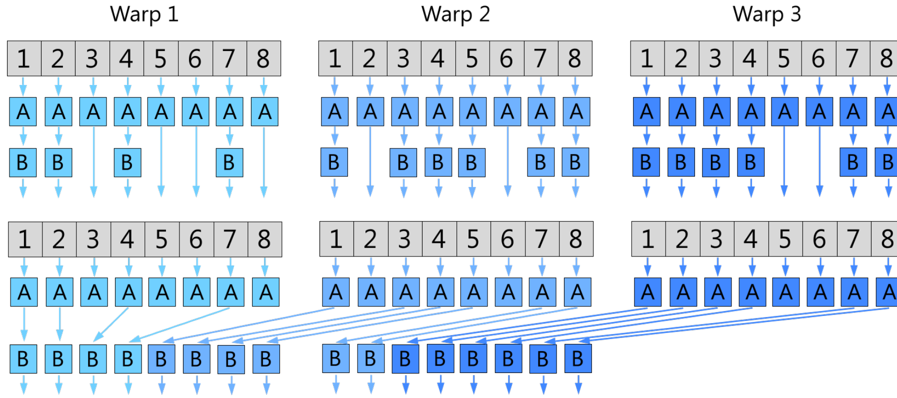
4 Massively Parallel Rule-Based Interpreter Execution

A well known method to improve the overall occupancy of the simple and straight-forward approaches (see Section 3) is *thread compaction*. Figure 2 visualizes the basic principle [4, 20]. Assume that some threads in the group have thread divergences with respect to our rule preconditions. If we leave everything untouched, this will lead to three active warps that will have to be scheduled by the warp dispatcher [1]. We can "remove" the inactive threads by "moving" the active threads to other warps in order to fill all gaps (compaction step). As shown in the figure, this can lead to fully inactive warps (containing no active threads) that do not have to be executed and can be replaced by other active ones from different groups by the warp dispatcher. This results in an increased occupancy of all units. Moving threads around works by moving the associated values (from the value range) to another parent thread in our scenario.

Applying compaction to our domain solves nearly all divergency issues regarding occupancy. This is due to the fact that most rules we have seen in practice rely on uniform control-flow inside the consequence code. From a practical point of view, an additional problem is related to the outer-most loops in Algorithm 1 and Algorithm 2. Too many threads in comparison to the variable range will sacrifice performance: Either many threads will not perform any work since they are out of range or the number of iterations is too small to achieve an overall reasonable workload per thread. Furthermore, large-scale problems involving many parallel states typically consume a lot of memory per state. This often leads to a lower number of states that fit into memory than the globally available number of threads on a GPU [11]. Hence, a single thread per state cannot be used without wasting performance. In our experience, all value ranges are relatively small (< 4096) compared to the overall state size in such scenarios, since more sophisticated problems use many rules that iterate over different parts of the state. Consequently, it is not beneficial to launch a single group per state consisting of many threads as the previously presented

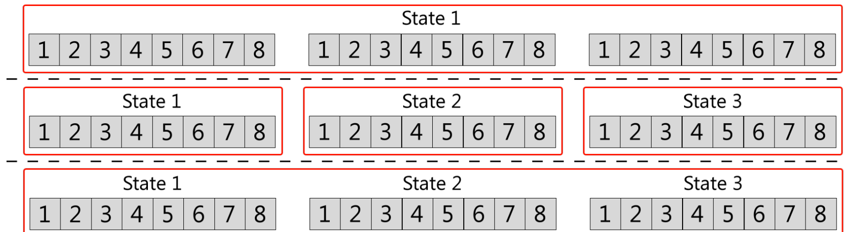
[1] A single SIMT unit will be referred to as a *warp* in the scope of this paper. Furthermore, every warp will be visualized with the help of eight lanes (threads). Groups will be visualized with the help of three warps.

Fig. 2 The principle of thread compaction [4,20]. Top: some threads have divergences (they do not execute the *B* block) and have to wait for the other threads in the scope of the same warp. Bottom: threads after the compaction step. Color coding: the different colors indicate the origin warps the threads were originally from.



algorithms do. As explained above, this would result in either many large inefficient thread groups or many small ones that will significantly increase scheduling/dispatching overhead on the device (Figure 3, middle).

