Large-Scale Parallel State Space Search on the GPU

Damian Sulewski
University of Bremen
sulewski@tzi.de

Abstract

In my thesis I will develop new algorithms to utilize the parallel processing power of current graphics processing units (GPUs) in state space search. The thesis will examine the potential of modern GPUs and compare to the multi-core approach, pointing out the differences in architecture and programming model. This short paper describes the results achieved so far, and gives an outlook into current and future work.

Introduction

Parallel computing has become a widely used standard in modern hardware. Utilizing more than one core was only possible on clusters of computers several years ago, today every workstation includes a *central processing unit* (CPU) with four or more computing cores. Inspired by connected workstations, and motherboards with multiple core chips, multi-core CPUs allow to use each core individually for outstanding tasks realising a *multiple instruction multiple data* (MIMD) structure.

In contrast, graphics processing units, constructed for visualising data on a computer screen evolved in powerful *single instruction multiple data* (SIMD) architecture. Driven by the fact that in graphics processing the same calculation has to be performed on thousands of image points, engineers have assembled hundreds of cores in one chip. This architecture prefers sequential memory access, and dislikes branches in source code.

Recently the GPU producer NVIDIA has presented a general programming interface called *Compute Unified Device Architecture* (CUDA) (Lindholm et al. 2008) allowing programmers to use the graphics card as a *general purpose graphics processing unit* (GPGPU or (GP)^{2}U). This inspired scientists from different domains to develop applications utilizing the GPU. Among others, algorithms for sorting (Leischner, Osipov, and Sanders 2009; Satish, Harris, and Garland 2009) and hashing (Alcantara et al. 2009) have been already presented.

Using external storage mainly, like Munagala and Ranade already proposed in 1999 for undirected graphs, and Korf (2003) for implicit graphs, or using it auxiliary, like Korf

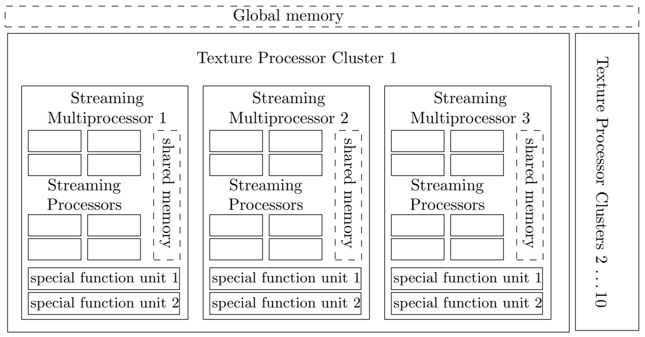
Global memory
Texture Processor Cluster 1
Streaming Multiprocessor 1
Streaming Multiprocessor 2
Streaming Multiprocessor 3
Streaming Processors
shared memory
special function unit 1
special function unit 2
Texture Processor Clusters 2 . . . 10

Figure 1: Sample GPU Architecture (G200 Chipset).

(2008) enabled the search engine to traverse larger search spaces. Externalizing the search space on disk reveals a nearly unlimited storage space with nowadays magnetic hard disks. Combining several of them to a RAID array results in high transfer speeds pushing the bottleneck back to the computing unit. With the recent advances in the technology of solid state drives, used already efficiently in model checking (Šimeček et al. 2008) exploration performance is becoming more of importance than storage.

A rising number of parallel search variants have been studied (Korf and Schultze 2005; Zhou and Hansen 2007) utilizing external memory, being developed for multi-core architecture. This algorithms rely on unsynchronized work flows running on disjoint cores, and are inefficient on SIMD hardware where all cores should execute the same instructions.

GPU Essentials

To motivate the decisions resulting in the studied algorithms the concepts are developed for the state-of-the-art graphics cards hosting NVIDIA GPUs, but portable to other vendors due to transferable basics.

As mentioned above, the GPU architecture mimics a SIMD computer with the same instructions running on all processors. It supports different layers of memory access, forbids simultaneous writes but allows concurrent reads from one memory cell.

