Block-Parallel IDA* for GPUs

Satoru Horie, Alex Fukunaga
Graduate School of Arts and Sciences
The University of Tokyo

Abstract

We investigate GPU-based parallelization of Iterative-Deepening A* (IDA*). We show that straightforward thread-based parallelization techniques which were previously proposed for massively parallel SIMD processors perform poorly due to warp divergence and load imbalance. We propose Block-Parallel IDA* (BPIDA*), which assigns the search of a subtree to a block (a group of threads with access to fast shared memory) rather than a thread. On the 15-puzzle, BPIDA* on a NVIDIA GRID K520 with 1536 CUDA cores achieves a speedup of 4.98 compared to a highly optimized sequential IDA* implementation on a Xeon E5-2670 core.

1 Introduction

Graphical Processing Units (GPUs) are many-core processors which are now widely used to accelerate many types of computation. GPUs are attractive for combinatorial search because of their massive parallelism. On the other hand, on many domains, search algorithms such as A* tend to be limited by RAM rather than runtime. A standard strategy for addressing limited memory in sequential search is iterative deepening (Korf 1985). We present a case study on the GPU-parallelization of Iterative-Deepening A* (Korf 1985) for the 15-puzzle using the Manhattan Distance heuristic. We evaluate previous thread-based techniques for parallelizing IDA* on SIMD machines, and show that these do not scale well due to poor load balance and warp divergence. We then propose Block-Parallel IDA* (BPIDA*), which, instead of assigning a subtree to a single thread, assigns a subtree to a group of threads which share fast memory. BPIDA* achieves a speedup of 4.98 compared to a state-of-the-art 15-puzzle solver on a CPU, and a speedup of 659.5 compared to a single-thread version of the code running on the GPU.

2 Background and Related Work

An NVIDIA CUDA architecture GPU consists of a set of *streaming multiprocessors (SMs)* and a GPU main memory (shared among all SMs). Each SM consists of shared memory, cache, registers, arithmetic units, and a warp scheduler. Within each SM the cores operate in a SIMD manner. However, each SM executes independently, so threads in different SMs can run asynchronously. A *thread* is the smallest unit of execution. A *block* is a group of threads which execute on the same SM and share memory. A *grid* is a group of blocks which execute the same function. Threads in a block are partitioned into *warps*. A warp executes in a SIMD manner (all threads in the same warp share a program counter). *Warp divergence*, an instance of *SIMD divergence*, occurs when threads belonging to the same warp follow different execution paths, e.g., IF-THEN-ELSE branches. *Shared memory* is shared by a block and is local within a SM, and access to shared memory is much faster than access to the GPU *global memory* which is shared by all SMs.

Rao et al (1987) parallelized each iteration of IDA* using work-stealing on multiprocessors. Parallel-window IDA* assigned each iteration of IDA* to its own processor (Powley and Korf 1989). Two SIMD parallel IDA* algorithms are by Powley et al (1993) and Mahanti and Daniels (1993). For each f-cost limited iteration of IDA*, they perform an initial partition of the workload among the processors, and then periodically perform load balancing between IDA* iterations and within each iteration. Hayakawa et al (2015) proposed a GPU-based parallelization of IDA* for the 3x3x3 Rubik's cube which searches to a fixed depth l on the CPU, then invokes a GPU kernel for the remaining subproblems. Their domain-specific load balancing scheme relies on tuning l using knowledge of "God's number" (optimal path length for the most difficult cube instance) and is fragile – perturbing l by 1 results in a 10x slowdown. Zhou and Zeng (2015) proposed a GPU-parallel A* which partitions OPEN into thousands of priority queues. The amount of global RAM on the GPU (currently $\leq 24GB$) poses a serious limitation for GPU-based parallel A*. Edelkamp and Sulewski (2010) investigated memory-efficient GPU search. Sulewski et al (2011) proposed a hybrid planner which uses both the GPU and CPU.

3 Experimental Settings and Baselines

We used the standard set of 100 15-puzzle instances by Korf (1985). These instances are ordered in approximate order of difficulty. All solvers used the Manhattan distance heuristic. Reported runtimes include all overheads such as data transfers between CPU and GPU memories (negligible). All experiments were executed on a non-shared, dedicated AWS EC2 g2.2xlarge instance. The CPU is an Intel Xeon E5-

2670. The GPU is an NVIDIA GRID K520, with 4GiB global RAM, 48KiB shared RAM/block, 1536 CUDA cores, warp size 32, and 0.80GHz GPU clock rate.

