Exploiting the Computational Power of the Graphics Card: Optimal State Space Planning on the GPU

Damian Sulewski and **Stefan Edelkamp** and **Peter Kissmann**
TZI Universität Bremen, Germany
{sulewski, edelkamp, kissmann}@tzi.de

Abstract

In this paper optimal state space planning is parallelized by exploiting the processing power of a graphics card. The two exploration steps, namely selecting the actions to be applied and generating the successors, are performed on a graphics processing unit. Duplicate detection, however, is delayed to be executed on the central processing unit. Multiple cores are employed to bypass main memory latency. To increase processing speed for exact duplicate detection, the hash tables are lock-free. Moreover, a bucket-based representation enhances the concurrent distribution of frontier states.

The planner supports cost-first exploration and is able to deal with a considerable fraction of current PDDL, including numerical state variables, complex objective functions, and goal preferences. It can maximize the net-benefit. Experimental findings show visible performance gains especially for larger benchmark problems.

Introduction

There is no doubt that the success of planners is sensitive to the amount of computational resources available. It is not hard to predict that – due to economic pressure – parallel computing on an increased number of cores both in central processing units (CPUs) and in graphics processing units (GPUs) will be essential to solve challenging problems in the future. The currently fastest computer (Tianhe-Ia) has more than $14{,}000$ (Intel Xeon X5670 6-core) CPUs and $7{,}000$ (Nvidia Tesla M2050) GPUs, while a combination of powerful multicore CPUs and many-core GPUs is becoming standard technology for the consumer market.

Planners trying to catch up with these hardware trends enhance state space planning using, e. g., different multicore CPU approaches (Kishimoto, Fukunaga, and Botea 2009; Vidal, Bordeaux, and Hamadi 2010; Burns et al. 2009b). Often, suboptimal planning is addressed (Kishimoto, Fukunaga, and Botea 2010; Burns et al. 2009a; Nakhost, Hoffmann, and Müller 2010), but also a sizable number of optimal parallel planners has been developed in the last few years (e. g., Zhou and Hansen 2007, and Zhou et al. 2010).

GPU-support has been shown as effective for enhancing single-agent search problems (Edelkamp, Sulewski, and Yücel 2010). Unfortunately, so far no domain-independent planner has been proposed that utilizes the GPU. This is partly due to the fact that the single instruction multiple data architecture of GPUs is more closely related to a vector computer that induces a distinguished programming model.

In this paper, we propose a domain-independent planner for which precondition checks and successor generation are executed on the GPU. As GPUs usually have no cache hierarchy and are relatively slow in accessing the global memory on the graphics card, duplicate detection is executed in the RAM using the CPU. Although we can restrict the planner to one core for duplicate detection, we found that running multiple CPU cores is beneficial to avoid memory latency. For large state spaces the planner supports exploration on disk, together with either delayed duplicate detection (Korf 2008) or bit-state hash tables (Bloom 1970). As our main interest is optimal planning, for this work we implemented a so-called *lock-free* hash table, a promising data structure based on low-level compare-and-swap operations that avoids using variables for locking (Laarman, van de Pol, and Weber 2010; Enzenberger and Müller 2009).

Imposed by the demands of the GPU we design a cost-first planner that differs substantially from existing ones. As a surplus, our planner can deal with a considerable fraction of current PDDL: propositional and numerical variables, ADL constructs like negative and disjunctive preconditions, action costs (including zero-cost actions), linear and non-linear objective functions, and preferences, which results in optimizing the (net-)benefit.

The paper is structured as follows. After brief introductions to the supported planning problems and to GPU programming we provide insights to the architecture of our multi- and manycore planner. We show how the checks for satisfied preconditions and the generation of successor states are executed on the GPU, and how duplicates are detected in lock-free hash tables on the CPU for fast memory access. We briefly reflect implementation refinements to standard planning algorithms. In the experiments we study improvements obtained wrt. other competitive planning systems.

Planning Problems

A *classical planning problem* is a tuple $\mathcal{P} = (\mathcal{F}, \mathcal{A}, \mathcal{I}, \mathcal{G})$ with $\mathcal{F}$ being a set of fluents, $\mathcal{A}$ a set of actions, $\mathcal{I} \subseteq \mathcal{F}$ the fluents that hold in the initial state and $\mathcal{G} \subseteq \mathcal{F}$ the fluents

that need to be satisfied in any goal state. An action $a \in \mathcal{A}$ is a tuple $a = (P, A, D)$, with $P \subseteq \mathcal{F}$ being the precondition that needs to be satisfied so that action a can be applied, $A \subseteq \mathcal{F}$ the set of fluents that are added to the current state and $D \subseteq \mathcal{F}$ the set of fluents that are removed from it after applying the action.

The aim is to find a plan, i. e., a sequence of actions, that transforms the initial state into a goal state. In case of optimal planning, this plan must be minimal.

In *cost-based planning*, actions can be assigned certain costs, so that the planning problem is extended to the tuple $\mathcal{P}_c = (\mathcal{F}, \mathcal{A}, cost, \mathcal{I}, \mathcal{G})$ with $cost : \mathcal{A} \mapsto \mathbb{N}_0^+$. For such a problem, the total cost of the resulting plan is the sum of the costs of all actions within the plan and in case of optimal planning, a plan with minimal total cost has to be found.