Consider the G200 chipset[1], as found in NVIDIA GPUs and illustrated in Figure 1, a core is a *streaming processor* (SP). 8 SPs are grouped together to one *streaming multiprocessor* (SM), and used like ordinary SIMD processors. Each of the 10 *texture processor clusters* (TPCs) combines 3 SMs, yielding 240 cores in one chip.

Memory, visualized dashed in the figure, is structured hierarchically, starting with the GPU's global memory (video RAM, or VRAM). Access to this memory is slow, but can be accelerated through *coalescing*, where adjacent accesses are combined to one access. Each SM includes 16 KB of memory (SRAM), which is shared between all SPs and can be accessed at the same speed as registers. Additional registers are also located in each SM but not shared between their SPs. Data has to be copied from the systems main memory to the VRAM to be accessible by the threads.

The GPU programming language links to ordinary C-sources. The function executed in parallel on the GPU is called *kernel*. The kernel is driven by threads, grouped together in *blocks*. All the SPs get the same chunk of code, so that SPs in an else-branch wait for the SPs operating in the according if-branch, being idle. After all threads have completed a chunk of code, the next one is executed. Threads waiting for data can be parked by the SM, while the SPs work on threads which have already received their data.

Achievements

Probabilistic Model Checking on the GPU

Significant runtime gains can be achieved exploiting the power of GPUs in probabilistic model checking. This is because basic algorithms for probabilistic model checking are based on matrix-vector multiplication. These operations enjoy very efficient implementation on GPUs. Because of the massive parallelism impressive speedups with regard to the sequential counterparts of the algorithms are common. The algorithm developed in (Bošnački, Edelkamp, and Sulewski 2009) is a parallel adaptation of the iteration method of Jacobi for solving linear equation based on a sequence of matrix-vector products. Jacobi's method chosen over other methods that usually outperform it on sequential platforms because of its lower memory requirements and potential to be parallelized because of fewer data dependencies. The algorithm features sparse matrix vector multiplication. It requires a minimal number of copy operations from RAM to GPU and back, and is implemented on top of the probabilistic model checker PRISM (Kwiatkowska, Norman, and Parker 2002). The prototype implementation was evaluated on several case studies and remarkable speedups (up to factor 18) were achieved compared to the sequential version of the tool.

Explicit State Model Checking on the GPU

My thesis also pioneers applying GPGPU technology to explicit-state model checking. This work covers the entire model checking process, including checking enabledness and generating the successors on the GPU. Meanwhile, a different approach to explicit-state GPU-based model has been published (Barnat et al. 2009) that transforms (MAP) LTL model checking to a matrix multiplication problem to apply fast operations on the graphics card. The speed-ups are considerable, but the approach applies to small models only. A conceptually different algorithm was presented by me in 2009, suited to parallel model checking large models. In such large-scale verification state spaces are likely to be too big to fit into main memory. Moreover, for very small models, the overhead of moving data from the CPU to the GPU and back can be larger than the savings obtained on the GPU. My thesis focuses on breadth-first search (BFS) to generate the entire state space. BFS is sufficient for the verification of safety properties. As identified e.g. in (Barnat et al. 2008), complete state space construction via BFS can be the performance bottleneck for large-scale external-memory checking of LTL. Last, but not least, BFS is also the basis for constructing a perfect hash function on disk, the basis for semi-external (Edelkamp, Sanders, and Šimeček 2008) and hash-memory efficient model checking (Edelkamp and Sulewski 2008). After having generated the state space on the hard disk, its compression is considerably fast. Sorting-based external-memory BFS (Korf 2003; Stern and Dill 1996) bares three computational intensive tasks, all portable to the GPU. Hence, the developed algorithm divides into three different stages applied to each BFS layer: 1) test the applicability of transitions against the current state; 2) generate the set of successors for all states and enabled transitions. 3) apply delayed duplicate detection by sorting and scanning all successors. The first two stages require transition guards and value assignments to stay in the GPU's global memory. Here, the polish reverse notation is chosen, since it offers the possibility to concatenate all transition descriptions to one integer vector and a memory efficient exploration. For all three stages significant individual speed-ups of more than one order of magnitude are obtained for analyzing benchmark protocols on the GPU. The overhead in combining the results of the different stages in the CPU and the I/O bandwidth limitation limits the – still noticeable – overall speed-ups.