First, we evaluated 3 baseline IDA* solvers:

Solver B: The efficient, Manhattan-Distance heuristic based 15-puzzle IDA* solver implemented in C++ by Burns et al. (2012). We used the current version at https://github.com/eaburns/ssearch.

Solver C: Our own implementation of IDA* in C (code at http://github.com/socs2017-48/anon48), This is the basis for G1 and all of our GPU-based code.

Solver G1: A direct port of Solver C to CUDA. The implementation is optimized so that all data structures are in the fast, shared memory (the memory which is local to a SM). This baseline configuration uses only 1 GPU block/thread, i.e., only 1 core is used, all other GPU cores are idle.

The total time to solve all 100 problem instances was 620 seconds for Solver B (Burns et al. 2012) and 475 seconds for our Solver C. Solver C was consistently 25% faster on every instance. Thus, Solver C is appropriate as a baseline for our GPU-based 15-puzzle solvers.

Next, we compare Solver C (1 CPU thread) to G1 (1 GPU thread). G1 required 62957 seconds to solve all 100 instances, 131 times slower than Solver C. This implies that on the GPU we used with 1536 cores, a perfectly efficient implementation of parallel IDA* might be able to achieve a speedup of up to 1536/131 = 11.725 compared to Solver C.

4 Thread-Based Parallel IDA*

Most of the previous work on parallel IDA* parallelizes each iteration of IDA* using a *thread-based parallel* scheme (Rao, Kumar, and Ramesh 1987; Powley, Ferguson, and Korf 1993; Mahanti and Daniels 1993; Hayakawa, Ishida, and Murao 2015).

We evaluated 3 thread-parallel IDA* configurations. Since these are relatively straightforward and not novel, we sketch the implementations below. Details are in the extended version (Horie and Fukunaga 2017).

PSimple (baseline) In this baseline configuration, for each f-bounded iteration of IDA*, PSimple performs A* search from the start state until as many unique states as the # of threads are in OPEN. Then, each root is assigned to a thread. No load balancing is performed. The subtree sizes under each root state can vary significantly, so some threads may finish their subproblem much faster than other threads. Each f-bounded iteration must wait for all threads to complete, so PSimple has very poor load balance. Therefore, *load balancing* mechanisms which redistribute the work among processors are necessary.

PStaticLB (static load balancing) This configuration adds static load balancing to PSimple. After each f-bounded iteration, PStaticLB implements a *static load balancing* mechanism somewhat similar to that of (Powley, Ferguson, and Korf 1993). In IDA*, the i-th iteration repeats all of the work done in iteration $i-1$. Thus, the # of states visited under each root state in the iteration $i-1$ can be used to estimate the # of states which will be visited in the current iteration i, and root nodes are redistributed based on these estimates (details in (Horie and Fukunaga 2017)).

PFullLB (thread-parallel with dynamic load balancing) This configuration adds dynamic load balancing (DLB) to PStaticLB, which moves work to idle threads from threads with remaining work *during* an iteration. On a GPU, work can be transferred between two threads *within* a single block relatively cheaply using the shared memory within a block, while transferring work between two threads in different blocks is expensive because it requires access to the global memory. When dynamic load balancing is triggered, idle threads steal work from threads with remaining work within a block. We experimented with various DLB strategies including variants of policies investigated by (Powley and Korf 1989; Mahanti and Daniels 1993), and used a policy we found for triggering DLB based on the policy by Powley and Korf. See (Horie and Fukunaga 2017) for additional details.

4.1 Evaluation of Thread-Parallel IDA*

PSimple on 1536 cores required a total of 3378 seconds to solve all 100 problems, a speedup of only 18.6 compared to G1 (1 core on the GPU). This is mostly due to extremely poor load balance. We define load balance as $maxload/averageload$, where $averageload$ is the average number of nodes expanded among all threads, and $maxload$ is the number of states expanded by the thread which performed the most work. The load balance for PSimple on the 100 problems was: mean 96.46, min 14, max 680, stddev 113.19. This is extremely unbalanced ($maxload$ is almost 100x $averageload$).

Static load balancing significantly improved load balance (PStaticLB: mean 9.96, min 3, max 56, stddev 8.96), and dynamic load balancing further improved load balance (PFullLB: mean 6.14 min 3 max 19 stddev 3.38). This resulted in speedups of 58.9 and 70.8 compared to G1 (Table 1). However, the 70.8 speedup vs G1 achieved by PFullLB is only a parallel efficiency of 70.8/1536 = 4.6%, which is extremely poor. We experimented extensively but could not achieve significantly better results with thread-parallel IDA*.

