Parallel Tracking and Reconstruction of States in Heuristic Optimization Systems on GPUs

1st Marcel Köster
Saarland Informatics Campus
Saarbrücken, Germany
firstname.lastname@dfki.de

2nd Julian Groß
Saarland Informatics Campus
Saarbrücken, Germany
firstname.lastname@dfki.de

3rd Antonio Krüger
Saarland Informatics Campus
Saarbrücken, Germany
firstname.lastname@dfki.de

***Abstract*—Modern heuristic optimization systems leverage the parallel processing power of Graphics Processing Units (GPUs). Many states are maintained and evaluated in parallel to improve runtime by orders of magnitudes in comparison to purely CPU-based approaches. A well known example is the parallel Monte Carlo tree search, which is often used in combination with more advanced machine-learning methods these days. However, all approaches require different optimization states in memory to update or manipulate variables and observe their behavior over time. Large real-world problems often require a large number of states that are typically limited by the amount of available memory. This is particularly challenging in cases in which older states (that are not currently being evaluated) are still required for backtracking purposes. In this paper, we propose a new general high-level approach to track and reconstruct states in the scope of heuristic optimization systems on GPUs. Our method has a considerably lower memory consumption compared to traditional approaches and scales well with the complexity of the optimization problem.**

***Index Terms*—Heuristic search, state reconstruction, massively-parallel processing, graphics processing units, GPUs**

I. Introduction

Heuristic optimization is a well researched field in computer science. A prominent example for search-based heuristic optimization algorithms is the Monte Carlo Tree Search (MCTS) [1]. It is particularly popular these days [2] in combination with sophisticated machine-learning models to solve challenging tasks. Over time, many advances have been made to improve performance on the pure algorithms and implementation side. Most recent developments involved the use of Graphics Processing Units (GPUs) to significantly speed up the optimization process. As it is typically straight-forward to parallelize over many different optimization states at the same point in time, it is much more sophisticated to satisfy memory-consumption requirements.

However, search-based algorithms rely on backtracking with respect to the domain-specific optimization problem and goal definitions. These different *states* in memory contain variables that can be selected and assigned to different values. Furthermore, there has to be the ability to jump to a previously expanded state and continue the search in another (unexplored) search direction. In order to realize a random-access capability across visited states that might be interesting for backtracking purposes, these are typically maintained in main memory. Unfortunately, large-scale optimization problems require many distinct variables and state-dependent context information of several kilobytes per state. This makes it even harder to comply with memory constraints in terms of the number of states that can be remembered and processed in parallel.

In this paper, we present a new general high-level approach that is designed for heuristic optimization systems and is GPU friendly. It tracks and reconstructs states on-the-fly using a recovery mechanism and a history that enables us to store state information in a dense and highly compressed manner. This achieves significantly lower memory consumption compared to traditional approaches that require much more detailed state information during the optimization process. We evaluate our concept on different scenarios and show that our method scales well with the complexity of the optimization problem.

II. Related Work

Campeotto et al. [3], [4] propose a framework for parallel constraint solving on GPUs. In their approach, they store state-dependent information on the CPU and leverage the GPU for constraint propagation. Abdelkafi et al. [5] use GPUs to evaluate the neighborhood of a state in parallel. Lam et al. [6] proposes a similar concept, while focusing on simulated annealing. Luong et al. [7], [8] use the GPU to realize a parallelized Tabu Search. Melab et al. [9] extend this work to a generalized framework approach to target several optimization domains. Further work in this area is the paper by Novoa et al. [10]. They realize a parallel heuristic search implementation on GPUs for quadratic assignment problems. Another well researched application is the parallel MCTS, which has been improved by many researchers over time [11]–[13] that used parallel architectures to decrease the overall runtime. Zhou et al. [14] make a comparable advance in this direction on other traditional algorithms like $A^\star$. Most of these papers track their states in CPU memory and copy the required information to the GPU for parallel processing. Alternatively, some related approaches keep all states in GPU memory without the need to copy states from the CPU during optimization. Nonetheless, they do not perform any state recovery compared to our method and suffer from memory-consumption issues on large-scale problems.