Exploration of Single-Player Games on the GPU

Cooperman & Finkelstein (1992) show that, given a minimal reversible perfect hash function, two bits per state are sufficient to conduct a complete breadth-first exploration of the search space. The running time of their approach is determined by the size of the search space times the maximum breadth-first layer (times the efforts to generate the children). Their algorithm uses two bits, encoding numbers from 0 to 3, with 3 denoting an unvisited state, and 0, 1, and 2 denoting the current depth value modulo 3. The main effect is that this allows to separate newly generated states and visited states from the current layer.

For the search we require certain characteristics of hash functions.

Definition 1 (Hash Function) *A* hash function h *is a mapping of some universe* U *to an index set* $[0, \ldots, m-1]$.

[1]Next generation GPUs effectively only differ in the amount of the components found on the chip and the sizes of the memories.

The set of reachable states S is a subset of U, i.e., $S \subseteq U$. An important class are injective hash functions.

Definition 2 (Perfect Hash Function) *A hash function $h : U \to [0, \ldots, m-1]$ is* perfect*, if for all $s \in S$ with $h(s) = h(s')$ we have $s = s'$.*

Definition 3 (Space Efficiency) *The space efficiency of h is the proportion $\lceil m/|S| \rceil$ of available hash values to states.*

Definition 4 (Minimal Perfect Hash Function) *A perfect hash function h is* minimal *if its space efficiency is* 1.

A minimal perfect hash function is a one-to-one mapping from the state space S to the set of indices $\{0, \ldots, |S|-1\}$, i.e., $m = |S|$. In contrast to open-addressed or chained hash table, perfect hash functions allow direct-addressing of bit-state hash tables, This allows to compress the set of visited states *Closed*. The other important property to also compress the list of frontier nodes *Open* is that the state vector can be reconstructed given the hash value, e.g. indicated by the position in a bit vector.

Definition 5 (Reversible Hash Function) *A perfect hash function h is* reversible*, if given $h(s)$, the state $s \in S$ can be reconstructed. A reversible minimum perfect hash function is called* rank*, while the inverse is called* unrank.

We will see that for an implicit exploration of the search space in which array indices serve as state descriptors, reversible hash functions are required.

Permutation Rank as a Hash Function Korf and Schultze (2005) use two lookup tables with a space requirement of $O(2^N \log N)$ bits to compute the lexicographic ranks (and their inverse).[2] Bonet (2008) discusses time-space trade-offs and provides a uniform algorithm that takes $O(N \log N)$ time and $O(N)$ space. As we are not aware of any $O(N)$ time and $O(N)$ space algorithm for lexicographic ranking and unranking, we studied the ordering induced by Myrvold and Ruskey (2001) in their *rank1* and *unrank1* functions. .

Binomial Coefficient as a Hash Function For states consisting of a fixed number of Boolean variables it suffices to store which of them is assigned to true to identify each state. Traversing the search graph and generating successors flips the status of individual state variables depending on the successor generating function. If the order of the variables is fixed and the number of satisfied bits given, we can identify their position using a binomial coefficient. A binomial coefficient $\binom{k}{n}$ is the number of possible k-sets in a set of n elements. Algorithm 1 describes how to assert a unique rank to a given state. Since the number of k-sets in a n-set is known, we can impose an ordering on these k-sets. This ordering is given by the position of the variables that are satisfied. The algorithm uses the number t of satisfied variables and starts with a rank $r = 0$. For each unsatisfied variable r is increased by the binomial coefficient given by the position of this entry and the number of the remaining satisfied variables.

[2]In a lexicographic ordering $(a, b) < (a', b')$ iff $a < a'$ or $a = a'$ and $b < b'$.