Oversubscription planning is the extension of classical planning to so-called soft goals, i. e., goals that may be satisfied but are not obligatory. Thus, the problem is a tuple $\mathcal{P}_o = (\mathcal{F}, \mathcal{A}, \mathcal{I}, \mathcal{G}, utility)$ with $utility : 2^{\mathcal{F}} \mapsto \mathbb{N}$ a function that assigns a certain reward for achieving a soft goal. The utility of the resulting plan is the utility of the achieved goal state and the aim is to maximize it.

Finally, *net-benefit planning* contains action costs and soft goals, i. e., the net-benefit planning problem is a tuple $\mathcal{P}_{nb} = (\mathcal{F}, \mathcal{A}, cost, \mathcal{I}, \mathcal{G}, utility)$. The net-benefit of a plan is the utility achieved by the plan minus the total action cost needed to achieve it and in case of optimal planning we are interested in finding a plan that maximizes this net-benefit.

All these definitions are based on STRIPS planning (Fikes and Nilsson 1971). Nowadays, the planning domain definition language PDDL (McDermott 1998) is the most frequently used formalism for modeling and it additionally supports, among others, complex Boolean formulas for the preconditions of actions and the goal descriptions, numerical state variables, and rational action costs (which we scale to integers). Matching the formalization above, our planner assumes a grounded problem representation with a fully instantiated PDDL description as input (e. g., provided by Haslum's *pddlcat*, Helmert's *translate*, or Hoffmann's *adl2strips*).

GPU Programming

Graphics processing units (GPUs) are general purpose multithreaded data parallel co-processors having hundreds of cores compared to the small number of cores a typical CPU contains. GPU power can been leveraged for many computationally intense operations, not only for graphics (Hwu 2011). However, it is a challenge to effectively use these massively parallel processors to achieve efficiency and performance goals. This imposes restrictions on the programs that should be executed on GPUs and leaves options for exploiting multicore parallelization on the CPU.

GPUs are programmed through *kernels* which are executed as sets of *threads*. Each thread of the kernel executes the same code. Threads of a kernel are grouped in *blocks*. Each block is uniquely identified by its index and each thread is uniquely identified by the index within its block. The dimensions of the thread and the thread block are specified at the time of launching the kernel.

Programming GPUs is facilitated by application programming interfaces (APIs). Programs are often C-like with extensions such as special declarations to explicitly place variables in some of the memories (e. g., shared, global, local), predefined keywords (variables) containing the block and thread IDs, synchronization statements for cooperation between threads, runtime API for memory management (allocation, deallocation), and statements to launch functions on the GPU. This minimizes the software's dependence of the given hardware.

The memory model loosely maps to the program thread-block-kernel hierarchy. Each thread has its own *on-chip registers* which are fast and *off-chip local memory*, which is quite slow. Per block there is also an on-chip *shared memory*. Threads within a block cooperate via this memory. For blocks executed in parallel the shared memory is equally split between them. All blocks and threads within them have access to the off-chip *global memory* at the speed of RAM. Usually, global memory is mainly used for communication between the host and the kernel. Threads within a block can communicate also via light-weight synchronization barriers.

The GPU architecture consists of a set of multiprocessor units. Communication with the off-chip device memory is relatively slow compared to the enormous peak computational power. This is usually the main performance bottleneck. To fully exploit the capacity of the GPU parallelism this memory latency must be minimized. Another issue that can lead to a performance degradation is unnecessary synchronization between thread blocks. The inter-thread communication within a block is cheap via the fast shared memory, but the accesses to the global and local memories are more than a hundred times slower.

Unlike CPU threads, GPU threads are light-weight with negligible overhead of creation and switching. This allows GPUs to use thousands of threads, whereas multicore CPUs use only a few. Usually more threads and blocks are created than the number of streaming processors, which allows the GPU to maximally use the capacity via smart scheduling – while some threads/blocks are waiting for data, the others that have their data ready are assigned for execution. Thus, another way to maximize the parallelism is by optimizing the thread mapping. This is often tightly coupled with the optimization of the memory access. One should strive towards an alignment of the data in the memory such that threads of the same block access memory locations which are as close as possible. In this case we have so-called coalesced accesses. Thus, threads that access physically close memory locations should be grouped together such that they can be provided data with the same memory access. Finally, in order to minimize the access to the slow global memory, one should exploit data reuse. The parts of the computation are localized to thread blocks which are synchronized as loosely as possible. These threads use local data as much as possible and the global results are written only at the end of the computation.

Planner Architecture

Our planner implements two different kernels, one designed to generate successors for cost-optimal planning and one

that deals with optimal planning for oversubscribed and net-benefit planning problems. Both kernels are able to deal with numbers. The main difference of the two kernels is that the former stops at expanding the first goal node, while the latter overwrites the action after being executed with the metric value of the state.

Successor Generation on the GPU

For successor generation on the GPU we check the satisfaction of the preconditions of actions against the state set that has been copied to the GPU and apply the effects to the ones that have passed the test. As this is a considerable amount of work for each GPU core, we exploit the postfix representation of the propositional and numerical expressions that appear in the precondition and effects. These representations are precomputed and broadcasted in the GPU. Extensive tests showed that using the postfix representation enhances checking the validity of a precondition and the computation of the assignment to an effect variable on the GPU due to using a flat evaluation stack.