Mostly similar to our approach in terms of memory reduction is the approach by Powley et al. [15]. They propose a strategy that is particularly designed for MCTS to reuse previously allocated states that are no longer required. For this recycling purpose, they leverage a pointer-based structure to link and remap nodes. However, this structure is not GPU friendly in any kind due to non-coalesced memory

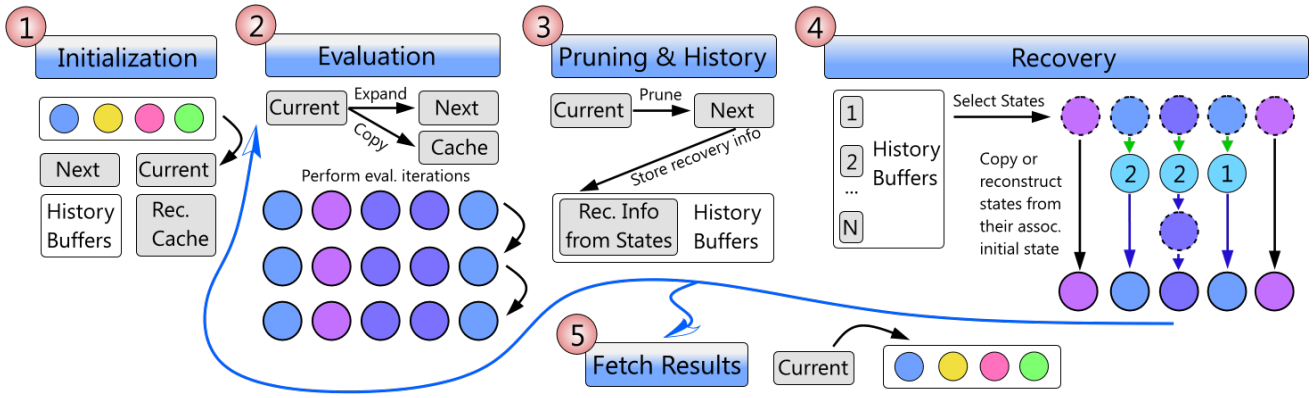

Fig. 1. Our visualized high-level method. In the first step (1), the required GPU and CPU buffers are allocated and the source states are copied from CPU to GPU memory. Next, we enter the main optimizer loop and perform nested evaluation steps (2) before pruning and resolving and storing history-information (3). Note that states will not be moved in memory and keep their associated indices during execution steps. Then, we select potentially interesting states based on history information and perform the recovery step (4). We either continue with the next outer solver loop or break the loop and fetch the final results (5).

accesses [16]. In contrast to their method, we reuse previously allocated memory in a very aggressive way and reconstruct states on demand. Our history approach for state reconstruction is inspired by the basic principle of the *neighborhood fitness structure* from Rashid et al. [17]. We follow their approach to fetch state-dependent information from the GPU into CPU memory. However, we do neither copy any neighborhood information nor detailed variable assignments or values from a state to the GPU. Instead, our method uses only several bytes of information. It includes a source index to refer to its parent state, from which it was constructed, and a successor index to replay previously determined decisions (see subsection III-B).

III. Our Approach

Heuristic optimization problems are commonly solved using tree-based search as shown in Figure 2. Thereby, we differentiate between two essential phases that have to be performed during search and state transitions: execution/simulation and variable assignments. The actual execution logic is realized in the first phase. In the case of round-based games (like chess), this phase realizes the actual rules of the game in form of game logic. More sophisticated rule sets come from real-world use cases which contain many different constraints and complex if-conditions. The second phase performs the assignment of all variables that are either currently not assigned at all or have been decided to require a new assignment during the execution step. This process typically involves (different) domain-specific heuristics that have been trained using ML methods or have been manually designed for a particular purpose. In most cases, this also involves drawing random numbers to decide for a variable or an assignable value. In such scenarios, it is required to remember the original state (seed) of the random-number generator for replaying an identical assignment process (see subsection III-B).