Algorithm 1 *Binomial-Rank*(s)

```
1: $i := 0; r := 0$
2: $t :=$ number of true values in $s$
3: while $t > 0$ do
4:     $i := i + 1$
5:     if $s_i = 1$ then
6:         $t := t - 1$
7:     else
8:         $r := r + \binom{n-i}{t-1}$
9: return $r$
```

Using the presented perfect hash functions and additional properties we can externalize and parallelize the algorithm presented in (Cooperman and Finkelstein 1992) to use the GPU efficiently. While the bitvectors are stored on the external memory, partitioned into smaller chunks if possible, only the ranks are transferred to the GPU memory. The GPU transforms the rank into a representative of a state and generates all successors of it, whose ranks are transferred back to the CPU and converted into positions in th bit-vectors. Using this strategy significant speedups were achieved for different problem domains.

Exploration of Two-Player Games on the GPU

Multinomial coefficients can be used to compress state vectors sets with a fixed but permuted value assignment, e.g., board games state (sub)sets where the number of pieces for each player does not change. For p players in a game on n positions we use k_i with $1 \leq i \leq p$ to denote the number of game pieces owned by player i, and k_{n+1} for the remaining empty positions.

Definition 6 *For natural numbers n, k_1, k_2, …, k_m with $n = k_1 + k_2 + \ldots + k_m$ the multinomial coefficient is defined as*

$$\binom{n}{k_1, k_2, \ldots, k_m} := \frac{n!}{k_1! \cdot k_2! \cdot \ldots \cdot k_m!}.$$

Since $\sum_{i=0}^{p+1} k_i = n$ we can deduce value k_{p+1} given $k_1, k_2, \ldots, k_p$. We present multinomial hashing for $p = 2$ but the extension to three and more players is intuitive.

We will write $\binom{n}{k_1, k_2}$ for $\binom{n}{k_1, k_2, k_3}$ with $k_3 = n - (k_1 + k_2)$ and distinguish pieces by enumerating their *colors* with 1, 2, and 0 (empty).

Let S_{k_1,k_2} be the set of all possible boards with k_1 pieces of color 1 and k_2 pieces in color 2. The computation of the rank for states in S_{k_1,k_2} is provided in Algorithm 2.

Using the value of the corresponding multinomial coefficient as a hash function the results of (Gasser 1996) were verified, proving his results in a strong solution complete state space search on a PC, while the paper applies pruning on a cluster.

Discussion and Outlook

The general aim of my work is to develop parallel algorithms for enumerating large state spaces exceeding the main memory of systems utilizing parallel external drives. The achieved results identify the GPU not only as a powerfull

Algorithm 2 Rank

```
Require: Game state vector: $state[0, \dots, n-1]$,
    number of pieces in color 1: $l_{ones}$,
    number of pieces in color 2: $l_{twos}$
Ensure: Rank: $r \in \{0, \dots, |S|\}$
1: $i \leftarrow 0, r \leftarrow 0$
2: while $i < n$ do
3:   if $state[i] = 2$ then
4:     $l_{twos} \leftarrow l_{twos} - 1$
5:   else if $state[i] = 1$ then
6:     if $l_{twos} > 0$ then
7:       $r \leftarrow r + \binom{n-i-1}{l_{ones}, l_{twos}-1}$
8:     $l_{ones} \leftarrow l_{ones} - 1$
9:   else
10:    if $l_{twos} > 0$ then
11:      $r \leftarrow r + \binom{n-i-1}{l_{ones}, l_{twos}-1}$
12:    if $l_{ones} > 0$ then
13:      $r \leftarrow r + \binom{n-i-1}{l_{ones}-1, l_{twos}}$
14:   $i \leftarrow i + 1$
15: return $r$
```

coprocessor for the CPU, but also as an efficient main actor in planning. The remaining work for this thesis will be to generalize the strategies applied to domain specific problems, to use them independent of the problem domain.

References

Alcantara, D. A.; Sharf, A.; Abbasinejad, F.; Sengupta, S.; Mitzenmacher, M.; Owens, J. D.; and Amenta, N. 2009. Real-time parallel hashing on the gpu. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia 2009)* 28(5).