Conceptually (and as illustrated in the simplified pseudocode in Algorithm 2), for each track we have one kernel for parallel state expansion on the GPU. In our practical implementation, however, successors are generated in two steps on the GPU. In a first step, the preconditions of the actions are checked against the states in the GPU. We allow rather complex Boolean and numerical expressions and check the preconditions of all actions against the state (without applying any static filter).

Splitting the kernels into two implies that we copy data from the CPU to the GPU and back twice for each expansion set. The reason is that we completely avoid dynamic memory allocation on the GPU, so that in the first kernel we determine the number of possible successors, and for the second kernel we provide sufficient space to actually compute the successors. This way we also save GPU memory and can exploit a higher degree of parallelism.

We also tried the one step approach by Edelkamp, Sulewski, and Yücel (2010), copying the states only once to the GPU. However, here the GPU has to check all actions for applicability twice, once for space allocation, once for generation of the successors. While performing this step in board games is faster then copying the data, in planning, where checking the applicability is more complex, we found it to be the slower solution.

Duplicate Detection on the CPU

After porting the successor generation to the GPU, duplicate detection turned out to be the most time consuming step in the whole planning process. We analyzed several techniques to port this step to the GPU and decided not to do so. A pure GPU-based approach would assume to store the closed and open list in the memory of the graphics card which is inefficient due to the available size. The alternative is to store already seen states externally and utilize the GPU to speed up the checking process. This strategy was already evaluated by us in model checking (Edelkamp and Sulewski 2010) and has shown to be superior compared to other external approaches. Since in this work we compare to internal planners which utilize only RAM this method is too slow because of moving the states to the hard disk and back.

To support large-scale problem solving, duplicate detection can be delayed to be executed on the disk, or by one- (or multi-) bit-state hash tables that can produce false positives but with very low probability. When experimenting with these filters we found that due to the efficiencies of the GPU, memory latency on the CPU (namely addressing the bits in RAM for reading and writing) was the bottleneck in the implementation, which urged us to consider using multiple CPU cores for duplicate detection.

As our main interest in this paper is optimal planning, we cannot use approximate hashing. Therefore, we have implemented lock-free hashing, a new trend for using hash tables in a multicore scenario.

The goal was to realize an efficient shared state storage for duplicate detection. Traditional hash tables associate a piece of data to a unique key in the table. Here, we only need to store and retrieve state vectors. Thus, the key is the state vector itself. The time efficiency of the lookup should scale with the number of cores executing it in parallel. Pointers and memory allocations are avoided. The implementation of the hash table uses open addressing tuned to use the cache more efficiently by small-sized linear probing. Lock-free algorithms guarantee system-wide progress in modern CPUs. They implement a compare-and-swap operation (CAS), so that always some thread can continue its work.

In our solution, however, we avoid explicit locks by CAS, but provide only statistical progress guarantees, which leads to a simpler implementation at no penalty in performance. Strictly speaking, our algorithm locks *in-situ* – it needs no additional variables for implementing the locking mechanism. CAS ensures atomic memory modification while at the same time preserving data consistency. This can be done by reading the value from memory, performing the desired computation on it and writing the result back.

The problem with lock-free hashing is that it relies on low-level CAS operations with an upper limit on the data size that can be stored (one memory cell). In order to store planning states that usually exceed the size of a memory cell, we need two arrays: the *index*, where the locking mechanism is realized by CAS and the *data* array. An index stores memorized hashes and the write status bit of the data in the data array. If h is the memorized hash, the possible values of the index are thus: $(-, \top)$ for being empty, $(h, \bot)$ for being blocked for writing and $(h, \top)$ for a released lock.

Since comparing large data vectors in main memory is a costly operation, the index table is used to avoid this if unnecessary. When checking if s is present in the closed list its hash value $h(s)$ along with the position in the hash table $p = h(s) \mod tablesize$ is computed and a block of the index table $index_{i,\ldots,j}$ with $i \neq j$ and $i \leq p \leq j$ is transferred into the cache of the CPU. Here we compare $index_p$ with $h(s)$ and increment p by one if they differ until $index_p = (-, \top)$ or $p = j$. In the first case, $h(s)$ is stored in the index table and the vector is stored in the data table. In the second case we set $p = i$. This strategy assures that a complete vector comparison is performed only for states s and s' if $h(s) = h(s')$.

GPU Planning Algorithm

In domains with uniform action costs a breadth-first enumeration of the search space is sufficient, while in domains with action costs optimal path finding resorts to some form of cost-first shortest path exploration. If the action costs are integers, a cost-based implementation of Dijkstra's shortest path algorithm on buckets is possible (Dial 1969).

For smaller problems we encountered that a strict cost-wise exploration of the search space can be unfortunate, since the overhead for copying the data from and to the GPU may dominate the run time.

Subsequently, with a *buffer-filling* implementation variant, we feature *eager* state expansion on the GPU, similar to Edelkamp and Schrödl (2000) by means that states may be expanded earlier than dictated by the monotonic non-decreasing ordering of the costs. For this case the first expanded goal state no longer necessarily has optimal solution cost. However, given that costs are non-negative, states can be omitted if the current cost exceeds the best one found so far. The performance gains due to improved parallel processing can exceed the additional amount for expanding more states than necessary.