5 Block Parallelization

The likely causes for the poor (4.6%) efficiency of PFullLB are: (1) SMs become idle due to poor load balance even after our load balancing efforts, (2) threads stall for warp divergence, and (3) load balancing overhead. All of these can be attributed to the thread-based parallelization scheme in PFullLB and PStaticLB, in which each processor/thread executes an independent subproblem during a single f-bound iteration. This scheme, based on parallel IDA* variants originally designed for SIMD machines (Powley, Ferguson, and Korf 1993; Mahanti and Daniels 1993), was appropriate for those SIMD architectures where all communications between processors were very expensive – paying the price of SIMD divergence overhead was preferable to incurring communication costs. On the other hand, in NVIDIA GPUs, threads in the same block (which execute on the same SM) can access fast shared memory on the SM with relatively low overhead. We exploit this in a *block-parallel* approach.

Rocki and Suda (2009) proposed a GPU-based parallel minimax gametree search algorithm for 8x8 Othello (without any $\alpha\beta$ pruning) which works as follows. Within each block a node n is selected for expansion. If n is a leaf, it is evaluated using a parallel evaluation function (32 threads, 1 thread per 2 positions in the 8x8 board). Otherwise a parallel successor generator function is called (1 thread/position) to generate successors of n, which are added to the node queue. This approach greatly reduced warp divergence, since all threads in the warp are synchronized to perform the fetch-evaluate-expand cycle. Because there is no $\alpha\beta$ pruning, their search trees have uniform depth (i.e., fixed-depth DFS), and also, the # of possible moves on the othello board (64) conveniently matched a multiple of the CUDA warp size (32).

We now propose a generalization of this approach for IDA*, which we call *Block-Parallel IDA**, shown in Alg. 1. In contrast to the parallel minimax of (Rocki and Suda 2009), BPIDA* handles variable-depth subtrees (due to the heuristic, IDA* tree depths are irregular) and does not depend on a fixed number of applicable operators (e.g., 64).

$openList$ is a stack which is shared among all threads in the same block, which supports two key parallel operations: `parallelPop` and `atomicPut`. `parallelPop` extracts $(\#threads\_in\_a\_block/\#operators)$ nodes from $openList$. `atomicPut` inserts nodes in t into the shared $openList$ concurrently. This is implemented as a linearizable (Herlihy and Wing 1990) operation.

The `BPDFS` function is similar to a standard, sequential f-limited depth-first search, but in each iteration of the repeat-until loop in lines 4-16 (Alg. 1), a warp performs the fetch-evaluate-expand cycle on $(\#threads\_in\_a\_block/\#operators)$ nodes. The number of threads per block is set to the warp size (32). This allows the following: (1) When a warp is scheduled for execution, all cores in the SM are active. (2) Since all threads in the block (=warp) share a program counter, explicit synchronizations become unnecessary.

BPIDA* applies a slightly modified version of the static load balancing used by PStaticLB (Sec. 4). While PStaticLB uses the number of expanded nodes to estimate the work in the next iteration, BPIDA* uses the number of repetitions executed in lines 4-16. BPIDA* does not use dynamic load balancing.

6 Evaluation of BPIDA*

Runtimes Figure 1a compares the relative runtime of BPIDA* vs. PFullLB. BPIDA* required a total of 95 seconds to solve all 100 problems, a speedup of 9.39 compared to PFullLB. Table 1 summarizes the total runtimes and speedups for all algorithms in this paper.

Other metrics There are 3 suspected culprits for the poor performance of thread-based parallel IDA*: (1) dynamic load overhead, (2) idle SMs (bad load balance), and (3) thread stalls for warp divergence. BPIDA* doesn't perform dynamic load balancing, so (1) is irrelevant. For factors (2) and (3), there are related metrics, sm_efficiency and IPC(instructions per cycle), which can be measured by the CUDA profiler, nvprof. sm_efficiency is the average % of time at least one warp is active on a SM. High sm_efficiency shows how busy the SMs are, and high IPC indicates there are few NOPs due to warp divergence. The (mean, min, max, stddev) sm_efficiency over 100 instances was (65.22, 31.8, 82.7, 7.94) for PFullLB, and (94.29, 32.3, 99.9, 9.76) for BPIDA*, and for IPC, the results were (0.30, 0.13, 0.39, 0.048) for PFullLB and (0.97, 0.60, 1.06, 0.059) for BPIDA*. For both metrics, the results of BPIDA* were better than PFullLB, and close to the ideal values (100% sm_efficiency and IPC=1.0).