A visualization of our method can be found in Figure 1 and the corresponding algorithm in pseudo-code in Algorithm 1. From a high-level point of view, our approach consists of five major phases. During initialization, we allocate all required buffers in GPU and CPU memory and copy the initial state information to the *current* buffer on the GPU. We use double-buffering to allow parallel writing of state output

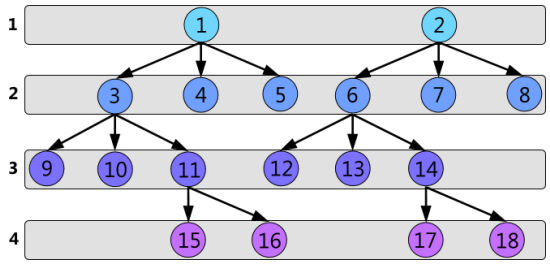

Fig. 2. A simple search tree and its correspondingly simplified view onto different iterations. For the sake of simplicity, every layer of the tree corresponds to a single solver iteration in this example (1-4). However, our method is not limited to such simple cases and also covers more sophisticated scenarios, in which states from different iterations are processed at the same point in time.

information to the *next* buffer during execution and assignment steps. Next, we enter the main optimizer loop that iteratively performs evaluation (exploration), pruning and recovery steps. This process is repeated until the maximum number of solver iterations is reached or another domain-specific break-criterion is encountered. The following sub-sections will give detailed explanations about the different steps and their functionality.

A. Evaluation Phase

The evaluation phase represents an exploratory step (line 6 in Algorithm 1), in which we apply several execution and assignments steps in parallel on all states until we reach the desired evaluation depth (or in other words: the maximum number of evaluation steps). In the scope of this phase, all input states will be expanded to the maximum number of states. Thereby, the expansion strategy is domain and typically problem specific. A common solution that is suitable for most cases is to create $n = \lceil \frac{|S_{max}|}{|S|} \rceil$ many clones of each state in the *next* buffer (see Figure 3). Afterwards, all resulting states will be evaluated step by step to find the best ones during pruning. Finally, both buffers are swapped to have the least-recent updates in the current buffer.

B. History

The recovery information of theses states will be stored in the history buffer to be available for future recovery purposes

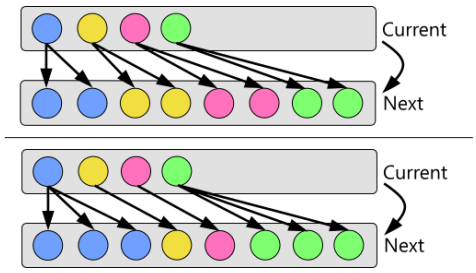

Fig. 3. Different expansion strategies before the actual evaluation phase. Every successor state will be expanded a domain- and problem-specific number of times from the current to the next buffer. Top: a simple regular expansion. Bottom: an irregular expansion pattern based on custom state properties.

(see Figure 4). This information is stored in CPU memory and consists of two unsigned 16 bit integers and a single 32 bit integer. The largest integer is used to store the state rating, which might be updated during the optimization process. One of the 16 bit variables contains the source state-index pointing to the original state index from the previous iteration that this state was resolved from. The other 16 bit variable contains the successor index that represents the current random seed for successor generation during the assignment process. Alternative realizations might require more involved history information. However, according to our experience the random seed and the successor index are typically sufficient for most optimization problems.

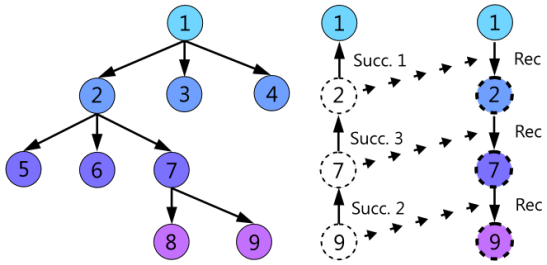

Fig. 4. A sample search tree (left) and an associated history trace for the path $1 \rightarrow 2 \rightarrow 7 \rightarrow 9$ (middle). The history entries contain the required recovery information (successor index from the previous iteration, state rating and random seed). All further state-dependent knowledge is discarded as it can be recovered afterwards (right). Recovery needs an original source state and a history knowledge about this unique trace to successfully recover the desired states.

C. Recovery Phase

During recovery, state information is reconstructed using the same high-level iteration logic that is used during the evaluation phase (see subsection III-A, lines 12-24 in Algorithm 1). However, the major difference between the two phases is the fact that we use previously determined recovery indices to reconstruct the states based on their successor information. Note that this information is stored in CPU memory in our implementation. This can be efficiently implemented, since the amount of memory that has to be transferred is usually around several megabytes and can be copied asynchronously with respect to all other operations.