Moreover, we relax duplicate elimination. While the algorithm prevents inserting states with larger costs than the one stored in the hash table, inserting states with smaller costs than the ones stored in the hash table does not imply that the latter ones are removed. This can result in significant re-expansions.

The pseudo-code of the eager buffer-filling planning algorithm is shown in Algorithm 1 with an expansion kernel routine for the GPU sketched in Algorithm 2.[1] *Open* is implemented as a hash map of lists (with the costs being the keys). This sparse representation of a virtual bucket array of lists, where all states in a bucket have the same action costs, allows to deal with larger action costs present in some of the domains. *Closed* is implemented as a (lock-free) hash table. All other structures are implemented as simple vectors or lists.

For the sake of clarity, the code abstracts from the implementation. We write of Locking and Unlocking the buckets of Open, but apply internal duplicate detection with lock-free hashing based on the buffer and data arrays. Moreover, as indicated above, for a better exploitation of the memory on GPU, in the implementation we have two GPU kernel routines for expansion, one that checks preconditions and outputs a vector of applicable actions, and a second one that applies the effects for the actions found to be applicable in the first step.

Considering net-benefit planning, the net-benefit is computed as the utility for each soft goal established minus the label of the cost layer (line 14). The continuously improving bound *best* records the best net-benefit found so far and terminates the exploration in case the optimum has been proven. As a result (and as a side effect of the eager buffer-filling approach), net-benefit problems can also be solved optimally with Algorithm 1.

[1]Throughout the algorithms we use square brackets to denote concepts only relevant in net-benefit planning.

Algorithm 1 Optimal Eager Buffer-Filling GPU Planning Algorithm.

Input: $\mathcal{P}_{c[nb]} = (\mathcal{F}, \mathcal{A}, cost, \mathcal{I}, \mathcal{G}[, utility])$
Output: Optimal [value of utility minus] action cost

```
1.  best := ∞; i := 0
2.  Open_i := {I}; Closed := {(I, 0)}
3.  loop
4.    if (∀j ≥ i : Open_j = ∅)
5.      if (best = ∞)
6.        return unsolvable
7.      else
8.        return best
9.    Buffer := ∅
10.   while (∃j ≥ i : Open_j ≠ ∅) and (|Buffer| < Max)
11.     i' := min{j ≥ i | Open_j ≠ ∅}
12.     for (v ∈ Open_i') and (|Buffer| < Max)
13.       Insert (v, i') in Buffer
14.       if (G ⊆ v) best := min{best, [utility(v)−]i'}
15.       Remove v from Open_i'
16.   Copy Buffer to GPU
17.   Succs := GPU-Expand(P_c[nb], Buffer)
18.   Copy Succs to CPU
19.   for (v, c) ∈ Succs
20.     c' := Search(v) in Closed
21.     if (c < c')
22.       Insert (v, c) in Closed
23.     if (c < best)
24.       Lock Open_c
25.       Insert v in Open_c
26.       Unlock Open_c
27.   i := min{j | Open_j ≠ ∅}
```

Theorem 1. *(Optimality) For cost-based optimal planning problems $\mathcal{P}_c$ and net-benefit planning problems $\mathcal{P}_{nb}$ the eager buffer-filling planning algorithm on the GPU computes an optimal solution.*

Proof. States are only eliminated from the *Open* list in case a strictly better (in terms of accumulated action costs) matching state has been found in the *Closed* list or if the non-decreasing accumulated metric value (measured either in accumulated action cost or net-benefit) is worse than the currently best established goal metric value (measured either in accumulated action cost or net-benefit). □

Theorem 2. *(Efficiency) For cost-based optimal planning problems $\mathcal{P}_c$ and net-benefit planning problems $\mathcal{P}_{nb}$ the eager buffer-filling planning algorithm expands each state at most once for every action cost-layer of the search frontier.*

Proof. As in each cost-layer we apply full duplicate detection, any state can appear at most once per layer. □

As the number of re-expansions in other cost-layers depends on the domain, for the experiments we kept a version

Algorithm 2 Kernel *GPU-Expand* executed on the GPU.

Input: $\mathcal{P}_{c[nb]} = (\mathcal{F}, \mathcal{A}, cost, \mathcal{I}, \mathcal{G}[, utility])$, *Buffer*
Output: *Succs*

1. *Succs* := $\emptyset$
2. **for each** *GPU-thread*
3. select state-cost pair (u, c) in *Buffer*
4. **for** $a \in \mathcal{A}$
5. **if** u satisfies precondition of a
6. v := compute result of applying a in u
7. Insert $(v, c + cost(a))$ in *Succs*
8. **return** *Succs*

that avoids buffer-filling and performs a Dijkstra-like state-space traversal. The changes for the pseudo-code are moderate. For cost-based planning we stop at the first goal to be expanded. In both cases the loop starting at line 10 is pruned to one iteration with i' being substituted by i.

Experiments

The benchmark domains are those of the sequential optimal and optimal net-benefit tracks of IPC 2008.[2] The competitors are the two best planners in each track. For sequential optimal planning these are the organizers' baseline planner, performing explicit-state Dijkstra search implemented as A* with a zero-heuristic and based on the efficient search code of Lama (Richter, Helmert, and Westphal 2008), as well as Gamer (Edelkamp and Kissmann 2009), a BDD-based bidirectional cost-first planner. The optimal net-benefit planners are Gamer (featuring BDDs and unidirectional cost-first branch-and-bound planning) and MIPS-XXL (Edelkamp and Jabbar 2008), an explicit-state breadth-first external-memory planner.

