Multi-GPU sliding tile puzzle solving with GA*, Groute and Abstract Zobrist Hashing

Håvard Pettersson

Master of Science in Computer Science
Submission date: August 2018
Supervisor: Anne Cathrine Elster, IDI

Norwegian University of Science and Technology
Department of Computer Science

Problem Description

Multi-GPU systems are more and more prevalent and provide more compute power than ever. This thesis project will explore how such systems can be used for graph algorithms. In particular we will look at A*, a core pathfinding and graph traversal algorithm. A* is an extension of Dijkstra's algorithm and has recently gotten renewed interest in the AI literature. It has previously been developed for multi-threading and the GPU, but we have not seen a multi-GPU version.

This work will build on Groute, a cutting-edge multi-GPU framework for asynchronous irregular processing, introduced by Ben-Nun, Sutton, Pai, and Pingali in 2017. Our work on implementing A* for multi-GPU on Groute should thus both shed light on how to explore Groute, as well as show how suitable A* is for multi-GPU implementation.

Abstract

The A* algorithm has been a key search algorithm and part of AI literature for a long time. It has applications in computational biology, natural language processing, pathfinding, puzzle solving, and more. The complexity of some of these applications has brought about many improvements and variations of the original algorithm, and recent research has shown that using GPUs to accelerate A* search can achieve substantial speed-ups over conventional CPU-based implementations.

Compute nodes with multiple GPUs have become commonplace, and techniques and methods for programming such systems is a popular area of research. We bring to life the first implementation of the A* search algorithm that distributes the search space across multiple GPU devices, and identify challenges that must be addressed in order to achieve appreciable performance.

Our implementation is based on the GA* algorithm of Zhou and Zeng, and builds on the asynchronous multi-GPU programming framework Groute by Ben-Nun, Sutton, Pai, and Pingali. We use Abstract Zobrist Hashing for distributing the search space across GPUs. The implementation is benchmarked by solving sliding tile puzzles, a commonly used benchmark for search algorithms.

Our multi-GPU version of A* allows for solving more difficult search problems than a single GPU can handle by utilizing the increased memory capacity of multiple GPUs. However, it does not in general yield improved performance in terms of wall-clock runtime when increasing the number of GPUs used. We propose how to eliminate much of the overhead introduced when scaling horizontally, and show that if this is done successfully, each additional GPU added may yield a search rate increase of at least 50% of the performance of a lone GPU.

Sammendrag

A*-algoritmen har lenge vært en viktig søkealgoritme og del av kunstig intelligens-litteraturen. Den har anvendelser i beregningsbiologi, naturlig språkbehandling, stifinning, puslespillløsning med flere. Kompleksiteten til noen av disse anvendelsene har ledet til mange forbedringer og nye varianter av den originale algoritmen, og nylig forskning har vist at å bruke GPU-er for å akselerere A*-søk kan oppnå betydelige forbedringer i ytelse over CPU-baserte implementeringer.

Beregningsnoder med flere GPU-er har blitt vanlig, og teknikker og metoder for å programmere slike systemer er et populært forskningsområde. Vi introduserer den første implementeringen av A*-søkealgoritmen som distribuerer søket over flere GPU-er, og utpeker flere utfordringer som må løses for å oppnå forbedret ytelse.

Implementeringen vår er basert på GA*-algoritmen til Zhou and Zeng og bygger på det asynkrone multi-GPU programmeringsrammeverket til Ben-Nun, Sutton, Pai, and Pingali. For å fordele søkerommet mellom flere GPU-er benytter vi Abstract Zobrist Hashing. Implementeringen er benchmarket ved å løse skyvepuslespill, som er et ofte brukt benchmark for

Multi-GPU-versjonen vår av A* lar oss løse vanskeligere søkeproblemer enn det en enkelt GPU kan håndtere ved å utnytte den økte minnekapasiteten til flere GPU-er. Den oppnår derimot ikke generelt bedre ytelse målt i kjøretid når man øker antallet GPU-er i bruk. Vi foreslår hvordan store deler av kostnadene som blir introdusert når man skalerer horisontalt kan elimineres, og viser at hvis dette kan oppnås kan hver ekstra GPU gi en økning i søkefrekvens på minst 50% av ytelsen til en enkelt GPU.

Preface

I would like to thank Dr. Anne C. Elster for supervising this work, and for providing useful guidance and assistance throughout the entire process.

A big thanks goes to all members of the HPC-Lab directed by Dr. Elster for continued inspiration and collaboration.

This work was done in collaboration with Dr. Gavin Taylor of the US Naval Academy. I would like to thank him for his assistance in running benchmarks on the Yoda supercomputer cluster, and for useful guidance and advice during his sabbatical at the HPC-Lab in 2017 and 2018.

Finally, I would like to thank NTNU for supporting the HPC-Lab with high-end GPUs, and IBM for their support of the HPC-Lab, including the loan of the IBM Minsky system used extensively in this work.