Every state that has to be recovered is typically restored with the help of an initial state (see Figure 4). We perform all required execution and assignment steps following the

Algorithm 1 High-Level Algorithm

Input: input states on CPU, GPU processing stream
Output: output states on CPU
1: current, next, recover $\Leftarrow$ AllocateOnGPU(#max states);
2: history $\Leftarrow$ AllocateOnCPU();
3: current $\Leftarrow$ CopyToGPU(stream, input);
4: **for** $i = 1$ to #iterations **do**
5: recover $\Leftarrow$ Copy(stream, current);
6: Evaluate(stream, current, next, #eval steps);
7: Prune(stream, current, next);
8: Synchronize(stream);
9: AddHistory(history, current);
▷ Prepare recovery from the fast cache
10: recoveryIndices $\Leftarrow$ ResolveRecovery(history);
11: recovery = recoveryIndices $\setminus$ indices(recover);
12: **for each** index $\in$ recovery **do**
13: Copy(stream, input, index, current);
14: **end for**
▷ Recover states from the beginning
15: **if** $|\text{recovery}| \neq 0$ **then**
16: Recover(stream, current, #recover steps $\cdot(i-1)$);
17: **end if**
18: **for each** index $\in$ recoveryIndices $\cap$ indices(recover) **do**
19: Copy(stream, recover, index, current);
20: **end for**
▷ Perform recovery of current iteration
21: Recover(stream, current, #recover steps);
22: **end for**
23: output $\Leftarrow$ CopyToCPU(stream, current);

previously stored decision information. This does not impose a significant overhead, since the number of states that have to be recovered is typically much smaller than the maximum number of states that has to be processed for exploration purposes. There is an exception to this process: If we have to recover a state from the previous iteration, we will use a fast recovery cache that stores all previous states. This allows us to skip many recovery iterations in most common cases in which we perform a single back-tracking operation only.

D. Fill Rate

Usually, an assignment step is much more expensive than a default execution step. In order to improve the overall runtime, we should try maximize the parallelism in terms of the number of states that require an assignment step. A straight-forward solution would be an immediate assignment step after a single variable in the scope of one state requires an assignment. This works reasonably well for a small number of states. However, if we increase the number of states, the probability that we require a new assignment step increases significantly. Let p be the probability that a single state requires an assignment. Then, the overall probability that we have to break for an assignment step with respect to the number of states is given by:

$$P = 1 - p^{|S|}, \tag{1}$$

where $|S|$ is the number of states. Since the naive approach does not scale well, we have to wait until more states require an assignment process and skip over them in the scope of the execution logic. The *fill rate* F allows to control exactly this intended behavior: it represents the ratio of states that require an assignment and the total number of states that is

required until we have to perform an assignment step (see Figure 5). The higher F is, the more states are required before we perform an assignment run. However, this number is highly domain-specific and depends on the actual work loads of the execution and assignment realizations.

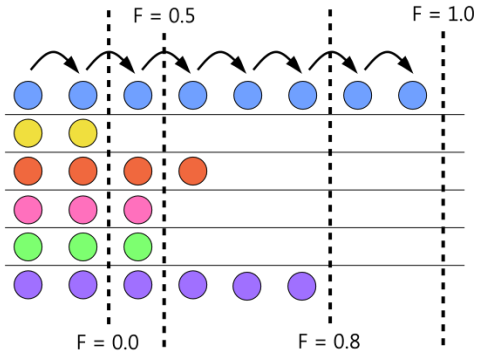

Fig. 5. Different traces of 6 distinct states during evaluation and/or recovery using 7 steps (from left to right). As visualized in the image, state 2 requires an assignment after the first step, whereas the others still require further steps until they need an assignment. Different fill rates are visualized in the diagram using dashed lines. For instance, a fill rate of $F = 0.5$ means that at least half of all states need an assignment before we break the execution loop and perform an expensive assignment step.

E. A Single Iteration

Algorithm 2 shows the main iteration functionality for a single optimizer iteration. Note that the actual execution, as well as all assignment steps, are parallelized across all states on the GPU. Initially, we have to perform an assignment step to ensure that no free (unassigned) variables remain. The actual high-level iteration loop (until we reach the number of desired steps; see section III) is realized with the help of two nested loops. In order to avoid invalid execution steps on states that already wait for an assignment, we use a single bit per state that indicates whether to continue processing or skip the state. This bit-set information is reset to its initial state in which all states require further execution steps in the beginning (line 4). The inner-most loop performs the actual execution until we have found a reasonable number (in terms of the *fill rate*) of states that require an assignment step or we have reached the maximum number of steps. As soon as this loop breaks, we require an additional assignment step and continue processing until we reach the desired number of exploration steps.