The computer infrastructure is an Intel $i7$ (4-core) 920 CPU with 2.67 GHz and 24 GB RAM as well as a hard disk drive with a capacity of 1 TB. The graphics card used is an NVIDIA 480 GTX with a 700 MHz GPU and 1.5 GB onboard memory, programmed in CUDA. All experiments are canceled after exhausting memory or 15 minutes of wall-clock time.[3] For lock-free hashing, our implementation uses the GNU gcc compiler for 64-bit x86 target platforms. A gcc built-in is used for the CAS operation and reads and writes from and to indexes are marked volatile.

In Figures 1 and 2 we have plotted the running time in seconds for each instance of a domain if it was solved by a planner. To score the planners we use the same system as in the last IPC, i. e., for each domain the number of solved instances is accumulated. If a domain is available in different formulations we use the maximum number of solved instances over all these formulations.

In Figure 1 we have displayed the results obtained in the six benchmarks of the optimal net-benefit track. Due to a bug in our planner's parserwe were not able to handle the Peg-Solitaire domain, so that we decided to omit it.

For the other domains the GPU Planner with buffer-filling enabled can solve 43 problems, the GPU Planner without buffer-filling 52, MIPS-XXL 22 and Gamer the highest number of 57 problems. However, investigating the cause for unsolved instances we can report that while Gamer was killed due to reaching the limit of 15 minutes, the GPU Planner without buffer-filling never reached this timeout. In many of the instances the lock-free hash table was completely filled and the planner stopped. Thus, a larger amount of main memory, or an effective compression strategy for the states would turn the picture in favor of the GPU Planner.

The plots also depict another fact about the GPU Planner, i. e., it depends on hard problems. In all plots but the Crewplanning domain we see that there is a threshold where the GPU Planner is significantly faster than the other planners. Furthermore, in Crewplanning we see advantages for buffer-filling while in the others this strategy is less effective. Currently we are uncertain of the reason for this, the only facts we could make out is that the state space of this domain is rather *flat*, i. e., each cost layer is small. A flat state space means a better distribution of states into separate layers and thus fewer re-expansions.

We cannot compare numbers of expanded nodes due to the basic differences of the planners. MIPS-XXL, using external memory, and Gamer, executing a symbolic search, are not comparable in this respect with the explicit state internal memory GPU-based planner.

In Figure 2 we have displayed the results obtained in the eight STRIPS benchmarks of the sequential optimal track.

The GPU planner with buffer-filling solves 124 problems, the GPU planner without buffer-filling 138 problems, the baseline planner 134 problems and Gamer 127 problems. The advantage used by the GPU planner here is the large amount of Boolean fluents in the problem description. Due to the usage of only one bit per fluent, compared to 32 bits for integer fluents, the vector identifying the state is smaller and more vectors fit into the hash table, enabling the planner to examine larger state spaces and better utilizing the computation power of the GPU.

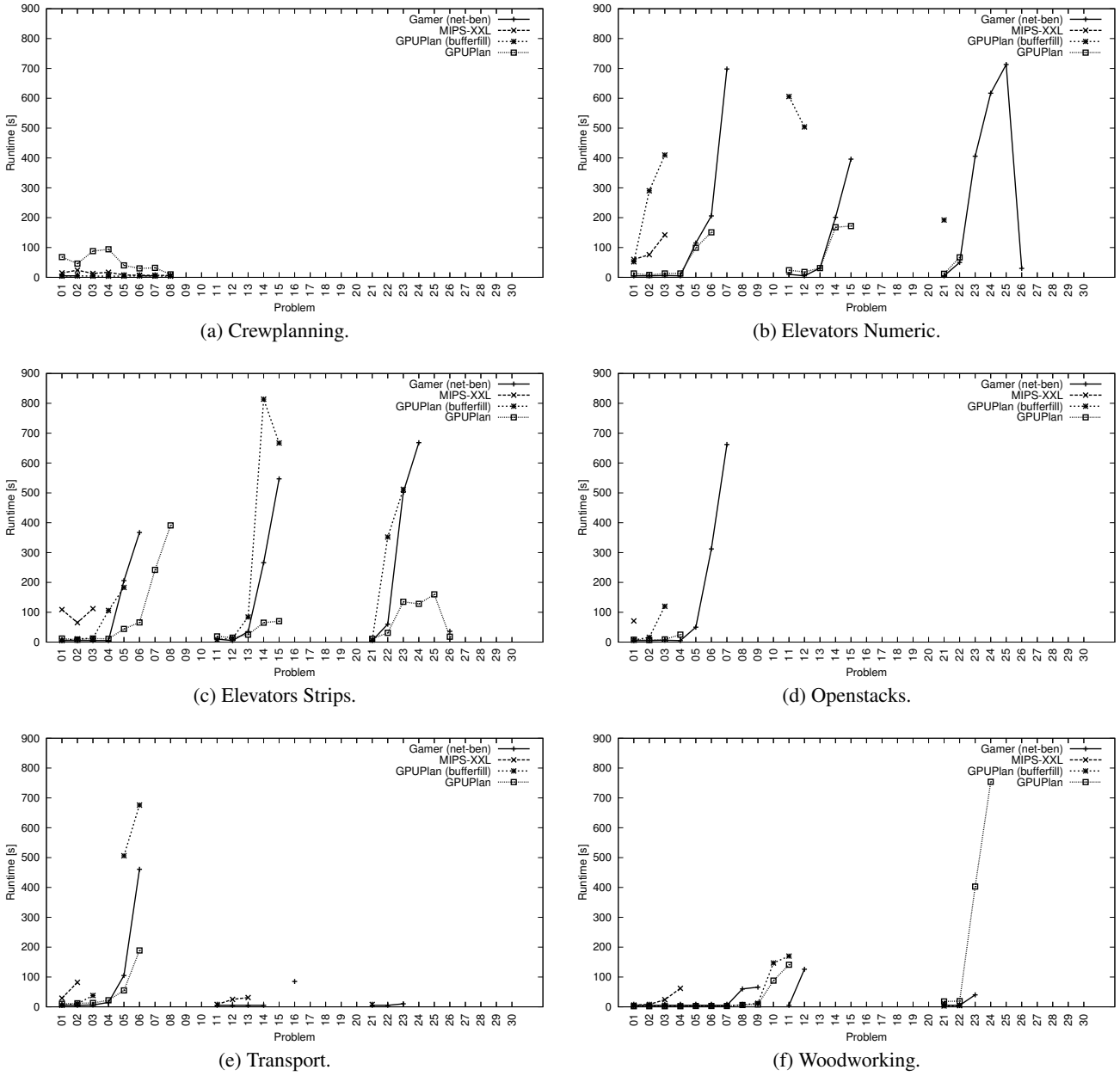
The plots emphasize the advantages of using the GPU. In every domain a threshold exists where the GPU planner is faster then the CPU based baseline planner. Unfortunately, even 24 GB of RAM are not enough to solve the hardest problems and demonstrate the achievable speed. In the best cases the speedup factor, defined as the running time of baseline divided by the running time of the GPU planner, exceeds a factor of seven, rising with the size of the problem.