Fig. 3 Different approaches to process multiple states in terms of thread groups. The red borders indicate the launched thread groups. Top: simple approach (see Section 3); one large thread group for each state consisting of multiple warps. Middle: simple approach adapted to small variable ranges (see Section 3); one small thread group for every state consisting of a single warp. Bottom: our approach; large thread groups with logical sub groups for every state; each warp processes a single state while belonging to a logical thread group that works on multiple states in parallel.



Instead, we propose large thread groups that process different states in parallel (Figure 3, bottom). This reduces the number of launched groups and allows to leverage thread compaction across multiple states to improve efficiency. The downside of this approach is that coalesced memory accesses will become more sophisticated, as discussed in Section 3. Reconsider the memory layout of the presented simple algorithms (Algorithm 1 and Algorithm 2) in the presence of multiple states (Figure 4). It is straight forward to map individual value ranges to coalesced memory addresses within a single state.

Other groups access different memory locations that are coalesced with respect to their states. The device scheduler will hide the occurring latencies for all memory accesses to the first elements of every state. If we apply the same memory layout to our proposed processing model, this will lead to coalesced memory accesses in every warp (Figure 5). However, the memory accesses do not target contiguous locations with respect to the whole group. Although the device dispatcher can hide much of the latency in many cases, this does not result in peak performance.

Fig. 4 Memory access pattern of the algorithms 1 and 2 in the scope of multiple states. Several warps within every thread group access coalesced memory locations in the context of the group.

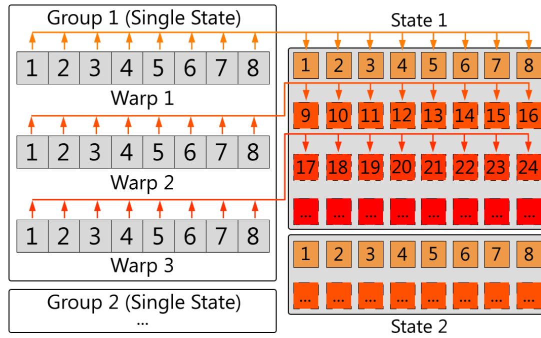
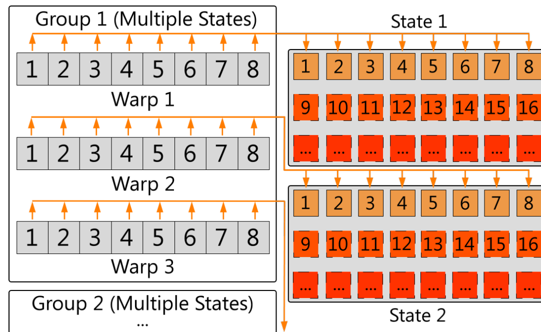
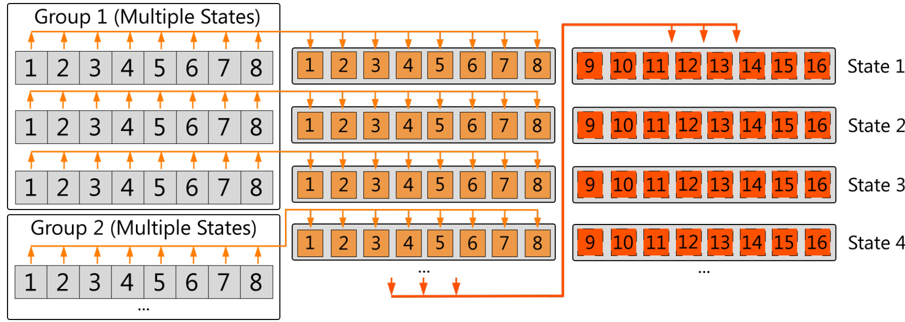


Fig. 5 Memory access pattern of our approach using the memory layout from the algorithms 1 and 2. Every warp performs coalesced memory accesses. However, they are not coalesced with respect to all other warps in the same group.