Algorithm 2 High-Level Iteration Logic

```
Input: GPU processing stream, current buffer, #steps
 1: Assign(stream, current);
 2: i ⇐ 0;
 3: do
 4:     Reset(stream);
 5:     do
 6:         i ⇐ i + 1;
 7:         #finishedStates = Execute(stream, view);
 8:         if #finishedStates / #maxStates ≥ F then
 9:             break
10:         end if
11:     while i < #steps
12:     Assign(stream, current);
13: while i < #steps
```

F. Memory Consumption

As previously mentioned, traditional methods have to keep all states in memory that are considered in the scope of the optimization system. Let $|S_{max}|$ be the maximum number of the states that can be considered in parallel at a single point in time for evaluation purposes. Further, let $|S_{back}(i)|$ be the maximum number of states that have to be remembered in iteration i to realize the required backtracking functionality. This yields a total memory consumption $M_{trad}(i)$ for i iterations of

$$M_{trad}(i) = 2 \cdot |S_{max}| \cdot m + \sum_{j=1}^{i-1} |S_{back}(j)| \cdot m, \qquad (2)$$

where m is the number of bytes of a single state in memory that includes all required variable assignments and state-dependent context information. The total memory consumption of the backtracking buffer contains all required states of every previous iteration to which we can jump in the case of backtracking. However, in many cases $|S_{back}(i)| \approx |S_{max}|$, which is the case for many heuristic optimization systems that evaluate the neighborhood of every state locally and try to apply pruning methods with respect to parent information of every state.

Our method has a significantly lower memory consumption, which can be further reduced when the fast recovery cache for the last iteration is disabled. This results in a total memory consumption of

$$M_{our}(i) = 3 \cdot |S_{max}| \cdot m + |S_{max}| \cdot i \cdot m_{hist} \qquad (3)$$

bytes, where m_{hist} refers to the size of a single history entry that has to be remembered. Since $m_{hist} << m$ in practice, our method is not primarily dominated by the number of iterations in comparison to existing methods. Consequently, $M_{our}(i) << M_{trad}(i)$ on practical problem instances for which $m >> m_{hist}$ and a reasonably large number of look-ahead iterations i.

IV. Implementation Details

We have implemented our approach and all related methods in C# using the ILGPU-compiler[1] to compile the kernels for NVIDIA GPUs via PTX [18] for all GPU programs. In the scope of the GPU kernels, a single thread is assigned to a single state. We use GPU streams to efficiently schedule the GPU operations from the CPU side. As outlined in the previous section, we have to fetch history values for recovery purposes. These fetching operations require CPU synchronization at two points in the implementation. In order to improve the overall efficiency of our implementation, we heavily rely on the SOA (structure of array) memory layout [16]. Moreover, we realize the copy operations of all states for recovery purposes (lines 10 to 19 in Algorithm 1) using two efficient copy kernels instead of a set of queued copy operations. The actual pruning step is realized on the CPU and requires two additional stream synchronizations. We decided to shift this work to the CPU side, since more involved state comparators can be conveniently realized on the CPU.

[1] www.ilgpu.net

V. Evaluation

For evaluation purposes, we abstract from real-world interpreter and assignment logic to avoid hard-to-reproduce and difficult to understand benchmarks. Since many modern heuristic-based search and optimization systems leverage neural networks for rating possibilities and assigning variables, we model the assignment step via iterative matrix-matrix multiplications. This simulates more- and less-expensive network evaluations during the assignment step. Similarly, the actual interpreter steps are also realized via matrix-matrix multiplications to model computationally-dependent logic:

$$A \in \mathbb{R}^{M \times N} \cdot B \in \mathbb{R}^{N \times O}, \tag{4}$$

which requires $2MNO - MO$ number of floating-point operations in our evaluation benchmark [19]. However, in real-world applications the computation load of the assignment step is typically much larger than a single interpreter step (for instance, a large-scale neural network has to be evaluated). For this reason, we introduce a *load* factor L that represents the overhead of an assignment step over an interpreter step. In other words, if an interpreter step has a computation load of C, then the assignment step will have a load of $C \cdot L$. We choose L to be at least 10, in order to have a realistic baseline. The down side of this approach is the fact that we do either not break for assignments at all or we always break at the same time. This does not reflect reality in any sense: Different states typically break at different points in time. We use distinct probabilities for every state during execution to determine whether to break for an assignment step or not. This probability is referred to as P in the scope of the evaluation, where a larger value indicates a higher probability of all states to break for an assignment. We choose the fill rate F to be larger or equal to 0, where $F = 0$ refers to the case in which we always break for an assignment as soon as a single state requires an assignment step. This simulates traditional approaches in which F is not used. The maximum value is $F = 100$, which causes the assignment component to wait until all states require an assignment step.

L	S	P	F	**1080 Ti**	σ	**Titan X**	σ
10	16384	30%	0%	**16.46**	0.01	**25.19**	0.02
*	*	*	30%	**7.17**	0.05	**10.70**	0.05
*	*	*	60%	**5.61**	0.08	**8.14**	0.02
*	*	*	100%	**5.62**	0.07	**8.16**	0.04
*	*	70%	0%	**16.62**	0.04	**25.23**	0.01
*	*	*	30%	**7.22**	0.01	**10.66**	0.01
*	*	*	60%	**5.65**	0.05	**8.15**	0.07
*	*	*	100%	**5.65**	0.02	**8.12**	0.03
*	65536	30%	0%	**45.23**	0.03	**71.54**	1.76
*	*	*	30%	**17.73**	0.27	**28.05**	0.10
*	*	*	60%	**13.17**	0.28	**19.73**	0.04
*	*	*	100%	**13.14**	0.12	**19.61**	0.05
*	*	70%	0%	**45.17**	0.02	**70.19**	0.01
*	*	*	30%	**17.74**	0.26	**26.84**	0.03
*	*	*	60%	**13.15**	0.27	**19.60**	0.06
*	*	*	100%	**13.16**	0.75	**19.62**	0.04

TABLE I
Influence of the number of states, the assignment probabilities and the fill rate on the overall runtime. GPUs: GeForce GTX 1080 Ti and GeForce GTX Titan X. Time in milliseconds.

We used two GPUs from NVIDIA (a GeForce GTX 1080 Ti and a GeForce GTX Titan X). Every performance measurement is the median execution time of 100 algorithm executions. The number of states S is chosen to be a power of 2 to fill some thread groups on both devices (see section IV).

L	S	P	F	**1080 Ti**	σ	**Titan X**	σ
100	16384	70%	0%	**130.65**	0.02	**207.30**	0.00
*	*	*	30%	**40.49**	0.03	**63.35**	0.02
*	*	*	60%	**25.45**	0.01	**39.36**	0.00
*	*	*	100%	**25.44**	0.00	**39.35**	0.01
1000	*	*	0%	**1270.03**	0.03	**2027.66**	631.9
*	*	*	30%	**372.12**	0.00	**589.45**	0.02
*	*	*	60%	**222.51**	0.01	**349.75**	0.02
*	*	*	100%	**222.47**	0.00	**349.77**	0.03
100	65536	*	0%	**375.29**	0.01	**618.452**	86.8
*	*	*	30%	**104.88**	0.02	**167.84**	0.94
*	*	*	60%	**59.86**	0.01	**93.98**	0.02
*	*	*	100%	**59.81**	0.00	**93.97**	0.02
1000	*	*	0%	**3677.69**	0.01	**6003.79**	1228
*	*	*	30%	**977.59**	0.02	**1559.17**	42.58
*	*	*	60%	**527.57**	0.03	**837.12**	0.05
*	*	*	100%	**527.57**	0.01	**837.21**	0.07

TABLE II
Influence of the computational load on the overall runtime. GPUs: GeForce GTX 1080 Ti and GeForce GTX Titan X. Time in milliseconds.