Barnat, J.; Brim, L.; Šimeček, P.; and Weber, M. 2008. Revisiting resistance speeds up i/o-efficient ltl model checking. In Ramakrishnan, C. R.; Rehof, J.; Ramakrishnan, C. R.; and Rehof, J., eds., *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, 48–62. Springer.

Barnat, J.; Brim, L.; Češka, M.; and Lamr, T. 2009. CUDA accelerated LTL Model Checking. In *15th International Conference on Parallel and Distributed Systems (ICPADS 2009)*, 34–41. IEEE Computer Society.

Bonet, B. 2008. Efficient algorithms to rank and unrank permutations in lexicographic order. In *AAAI-Workshop on Search in AI and Robotics*.

Bošnački, D.; Edelkamp, S.; and Sulewski, D. 2009. Efficient probabilistic model checking on general purpose graphics processors. In *Proceedings of the 16th International SPIN Workshop on Model Checking Software*. Berlin, Heidelberg: Springer-Verlag. 32–49.

Cooperman, G., and Finkelstein, L. 1992. New methods for using cayley graphs in interconnection networks. *Discrete Applied Mathematics* 37/38:95–118.

Edelkamp, S., and Sulewski, D. 2008. Flash-efficient ltl model checking with minimal counterexamples. In *SEFM '08: Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, volume 0, 73–82. Washington, DC, USA: IEEE Computer Society.

Edelkamp, S.; Sanders, P.; and Šimeček, P. 2008. Semi-external ltl model checking. In Gupta, A.; Malik, S.; Gupta, A.; and Malik, S., eds., *CAV*, volume 5123 of *Lecture Notes in Computer Science*, 530–542. Springer.

Gasser, R. 1996. Solving nine men's morris. *Computational Intelligence* 12:24–41.

Korf, R. E., and Schultze, P. 2005. Large-scale parallel breadth-first search. In Veloso, M. M.; Kambhampati, S.; Veloso, M. M.; and Kambhampati, S., eds., *AAAI*, 1380–1385. AAAI Press / The MIT Press.

Korf, R. E. 2003. Delayed duplicate detection: extended abstract. In *IJCAI'03: Proceedings of the 18th international joint conference on Artificial intelligence*, 1539–1541. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Korf, R. E. 2008. Minimizing disk I/O in two-bit breadth-first search. In *AAAI'08: Proceedings of the 23rd national conference on Artificial intelligence*, 317–324. AAAI Press.

Kwiatkowska, M.; Norman, G.; and Parker, D. 2002. Prism: Probabilistic symbolic model checker. 200–204. Springer.

Leischner, N.; Osipov, V.; and Sanders, P. 2009. Gpu sample sort.

Lindholm, E.; Nickolls, J.; Oberman, S.; and Montrym, J. 2008. Nvidia tesla: A unified graphics and computing architecture. *Micro, IEEE* 28(2):39–55.

Munagala, K., and Ranade, A. 1999. I/o-complexity of graph algorithms. In *SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, 687–694. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics.

Myrvold, W., and Ruskey, F. 2001. Ranking and unranking permutations in linear time. *Information Processing Letters* 79(6):281–284.

2009. *Efficient Explicit-State Model Checking on General Purpose Graphic Processors*.

Satish, N.; Harris, M.; and Garland, M. 2009. Designing efficient sorting algorithms for manycore gpus. *Parallel and Distributed Processing Symposium, International* 0:1–10.

Stern, U., and Dill, D. L. 1996. Combining state space caching and hash compaction. In *In Methoden des Entwurfs und der Verifikation digitaler Systeme, 4. GI/ITG/GME Workshop*, volume 4, 81–90.

Šimeček, P.; Sulewski, D.; Edelkamp, S.; Brim, L.; and Barnat, J. 2008. Can flash memory help in model checking? In *13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2008)*. ERCIM.

Zhou, R., and Hansen, E. A. 2007. Parallel structured duplicate detection. In *AAAI'07: Proceedings of the 22nd national conference on Artificial intelligence*, 1217–1223. AAAI Press.