The GPU planner with buffer-filling behaves satisfactorily only in the Parcprinter domain, being slower than the other planners on all the remaining ones. Here we also experience a flat state space, in this case due to the strong divergence of the costs, keeping the number of re-expansions small.

We also performed additional experiments for the domains of the sequential optimal track with a timeout of 30 minutes. In these, the GPU planner can solve four additional instances, the baseline planner only one and Gamer eight, so that the GPU planner still beats both competitors.

[2] `http://ipc.informatik.uni-freiburg.de`

[3] For the version of Gamer used in the sequential optimal track we set the time for the backward search to 450 seconds.



(a) Crewplanning.

(b) Elevators Numeric.

(c) Elevators Strips.

(d) Openstacks.

(e) Transport.

(f) Woodworking.

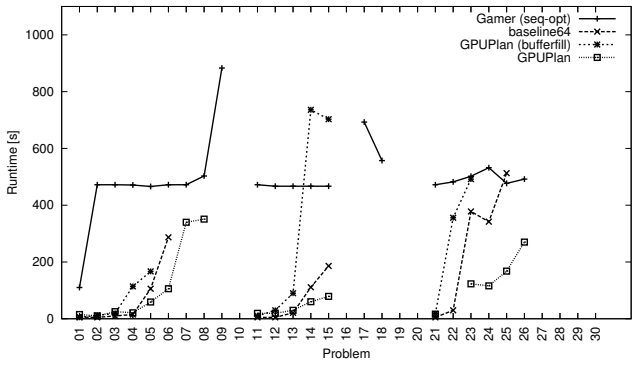
Figure 1: Performance Results Optimal Net-Benefit Track.

Conclusion and Discussion

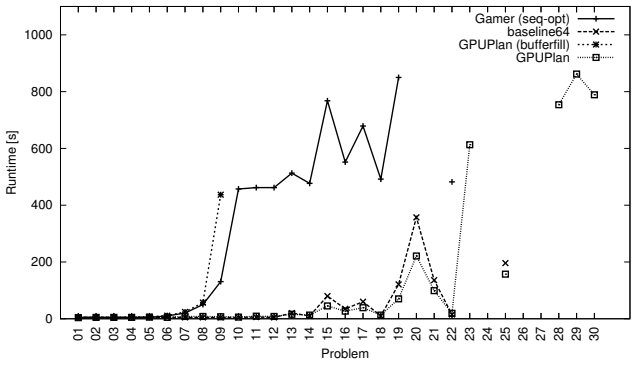
In this paper we have proposed the first domain-independent planner that exploits processing power available on the graphics card. To enhance precondition checks as well as assignments to effect variables on the GPU, we use a postfix notation of the expressions. For duplicate detection, we employ lock-free hash tables that certify optimal solutions in algorithms like cost-first search. Variable locks are needed only once for each cost-layer. We exploit the distribution of successors to enhance parallelism for pushing states into the search frontier.

The expressiveness of the planner is considerable as it can handle a substantial fraction of grounded PDDL. The support of a multi-valued variable encoding may be utilized based on SAS$^+$ information inferred during the instantiation process (we tested that state vectors can be compressed and uncompressed on the GPU). To support multi-valued variables we prefer to re-write the input, so that preconditions and effects become numerical. The planner also supports indirect variable addressing, a feature recently introduced in PDDL but not present in most of the current benchmarks.