The optimal memory layout for our approach can be found in Figure 6. We use an interleaved layout that stores state-dependent information with a blocking of the size of a single warp. In other words, all data that is accessed by all threads in every warp in one single value-range iteration is stored

Fig. 6 Coalesced memory access pattern of our approach using an adapted memory layout. All threads in the same group access coalesced memory locations although they belong to different states.



contiguously in memory with a stride of the number of states. This involves additional overhead during address computations which require integer modulo and division operations. However in our experience, the benefit of coalesced memory accesses often outperforms the overhead of the additional arithmetic instructions.

Algorithm 3 shows our approach in pseudo-code which represents a single GPU kernel. The term *thread index* refers to the index of the i-th thread inside the group and *group index* refers to the index of the j-th group. It is designed in a way that it can be directly converted into application code with minor adjustments (see Section 5). In the scope of this algorithm, we process a single state per warp, which results in 32 states per group ($= 32 \cdot$ warp size $= 32 \cdot 32 =$ 1024 threads) on current Nvidia GPUs [18].

First, we initialize a 32 bit integer per thread to store tuples of the current value index and the state index (both treated as 16 bit integers, Line 1) [2]. Second, we compute the source state index this thread belongs to and the initial value index to try with this thread (lines 2–3). We then perform the actual iteration over all values in the range of our first rule R_1. Note that this range is identical in all states, and thus, cannot lead to thread divergences at this level. Divergent control flow would be problematic at this point as some threads of the group would not participate in the group-wide barriers later on. Line 8 performs the actual precondition check of R_1 that evaluates either to true ($= 1$) or to false ($= 0$). We refer to threads that have passed the precondition check as *enabled* threads.

We follow the commonly used prefix-sum based approach to perform thread compaction. For this reason, we first compute a group-wide prefix sum based on the result of the precondition check (Line 9, see Section 5) that uses the previously allocated shared memory. The prefix sum returns the *offset* of the current thread within the group and the group-wide *thread offset*. The latter refers to the total number of threads in this group that are enabled (a side product of the prefix-sum evaluation). Afterwards, every enabled thread writes

[2] This perfectly fits to the common bank-size configuration of 4 bytes on common GPUs.

its current value index and its state index to the computed location in shared memory (Lines 10–12). Note that we need a group barrier at this point to ensure that all updates of shared memory are visible to all threads in the group (Line 13). If the current thread will be active after thread compaction (its thread index is smaller than the total number of active threads after compaction), the current index and the state index will be resolved from shared memory (Lines 14–17). Next, we evaluate the actual rule R_1 with the updated thread information. Finally, we update our next index by adding the current warp size to it (the number of threads per state) and wait for all threads to reach this point. This is important to ensure that the values loaded from shared memory will not be overwritten by other threads in the group that already perform the next iteration. We process further rules in the same way by using additional loops for every rule R_i.

Algorithm 3: Our Execution Algorithm

```
   Input: state information, #states per group
1  shared := shared memory int[group size];
2  sourceStateIndex := group index * group size /
3      #states per group + thread index/ warp size;
4  nextIndex := thread index % warp size;
   /* Apply rule R1                                                   */
5  while nextIndex < value range of R1 do
6      currentIndex := nextIndex;
7      stateIndex := sourceStateIndex;
       /* Check rule precondition                                     */
8      enabled = R1.Condition(currentIndex);
       /* Compute prefix sum, as shown in Algorithm 4                 */
9      (offset, threadOffset) := prefix sum (enabled, shared);
       /* Perform compaction in shared memory                         */
10     if enabled then
11         shared[offset - 1] := (currentIndex, stateIndex);
12     end
13     group barrier;
       /* If we are an active thread...                               */
14     if thread index < threadOffset then
           /* Extract state and value index from shared memory        */
15         (currentIndex, stateIndex) := shared[thread index];
           /* Evaluate rule with the appropriate state and value index */
16         R1.Evaluate(currentIndex, stateIndex);
17     end
18     nextIndex := nextIndex + warp size;
19     group barrier;
20 end
   /* Evaluate additional rules...                                    */
21 ...
```

5 Implementation Details

We have implemented our approach in C++ using CUDA for all GPU computing tasks. In our case, algorithms 3 and 4 are fused to form specifically tuned GPU kernels. We leverage variadic template specialization to instantiate different loops for every rule. Moreover, we execute every rule after each other in a sequential manner to ensure that side effects from previous rules are visible to the upcoming ones.