Algorithm 1 BlockParallel IDA*

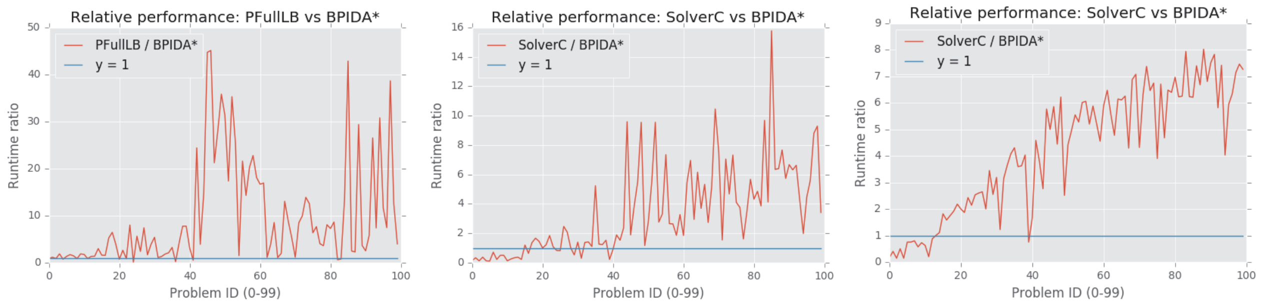
```
 1: function BPDFS($root, goals, limit_f$)
 2:   $openList \leftarrow root$
 3:   $f_{next} = \infty$
 4:   repeat
 5:     $s \leftarrow$ PARALLELPOP($openList$)
 6:     if $s \in goals$ then
 7:       return $s$ and its parents as a shortest path
 8:     $a \leftarrow (threadID \mod \#actions)$th action
 9:     if $a$ is applicable on $s$ then
10:       $t \leftarrow successor(a, s)$
11:       $f_{new} \leftarrow g(s) + cost(a) + h(t)$
12:       if $f_{new} <= limit_f$ then
13:         ATOMICPUT($openList, t$)
14:       else
15:         $f_{next} \leftarrow min(f_{next}, f_{new})$
16:   until $openList$ is empty
17:   return $f_{next}$                              ▷ no plan is found
18:
19: function BPIDA*($start, goals$)
20:   $roots \leftarrow$ CREATEROOTSET($start, goals$)
21:   $limit_f \leftarrow$ DECIDEFIRSTLIMIT($roots$)
22:   repeat
23:     parallelForByBlocks $root \in roots$ do
24:       $limit_f, stat \leftarrow$ BPDFS($root, goals, limit_f$)
25:     end parallelForByBlocks
26:   until shortest path is found
```

6.1 Comparison with Sequential Solver C

We now compare BPIDA* with the CPU-based, sequential Solver C (Sec. 3). Fig. 1b compares the relative runtimes of

configuration	total runtime (seconds)	speedup vs. G1
CPU-based sequential algorithms (1 CPU thread)		
Solver B (Burns et al. 2012)	620	n/a
Solver C	475	n/a
GPU-based sequential algorithm (1 thread)		
G1	62957	1
GPU-based parallel algorithms (1536 threads)		
PSimple	3378	18.6
PStaticLB	1069	58.9
PFullLB	892	70.8
BPIDA*	**95**	**659.5**

Table 1: Total Runtimes for 100 15-Puzzle Instances



(a) Relative runtimes: PFullLB vs. BPIDA* (finding 1 optimal solution)

(b) Relative runtimes: Solver C vs. BPIDA* (finding 1 optimal solution)

(c) Relative runtimes: Solver C vs. BPIDA* (finding all optimal solutions)

Figure 1: BP-IDA* Evaluation

Solver C (1 CPU core) and BPIDA* (1536 GPU cores). The y-axis shows Runtime(SolverC)/Runtime(BPIDA*) for each instance. Comparing the total time to solve all 100 instances, BPIDA* was 4.98 times faster.