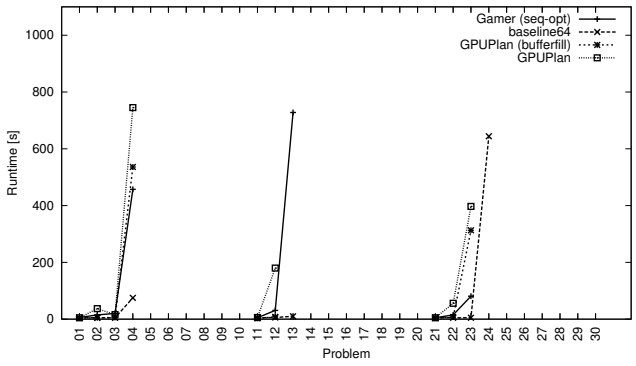
The approach is distinct to existing multicore approaches as multiple cores on the CPU are only used for delayed duplicate detection. In contrast, other parallelizations dis-



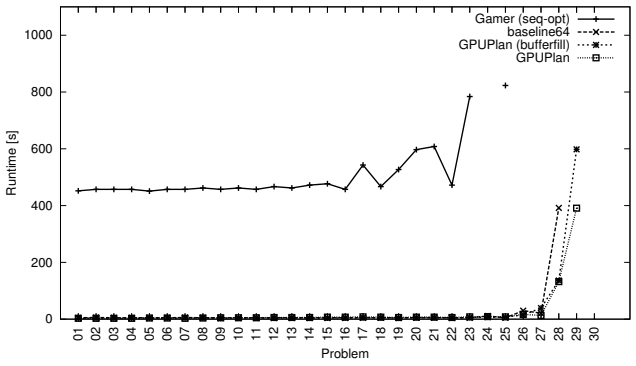
(a) Elevators.



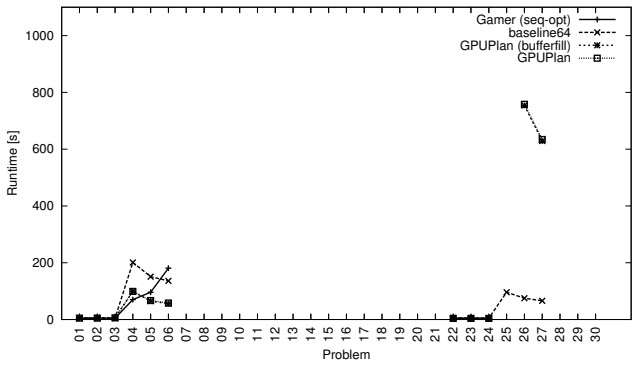
(b) Openstacks.



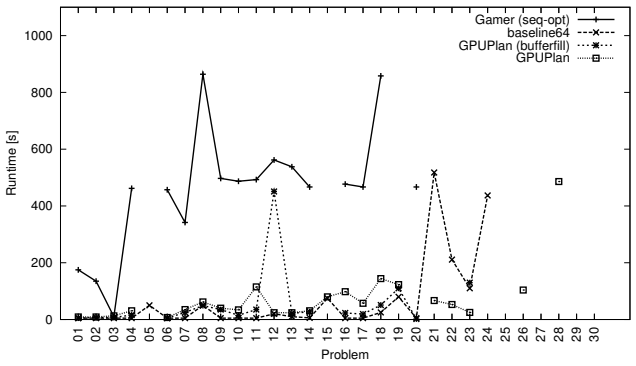
(c) Parcprinter.



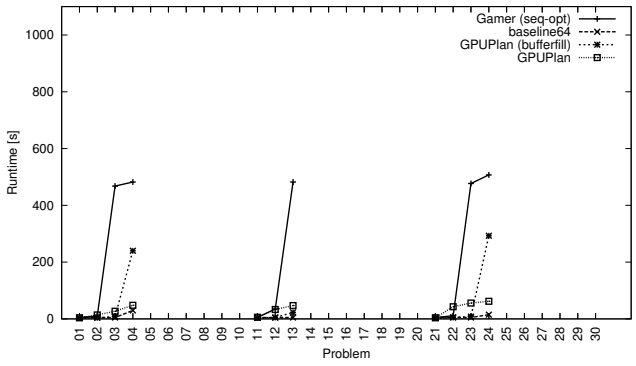
(d) Peg Solitaire.



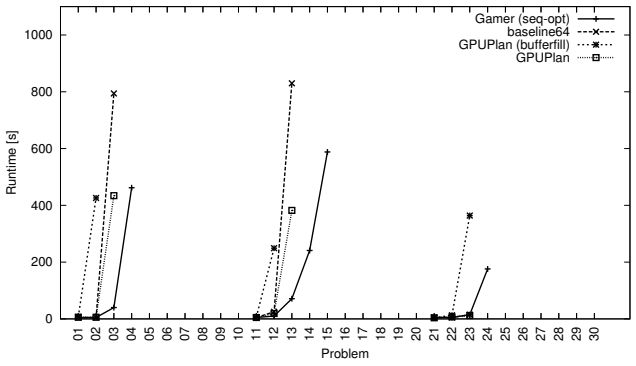
(e) Scanalyzer.



(f) Sokoban.



(g) Transport.



(h) Woodworking.