Algorithm 4 shows further implementation details of the prefix-sum computation. We follow a well known approach that uses warp-shuffle operations to compute a prefix-sum in every warp (Line 1) [17]. Afterwards, the last lane of every warp writes its own prefix-sum value into shared memory (lines 2–4). Next, the first warp uses the same shuffle operations to compute all prefixes for every warp within the group. We have to adjust all offsets of every warp except the first one since their left boundary values have been (potentially) updated (Lines 10–12). Finally, we can resolve the required value for the *thread offset* (see Section 4) by loading it from shared memory (Line 13). The last barrier ensures that no other updates of the shared memory happen in between. Note that this implementation assumes that the total `group size` is less or equal to `warp size * warp size`.

Algorithm 4: Implementation details of the prefix-sum computation

```
   Input: initialValue, shared
   Output: offset, threadOffset
 1 offset := WarpPrefixSum (initialValue);
   /* Only one lane of a warp writes to shared memory                 */
 2 if LaneId + 1 == warp size then
 3 |  shared[warp index] = offset;
 4 end
 5 group barrier;
   /* Complete prefix sum in the first warp                           */
 6 if warp index == 0 then
 7 |  shared[warp index] := WarpPrefixSum (shared[warp index]);
 8 end
 9 group barrier;
   /* Read results from shared memory                                 */
10 if warp index > 0 then
11 |  offset := offset + shared[warp index - 1];
12 end
13 threadOffset := shared[warp size - 1];
14 group barrier;
```

Load	# State	Value Range	ϕ	Layout	1080 Ti	σ	Titan X	σ
20	2048	1024	2	A	0.36	0.03	0.39	0.07
*	*	*	*	B	1.65	0.15	1.38	0.15
*	*	*	*	C	0.35	0.03	0.56	0.11
*	*	*	3	A	0.33	0.04	0.42	0.07
*	*	*	*	B	1.64	0.16	1.28	0.12
*	*	*	*	C	0.35	0.04	0.5	0.05
*	*	4096	2	A	1.47	0.07	1.33	0.13
*	*	*	*	B	10.90	0.26	14.21	0.97
*	*	*	*	C	1.38	0.07	1.5	0.14
*	*	*	3	A	1.47	0.07	1.58	0.14
*	*	*	*	B	10.96	0.24	13.98	0.89
*	*	*	*	C	1.38	0.07	1.54	0.15
*	16384	1024	2	A	2.28	0.13	2.73	0.20
*	*	*	*	B	52.08	0.72	42.37	4.21
*	*	*	*	C	2.27	0.14	2.72	0.21
*	*	*	3	A	2.27	0.14	2.72	0.20
*	*	*	*	B	52.32	0.7	42.48	3.25
*	*	*	*	C	2.27	0.13	2.69	0.22
*	*	4096	2	A	8.88	0.49	9.71	0.74
*	*	*	*	B	241.38	1.93	513.08	33.89
*	*	*	*	C	8.63	0.49	9.59	0.73
*	*	*	3	A	8.85	0.49	9.69	0.74
*	*	*	*	B	243.75	1.32	500.2	38.63
*	*	*	*	C	8.61	0.47	9.58	0.73

Table 1 Influence of the number of states, value range, divergency rate and memory layout on the overall run time in ms with fixed load using Algorithm 1.