Runtime comparisons between parallel vs. sequential IDA* can be obfuscated by the fact that they do not necessarily expand the same set of nodes in the final iteration, although they expand the same set of nodes in non-final iterations (the same issue exists with comparisons among parallel IDA* variants, but from Fig. 1a and Table 1, it is clear that BPIDA* significantly outperforms the other parallel algorithms, so above, we simply reported the time to find a single solution, as is standard practice in previous works).

To eliminate differences in search efficiency (node expansion order) from the comparison, the next experiment compares the time required to find *all optimal-cost solutions* of every problem, i.e., the search does not terminate until all nodes with $f \leq OptimalCost$ have been expanded. This eliminates node ordering effects, allowing comparison of the wall-clock time required to perform the same amount of search. Fig. 1c compares the relative runtimes of Solver C (1 CPU core) and BPIDA* (1536 GPU cores). The y-axis shows Runtime(SolverC)/Runtime(BPIDA*) to find all optimal solutions for each instance. Comparing the total time to find all optimal solutions for all 100 instances, BPIDA* was 6.78 times faster.

7 Conclusions and Future Work

We proposed Block-Parallel IDA*, which assigns subtrees to GPU blocks (groups of threads with fast shared memory). Compared to thread-parallel approaches, this greatly reduces warp divergence and improves load balance. BPIDA* also does not require explicit dynamic load balancing, making it relatively simple to implement. On 1536 cores, BPIDA* achieves a speedup of 659.5 vs. a 1-thread GPU baseline, i.e., 42% parallel efficiency. Compared to a highly optimized single-CPU IDA*, BPIDA* achieves a 6.78x speedup when comparing the time to find all optimal solutions.

The successful parallelization of BPIDA* on the 15-puzzle with Manhattan distance (MD) heuristic exploits the following factors: (1) compact states, (2) the MD heuristic requires little memory, and (3) standard IDA* doesn't perform duplicate state detection. Thus, all work could be performed in the SM local+shared memories, without using global memory. In many domains, data structures representing each state are larger and the IDA* state stacks will not fit in local memory. Also, some powerful memory-intensive heuristics, e.g,. PDBs (Korf and Felner 2002), will require at least the use of global memory. Finally, standard approaches for reducing duplicate state expansion, e.g., transposition tables (Reinefeld and Marsland 1994) requires significant memory. Thus, future work will focus on methods which use GPU global memory effectively so that domains with larger states, memory-intensive heuristics, and memory-intensive duplicate pruning techniques can be used.

Acknowledgments

This research was supported by JSPS KAKENHI grants 25330253 and 17K00296.

References

Burns, E. A.; Hatem, M.; Leighton, M. J.; and Ruml, W. 2012. Implementing fast heuristic search code. In *Proc. SoCS*.

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

Hayakawa, H.; Ishida, N.; and Murao, H. 2015. GPU-acceleration of optimal permutation-puzzle solving. In *Proceedings of the 2015 International Workshop on Parallel Symbolic Computation*, 61–69. ACM.

Herlihy, M. P., and Wing, J. M. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12(3):463–492.

Horie, S., and Fukunaga, A. 2017. Block-parallel IDA* for GPUs (extended version). http://arxiv.org/abs/1705.02843.

Korf, R. E., and Felner, A. 2002. Disjoint pattern database heuristics. *Artificial intelligence* 134(1):9–22.

Korf, R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence* 27(1):97–109.

Mahanti, A., and Daniels, C. J. 1993. A SIMD approach to parallel heuristic search. *Artificial Intelligence* 60(2):243–282.

Powley, C., and Korf, R. E. 1989. Single-agent parallel window search: A summary of results. In *Proc. IJCAI*, 36–41.

Powley, C.; Ferguson, C.; and Korf, R. E. 1993. Depth-first heuristic search on a SIMD machine. *Artificial Intelligence* 60(2):199–242.

Rao, V. N.; Kumar, V.; and Ramesh, K. 1987. A parallel implementation of iterative-deepening-a*. In *Proc. AAAI*, 178–182.

Reinefeld, A., and Marsland, T. A. 1994. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 16(7):701–710.

Rocki, K., and Suda, R. 2009. Parallel minimax tree searching on GPU. In *International Conference on Parallel Processing and Applied Mathematics*, 449–456. Springer.

Sulewski, D.; Edelkamp, S.; and Kissmann, P. 2011. Exploiting the computational power of the graphics card: Optimal state space planning on the GPU. In *Proc. ICAPS*.

Zhou, Y., and Zeng, J. 2015. Massively parallel A* search on a GPU. In *Proc. AAAI*, 1248–1255.