The first part of the evaluation focuses on a parallel tracking of all states in the scope of a single solver iteration since all required assignment and execution steps happen within one iteration. Increasing the number of iterations will (approximately) result in a linear scale of the measured runtime. However, in practice the approach scales better than linear in many cases since we can avoid different accelerator-stream synchronization operations between iterations. Within this single iteration, we use a straight-forward Runge-Kutta scheme to evaluate and recover states. We always perform 8 iterations per evaluation step (Runge-Kutta look-ahead step) including assignments and a single recovery step (Runge-Kutta forward step) including assignments. Table I shows the main evaluation table demonstrating the impact of the number of states, the assignment probabilities P and the fill rate F on the overall runtime. The GeForce GTX 1080 Ti is approximately between $1.5\times$ and $2\times$ faster on all benchmarks compared to the GeForce Titan X. As visualized in the benchmarks, increasing P does not change the overall runtime in any case. This is due do the fact that already a small number of 16384 states with a low probability of $P = 30\%$ will cause the iteration loop to break in nearly every step (the worst case). In all situations, a fill rate $F \geq \frac{2}{3}$ does not lead to any further performance improvements, since most of the states are already waiting for an assignment in these cases. Table II shows the impact of the load factor on the runtime. The measurements clearly show the scalability of our method on the GPUs when L is increased. In comparison to Table I, we can see the same influence of F on the overall runtime.

The second part of the evaluation targets memory consumption. For this purpose, we vary the number of states and the size in bytes of a single state. We assume that we are already in solver iteration 16 and have to backtrack to iteration 15, where $\forall i : |S_{back}(i)| = |S_{max}|$ (see subsection III-F) and a single state in history consumes 8 bytes (see subsection III-B). Table III shows details about the required total memory consumption (CPU + GPU) of our method in comparison to traditional approaches that keep all states in memory. However, as we treat memory consumption for runtime performance, we also evaluated the overall runtime overhead in Table IV. In this test, we performed 16 recovery iterations based on the previously described Runge-Kutta scheme using $L = 10$, as

before. We compared our runtime overhead against a traditional approach that either keeps all states in GPU memory or in CPU memory. In all cases, the required state information has to be copied to the current buffer for further processing. Comparing our method to methods having all states in GPU-memory is roughly $3\times$ slower. As not all states can be kept in memory (as shown in Table III), a CPU-based storage is often required. Comparing our recovery method to this concept yields speed-ups of approximately factor 6.

S	M_{hist}	M	Rec. Cache	**Our Method**	**All states**
16384	8	32KB	$\times$	**3.5GB**	**8.5GB**
*	8	*	-	**3GB**	*
*	8	64KB	$\times$	**5GB**	**17GB**
*	8	*	-	**4GB**	*
*	8	128KB	$\times$	**8GB**	**34GB**
*	8	*	-	**6GB**	*
65536	8	32KB	$\times$	**14GB**	**34GB**
*	8	*	-	**12GB**	*
*	8	64KB	$\times$	**20GB**	**68GB**
*	8	*	-	**16GB**	*
*	8	128KB	$\times$	**32GB**	**136GB**
*	8	*	-	**24GB**	*

TABLE III
TOTAL MEMORY CONSUMPTION WITH A RECOVERY PHASE FOR SOLVER ITERATION 16 WITH AND WITHOUT OUR FAST RECOVERY CACHE. M REFERS TO THE SIZE IN BYTES OF A SINGLE OPTIMIZATION STATE. M_{hist} IS 8 BYTES IN ALL CASES. TIME IN MILLISECONDS.

S	M	**Our Method**	σ	**All states**	GPU	σ
16384	32KB	**11.43**	0.51	**3.31**	$\times$	1.48
*	*	*	*	**70.99**	-	26.26
*	64KB	**20.59**	0.88	**6.59**	$\times$	1.97
*	*	*	*	**144.96**	-	39.83
*	128KB	**39.31**	3.19	**12.87**	$\times$	2.75
*	*	*	*	**285.07**	-	89.07

TABLE IV
RUNTIME OVERHEAD WITH A RECOVERY PHASE OF 15 ITERATIONS IN THE CURRENT SOLVER ITERATION 16. L WAS SET TO 10 AND F TO $\frac{2}{3}$. GPU: GEFORCE GTX 1080 TI. TIME IN MILLISECONDS.

VI. CONCLUSION

In this paper, we have presented a new high-level concept to track and reconstruct states in heuristic optimization systems on GPUs. Our method is a suitable extension to nearly every currently available optimization system, as it is straightforward to implement and provides a good balance between performance and memory consumption.