Load	# State	Value Range	ϕ	Layout	1080 Ti	σ	Titan X	σ
200	2048	1024	2	A	0.34	0.04	0.49	0.05
*	*	*	*	B	1.43	0.23	1.35	0.12
*	*	*	*	C	0.34	0.04	0.57	0.04
2000	*	*	*	A	2.06	0.23	3.24	0.12
*	*	*	*	B	2.52	0.06	3.81	0.11
*	*	*	*	C	2.06	0.05	3.26	0.11
200	16384	*	*	A	3.06	0.15	4.11	0.22
*	*	*	*	B	51.62	0.57	37.15	5.32
*	*	*	*	C	2.97	0.14	3.98	0.21
2000	*	*	*	A	12.5	1.21	22.19	0.98
*	*	*	*	B	50.85	0.73	36.97	0.73
*	*	*	*	C	11.64	1.1	20.78	0.61

Table 2 Influence of the load, number of states, and memory layout on the overall run time in ms with fixed value range and divergency rate using Algorithm 1.

6 Evaluation

The evaluation section focuses on general performance improvements that can be achieved using the presented method in combination with our proposed memory layout. Therefore, we have evaluated the performance of our Algorithm 3 in comparison to the straight-forward Algorithm 1 that is commonly used (see Section 3). We abstract from detailed rule implementations with the help of an artificially designed one. Since rule-based interpreters are nowadays commonly used to model simulation-like behavior in the scope of optimization

Load	# State	Value Range	ϕ	Layout	1080 Ti	σ	Titan X	σ
20	2048	1024	2	A	0.21	0.13	0.19	0.004
*	*	*	*	B	0.35	0.03	0.54	0.02
*	*	*	*	C	0.20	0.02	0.20	0.01
*	*	*	3	A	0.21	0.21	0.19	0.01
*	*	*	*	B	0.35	0.02	0.50	0.03
*	*	*	*	C	0.21	0.03	0.19	0.008
*	*	4096	2	A	0.47	0.12	0.69	0.03
*	*	*	*	B	1.01	0.03	1.45	0.1
*	*	*	*	C	0.43	0.03	0.62	0.04
*	*	*	3	A	0.5	0.20	0.71	0.03
*	*	*	*	B	0.99	0.05	1.60	0.11
*	*	*	*	C	0.43	0.03	0.63	0.02
*	16384	1024	2	A	0.67	0.17	0.94	0.06
*	*	*	*	B	2.42	0.03	5.71	0.11
*	*	*	*	C	0.66	0.02	0.98	0.07
*	*	*	3	A	0.67	0.21	1.07	0.07
*	*	*	*	B	2.38	0.02	5.76	0.1
*	*	*	*	C	0.67	0.02	0.87	0.02
*	*	4096	2	A	2.53	0.14	3.16	0.09
*	*	*	*	B	8.33	0.06	20.29	0.19
*	*	*	*	C	2.45	0.03	3.12	0.08
*	*	*	3	A	2.56	0.17	2.93	0.11
*	*	*	*	B	8.14	0.07	20.91	0.15
*	*	*	*	C	2.45	0.02	3.11	0.09

Table 3 Influence of the number of states, value range, divergency rate and memory layout on the overall run time in ms with fixed load using Algorithm 3.

Load	# State	Value Range	ϕ	Layout	1080 Ti	σ	Titan X	σ
200	2048	1024	2	A	0.33	0.23	0.48	0.03
*	*	*	*	B	0.36	0.02	0.51	0.02
*	*	*	*	C	0.33	0.02	0.57	0.04
2000	*	*	*	A	2.03	0.25	2.95	0.11
*	*	*	*	B	1.42	0.03	2.54	0.11
*	*	*	*	C	2.06	0.02	3.08	0.07
200	16384	*	*	A	0.96	0.19	1.77	0.11
*	*	*	*	B	2.42	0.02	5.73	0.09
*	*	*	*	C	0.99	0.12	1.94	0.1
2000	*	*	*	A	7.06	0.19	13.87	0091
*	*	*	*	B	6.55	0.02	13.01	0.08
*	*	*	*	C	7.27	0.03	13.63	0.07

Table 4 Influence of the load, number of states, and memory layout on the overall run time in ms with fixed value range and divergency rate using Algorithm 3.

problems [11], they often have to execute machine-learning models. These are used to realize heuristics that guide the surrounding optimization system [10]. Therefore, our actual implementation of a rule consists of matrix-matrix multiplications to mimic the execution of such a machine-learning model:

$$A \times B, \tag{1}$$

where $A \in R^{M \times N}$ and $B \in R^{N \times O}$. This computation requires $2MNO - MO$ floating-point operations in our evaluation benchmarks [8,11]. We have introduced a loop to perform multiple matrix-matrix multiplications in the same

rule. Thereby, the *load* factor refers to this number of iterations to simulate more and less computationally expensive interpreter rules. This allows us to make a statement about the general scaling behavior in terms of computational effort.