Figure 2: Performance Results Sequential Optimal Track.

tribute the search space based on different sorts of hash-partitioning, e. g., HDA* (Kishimoto, Fukunaga, and Botea 2009) combines ideas from a parallel version of A* (Evett et al. 1995) and transposition-driven scheduling (Romein et al. 2009), while PBNF (Burns et al. 2009b) extends the idea of parallel structured duplicate detection (Zhou and Hansen 2007), exploiting the locality of a search space. Arvand (Nakhost, Hoffmann, and Müller 2010) is a planner based on Monte-Carlo searches and random restarts. As different starts are independent, assuming sufficient memory, these searches can be parallelized easily. The results are promising but solutions are typically sub-optimal. (Multi-core) UCT, a dynamical tree-growing learning algorithm on top of Monte-Carlo search, can be used for finding optimal solutions in the limit, but has not yet been implemented.

For STRIPS-like actions a known filter neglects actions that do not have any propositions in the preconditions matching the proposition of the current state. Such specialized kernel precomputing lists of actions linked to the precondition they contain has not yet been implemented.

While we are mainly interested in optimal plans and blind search, it is not difficult to add heuristics to enhance the planning process. For consistent heuristics A* is a variant of Dijkstra's algorithm. Moreover, adapting the search to algorithms like BFHS (Zhou and Hansen 2004) itself does not pose a large implementation overhead, and duplicate detection by lock-free hashing virtually remains the same. The crucial point for efficient heuristic search is the parallel computation time for the heuristic. If calculated on the GPU, the restriction is that the heuristic does not consume much space to exploit shared memory best.

Acknowledgments

Thanks to DFG for support in projects ED74/8-1 and ED74/11-1. We also wish to thank Alfons Laarman and Michael Weber for the access to their lock-free hash table implementation.

References

Bloom, B. 1970. Space/time trade-offs in hashing coding with allowable errors. *Communication of the ACM* 13(7):422–426.

Burns, E.; Lemons, S.; Ruml, W.; and Zhou, R. 2009a. Suboptimal and anytime heuristic search on multi-core machines. In *ICAPS*, 42–49.

Burns, E.; Lemons, S.; Zhou, R.; and Ruml, W. 2009b. Best-first heuristic search for multi-core machines. In *IJCAI*, 449–455.

Dial, R. B. 1969. Shortest-path forest with topological ordering. *Communications of the ACM* 12(11):632–633.

Edelkamp, S., and Jabbar, S. 2008. MIPS-XXL: Featuring external shortest path search for sequential optimal plans and external branch-and-bound for optimal net benefit. In *IPC*.

Edelkamp, S., and Kissmann, P. 2009. Optimal symbolic planning with action costs and preferences. In *IJCAI*, 1690–1695.

Edelkamp, S., and Schrödl, S. 2000. Localizing A*. In *AAAI*, 885–890.

Edelkamp, S., and Sulewski, D. 2010. External memory breadth-first search with delayed duplicate detection on the GPU. In *MoChArt*.

Edelkamp, S.; Sulewski, D.; and Yücel, C. 2010. Perfect hashing for state space exploration on the GPU. In *ICAPS*, 57–64.

Enzenberger, M., and Müller, M. 2009. A lock-free multithreaded Monte-Carlo tree search algorithm. In *Advances in Computer Games (ACG)*, 14–20.

Evett, M. P.; Hendler, J. A.; Mahanti, A.; and Nau, D. S. 1995. Pra*: Massively parallel heuristic search. *Journal of Parallel and Distributed Computing* 25(2):133–143.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2(3–4):189–208.

Hwu, W.-M. W. 2011. *GPU Computing Gems*. Morgan Kaufmann.

Kishimoto, A.; Fukunaga, A. S.; and Botea, A. 2009. Scalable, parallel best-first search for optimal sequential planning. In *ICAPS*, 201–208.

Kishimoto, A.; Fukunaga, A.; and Botea, A. 2010. On the scaling behavior of HDA*. In *SoCS*, 61–62.

Korf, R. E. 2008. Linear-time disk-based implicit graph search. *Journal of the ACM* 55:26:1–26:40.

Laarman, A.; van de Pol, J.; and Weber, M. 2010. Boosting multi-core reachability performance with shared hash tables. In *Formal Methods in Computer Aided Design (FMCAD)*, 247–255.

McDermott, D. 1998. PDDL – the planning domain definition language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Nakhost, H.; Hoffmann, J.; and Müller, M. 2010. Improving local search for resource-constrained planning. In *SoCS*, 81–82.

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks revisited. In *AAAI*, 975–982.

Romein, J. W.; Plaat, A.; Bal, H. E.; and Schaeffer, J. 2009. Transposition table driven work scheduling in distributed search. In *AAAI*, 725–731.

Vidal, V.; Bordeaux, L.; and Hamadi, Y. 2010. Parallel, dynamic k-best-first search: A simple but efficient algorithm for multi-core domain-independent planning. In *SoCS*, 100–107.

Zhou, R., and Hansen, E. 2004. Breadth-first heuristic search. In *ICAPS*, 92–100.

Zhou, R., and Hansen, E. A. 2007. Parallel structured duplicate detection. In *AAAI*, 1217–1222.

Zhou, R.; Schmidt, T.; Hansen, E.; Do, M.; and Uckun, S. 2010. Edge partitioning in parallel structured duplicate detection. In *SoCS*, 137–138.