The presented evaluation demonstrates the scalability of our method with respect to different workloads, memory consumption and recovery performance overhead. We detected that a fill rate of larger $\frac{2}{3}$ is usually sufficient to achieve maximum performance based on our evaluation scenarios. On the one hand, the memory consumption can be significantly reduced in comparison to traditional methods that have to keep nearly all states in memory. On the other hand, our method trades memory consumption for computational power which leads to minor slow-downs in scenarios in which all states can be kept in GPU memory. However, as these cases are uncommon for large optimization problems, many states have to be copied back to CPU memory, which performs much slower than our recovery approach.

In the future, we would like to experiment with more advanced recovery-mechanisms that automatically decide which states to cache in which memory buffer. Moreover, we would like to add a multi-stage caching concept that manages and combines recovery information across different iterations. Applying our approach to multi-GPU scenarios and their synchronization with respect to state information could also be a very promising next step.

ACKNOWLEDGMENT

The authors would like to thank Gian-Luca Kiefer and Thomas Schmeyer for their suggestions and feedback regarding our paper.

REFERENCES

[1] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. I. Cowling, S. Tavener, D. Perez, S. Samothrakis, S. Colton, and et al., "A survey of Monte Carlo tree search methods," *IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI*, 2012.

[2] S. Gelly, L. Kocsis, M. Schoenauer, M. Sebag, D. Silver, C. Szepesvári, and O. Teytaud, "The Grand Challenge of Computer Go: Monte Carlo Tree Search and Extensions," *Communications of the ACM*, 2012.

[3] F. Campeotto, A. Dal Palù, A. Dovier, F. Fioretto, and E. Pontelli, "Exploring the Use of GPUs in Constraint Solving," in *Practical Aspects of Declarative Languages*, 2014.

[4] F. Campeotto, A. Dovier, F. Fioretto, and E. Pontelli, "A GPU Implementation of Large Neighborhood Search for Solving Constraint Optimization Problems," in *Proceedings of the Twenty-first European Conference on Artificial Intelligence*, 2014.

[5] O. Abdelkafi, K. Chebil, and M. Khemakhem, "Parallel local search on GPU and CPU with OpenCL Language," in *Proceedings of the first international conference on Reasoning and Optimization in Information Systems*, 09 2013.

[6] Y. Ming Lam, K. Hung Tsoi, and W. Luk, "Parallel neighbourhood search on many-core platforms," *International Journal of Computational Science and Engineering*, vol. 8, 2013.

[7] T. V. Luong, N. Melab, and E.-G. Talbi, "Neighborhood Structures for GPU-based Local Search Algorithms," *Parallel Processing Letters*, 2010.

[8] T. V. Luong, L. Loukil, N. Melab, and E. Talbi, "A GPU-based iterated tabu search for solving the quadratic 3-dimensional assignment problem," in *ACS/IEEE International Conference on Computer Systems and Applications (AICCSA)*, 2010.

[9] N. Melab, T. V. Luong, K. Boufaras, and E.-G. Talbi, "ParadisEO-MO-GPU: a framework for parallel GPU-based local search metaheuristics," in *11th International Work-Conference on Artificial Neural Networks*, 2011.

[10] C. Novoa, A. Qasem, and A. Chaparala, "A SIMD Tabu Search Implementation for Solving the Quadratic Assignment Problem with GPU Acceleration," in *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, 2015.

[11] G. M. J. B. Chaslot, M. H. M. Winands, and H. J. van den Herik, "Parallel Monte-Carlo Tree Search," in *Computers and Games*, 2008.

[12] K. Rocki and R. Suda, "Massively Parallel Monte Carlo Tree Search," *Proceedings of the 9th International Meeting High Performance Computing for Computational Science*, 2010.

[13] C. Xiao, J. Mei, and M. Müller, "Memory-Augmented Monte Carlo Tree Search," 2018.

[14] Y. Zhou and J. Zeng, "Massively Parallel A* Search on a GPU," 2015.

[15] E. Powley, P. Cowling, and D. Whitehouse, "Memory Bounded Monte Carlo Tree Search," 2017.

[16] NVIDIA, *CUDA C Programming Guide v10*, 2019.

[17] L. T. Mohammad Harun Rashid, "Parallel Combinatorial Optimization Heuristics with GPUs," *Advances in Science, Technology and Engineering Systems Journal*, 2018.

[18] D. Lustig, S. Sahasrabuddhe, and O. Giroux, "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*, 2019.

[19] R. Hunger, "Floating Point Operations in Matrix-Vector Calculus," Technische Universität München, Tech. Rep., 2007.