We used two GPUs from NVIDIA (a GeForce GTX Titan X and a GeForce GTX 1080 Ti) for all benchmarks. Every performance measurement is the median execution time in milliseconds of 100 algorithm executions. The CUDA code was compiled with *nvcc v10.1.105* with all compiler optimizations enabled. We refer to the presented memory layouts as follows:

- **A** corresponds to the default memory layout from Figure 4,
- **B** corresponds to the transposed memory layout of **A**,
- **C** corresponds to the optimal memory layout for our method from Figure 6.

We have included the transposed layout **B** to demonstrate the effect of using a non-coalesced memory layout, which leads to the worst performance.

Tables 1 and 3 show the main evaluation results of the simple algorithm and our approach. They demonstrate the impact of the number of states, the value range, the *divergency rate* (ϕ) and the memory layout on the overall run time. We start with a value range of 1024 in order to have a reasonable number of values per rule. Furthermore, ϕ describes that every ϕ-th thread will have a divergence on average. We initialize every value in every state with a random integer number $\in [0, \ldots \phi - 1]$, where 0 means this thread will not pass the precondition checks. We chose ϕ to be either 2 or 3 to reflect our real-world experience regarding typical divergencies that often occur in optimization problems [11]. In this case, the load was set to 20 in order to have a reasonably expensive approximation of a rule performing some computations.

The simple algorithm scales linearly with respect to the value range and the number of states. As expected, the worst memory layout is layout B. Memory layout C performs slightly faster in most of the cases, even though layout A should be fastest for this algorithm. Our approach scales much better: increasing the value range by $4\times$ while having a small number of states yields $\approx 2\times$ to $\approx 3.5\times$ run-time increase. Fixing the value range and increasing the number of states by $8\times$ will increase the run time by $\approx 3\times$ to $\approx 5\times$. Memory layout C performs fastest in all cases on the more recent GPU architecture. This can increase performance by additional 14% compared to layout A. A change of ϕ does not produce significantly different results for both algorithms. However, our approach performs up to $\approx 4\times$ faster regarding all evaluation cases.

Tables 2 and 4 visualize the impact of the load, the number of states and the memory layout, while fixing the value range and ϕ. Increasing the load by a factor of $10\times$ leads to an increase of the run time of up to $\approx 7\times$ for both algorithms.

7 Conclusion

We present a new approach (Algorithm 3 and Algorithm 4) to realize generic rule-based interpreters for many states on GPUs. In order to achieve high occupancy without creating too much dispatcher overhead, we execute multiple states within the same group. Thread compaction improves efficiency even across multiple states within the same group. A specially chosen memory layout (Figure 6) ensures coalesced memory accesses within each group and results in better performance.

Our approach scales very well with the value range and the number of states. For instance, an execution of an interpreter on 2048 states, in which every rule fails with a probability of 33%, requires only 0.21 milliseconds on current hardware to complete. Comparing the performance to existing straightforward implementations on GPUs yields significant speedups of up to $\approx 4\times$. Furthermore, using our proposed memory layout improves performance by up to 14% compared to a thread-compaction based implementation with a non-optimal memory layout.

Probably the main disadvantage of our method is the address-computation overhead. Multiple additional arithmetic instructions are required to compute the final memory address. Although we have not seen any major degradation in terms of performance, it may be possible that rules using many loads and stores might be negatively affected.

In the future we would like to relax the association of a single warp to a single state. This provides the opportunity to have a variable number of states per group depending on the actual state information. In addition, we want to experiment with locally cached state information in shared memory and applications to multiple GPUs in order to improve rule evaluation in general.

Acknowledgements The authors would like to thank Wladimir Panfilenko and Thomas Schmeyer for their suggestions and feedback regarding our method. Furthermore, we would like to thank Gian-Luca Kiefer for additional feedback on the paper. This work was funded by the German Ministry of Education and Research: Project Hybr-iT: Hybrid and Intelligent Human-Robot Collaboration (grant number 01IS16026A).

References

1. AMD: AMD Vega Instruction Set Architecture (2019)
2. Billeter, M., Olsson, O., Assarsson, U.: Efficient stream compaction on wide simd many-core architectures. In: Proceedings of the Conference on High Performance Graphics 2009, HPG '09, p. 159–166. Association for Computing Machinery, New York, NY, USA (2009)
3. Campeotto, F., Dal Palù, A., Dovier, A., Fioretto, F., Pontelli, E.: Exploring the use of gpus in constraint solving. In: M. Flatt, H.F. Guo (eds.) Practical Aspects of Declarative Languages, pp. 152–167. Springer International Publishing, Cham (2014)
4. Fung, W.W.L., Aamodt, T.M.: Thread block compaction for efficient simt control flow. In: 2011 IEEE 17th International Symposium on High Performance Computer Architecture, pp. 25–36 (2011)
5. Granvilliers, L., Hains, G.: A conservative scheme for parallel interval narrowing. Information Processing Letters **74**, 141–146 (2000)
6. Hoberock, J., lu, V., Jia, Y., Hart, J.: Stream compaction for deferred shading. pp. 173–180 (2009)
7. Hughes, D.M., Lim, I.S., Jones, M.W., Knoll, A., Spencer, B.: Ink-compact: In-kernel stream compaction and its application to multi-kernel data visualization on general-purpose gpus. Comput. Graph. Forum **32**(6), 178–188 (2013)
8. Hunger, R.: Floating Point Operations in Matrix-Vector Calculus. Tech. rep., Technische Universität München (2007)
9. Kelager, M.: Lagrangian fluid dynamics using smoothed particle hydrodynamics. Camb. Monogr. Mech. **77** (2006)
10. Köster, M., Groß, J., Krüger, A.: Fang: Fast and efficient successor-state generation for heuristic optimization on gpus. In: International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP). Springer (2019)
11. Köster, M., Groß, J., Krüger, A.: Parallel tracking and reconstruction of states in heuristic optimization systems on gpus. In: International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT-2019). IEEE (2019)
12. Köster, M., Krüger, A.: Adaptive position-based fluids: Improving performance of fluid simulations for real-time applications. International Journal of Computer Graphics and Animation **6**, 01–16 (2016)
13. Köster, M., Leißa, R., Hack, S., Membarth, R., Slusallek, P.: Code refinement of stencil codes. Parallel Processing Letters **24**, 1441003 (2014)
14. Lustig, D., Sahasrabuddhe, S., Giroux, O.: A formal analysis of the nvidia ptx memory consistency model. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19, p. 257–270. Association for Computing Machinery, New York, NY, USA (2019)
15. Macklin, M., Müller, M.: Position based fluids. ACM Transactions on Graphics **32**, 104:1– (2013)
16. Müller, M., Heidelberger, B., Hennix, M., Ratcliff, J.: Position based dynamics. J. Vis. Comun. Image Represent. **18**(2), 109–118 (2007)
17. NVIDIA: Faster Parallel Reductions on Kepler (2014)
18. NVIDIA: CUDA C Programming Guide v10 (2019)
19. Pierce, B.C.: Types and Programming Languages, 1st edn. The MIT Press (2002)
20. Rhu, M., Erez, M.: Maximizing simd resource utilization in gpgpus with simd lane permutation. SIGARCH Comput. Archit. News **41**(3), 356–367 (2013)
21. Ruiz-Andino, A., Araujo, L., Sáenz-Pérez, F., Ruz, J.J.: Parallel arc-consistency for functional constraints. In: Implementation Technology for Programming Languages based on Logic (1998)
22. Wald, I.: Active thread compaction for gpu path tracing. In: Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics, HPG '11, p. 51–58. Association for Computing Machinery, New York, NY, USA (2011)