Contents

List of Tables

List of Figures

List of Algorithms

List of Listings

List of Abbreviations

A* “A-star”, an extension of Dijkstra’s graph traversal algorithm

AHDA* Hash Distributed A*. (With state space abstraction [3])

AI Artificial Intelligence

APRA* Parallel Retracting A*. (With state space abstraction [3])

AZH Abstract Zobrist Hashing

BPIDA* Block-Parallel Iterative Deepening A*

BSP Bulk Synchronous Parallel

CO Communication Overhead

CPU Central Processing Unit

CTA Cooperative Thread Array

DWL distributed worklist

ECC Error Correcting Code

gcc GNU Compiler Collection

GPGPU General-Purpose computing on Graphics Processing Units

GPU Graphics Processing Unit

HDA* Hash Distributed A*

IDA* Iterative Deepening A*

PCIe PCI Express. (Peripheral Component Interconnect Express)

PE Processing Element. (CUDA processing core)

PRA*	Parallel Retracting A*
SIMT	single instruction, multiple thread
SM	Streaming Multiprocessor
SO	Search Overhead
TO	Termination Overhead
XOR	exclusive or

Chapter 1

Introduction

The persistent demand for increased compute power in the high performance computing community has lead to the prevalence of compute nodes equipped with multiple Graphics Processing Units (GPUs) [4]. GPUs provide high memory bandwidth and parallelism, and are attractive targets for accelerating resource-hungry algorithms. However, these benefits do not come for free. In order to achieve high memory bandwidth and parallelism, GPUs use a single instruction, multiple thread (SIMT) execution model, and requires threads of execution to cooperate in order to fully benefit from the memory and parallelism capabilities of the hardware. Though this means performing irregular and sparse computation on GPUs becomes difficult, it is still possible to achieve substantial speed-ups over CPU-based alternatives.

The proliferation of multi-GPU compute platforms means research and development has been dedicated to the programming of such systems. Utilising resources efficiently on a GPU is challenging, and adding additional GPUs to the mix does not make it any easier. These difficulties are further amplified with irregular algorithms, of which A* is an example.

Groute [1] is a programming framework which aims to simplify and quicken the development of asynchronous, irregular applications for multi-GPU platforms. It implements low-level communication constructs for inter-GPU communication, achieving good performance using techniques like pipelining and packetisation. It also provides higher-level constructs like distributed worklists, easing development of new applications that need them.

A* is an informed search algorithm with applications in planning, scheduling, path-finding, protein design and other biological sequence alignment problems, natural language processing, and puzzle-solving [5]–[11]. Many of these problems are very demanding, and much work has been devoted to improving the performance of the original A* algorithm of Hart, Nilsson, and Raphael [12]. Recent research has been dedicated to bringing A* search to GPUs, but inherent sequentiality in both the algorithm and the involved data structures make this a non-trivial effort [2], [9], [13], [14]. Other authors have looked into distributing A* search across multiple threads, processors, and even compute nodes [6], [15], [16], working on mitigating issues such as load balance and communication overhead. Abstract Zobrist Hashing [16] is a recent technique which combines state space abstraction [3] with the hashing method of Zobrist [17] to achieve a compromise between communication overhead and load balancing.

1.1 Goals and contributions

Based on these previous efforts in parallelising A* and multi-GPU computing, we implement the first A* search algorithm distributed across multiple GPUs. The goal of our research is to provide an initial leap into the multi-GPU execution of A* search and show that achieving good performance by utilising multiple GPU accelerators is possible. We aim to provide a stepping stone for further research and development, identifying the challenges and obstacles that arise when performing A* search on multi-GPU platforms, and proposing approaches to overcome these hurdles.

The problem we focus on solving with the A* algorithm is the sliding tile puzzle, which has a long history as one the main benchmark problems for A* algorithm variations. Specific to the sliding tile puzzle, we utilise additive pattern databases as the heuristic.

1.2 Outline

The remainder of this report is structured as follows.

- In Chapter 2, we present background material relating to GPU programming, A* search, and sliding tile puzzles.
- In Chapter 3, we present specifics concerning our multi-GPU implementation of the A* algorithm.
- In Chapter 4, we present and discuss benchmark results.
- In Chapter 5, we present conclusions and suggestions for future work.
- Finally, in Appendix A, we list some excerpts from our implementation's source code.

Chapter 2

Background

This chapter will provide an overview of the main background material on which this work is built, giving an overview of GPU and multi-GPU computing, the A* algorithm and efforts on parallelising it, and finally an outline of the sliding tile puzzle problem used to benchmark A*.

2.1 The Graphics Processing Unit[1]

Traditionally, Graphics Processing Units (GPUs) were exactly that — *graphics* processing units. The massive parallelism and memory bandwidth they provide has incentivized researchers and hardware manufacturers to investigate and facilitate General-Purpose computing on Graphics Processing Units (GPGPU) [18]. GPGPU refers to the practice of solving a wide spectrum of computational problems, not just graphics problems, on GPUs.

The high parallelism and memory bandwidth of GPUs stem from their original use as graphics accelerators. Rendering graphics is a highly parallel problem, where typically each pixel of a scene can be rendered independently of other pixels. Although early GPUs were specialized for graphics rendering, the Nvidia GeForce 8800, introduced in 2006, had the first unified graphics and general purpose computing GPU architecture, and could be programmed programmed using C and CUDA [19]. Subsequent GPUs have increasingly improved support for general-purpose computing, including supporting IEEE 754 floating-point operations, ECC memory protection, cached memory, and most recently, tensor cores [20], [21].

[1]Adapted from author's specialization project from fall 2017 at NTNU.

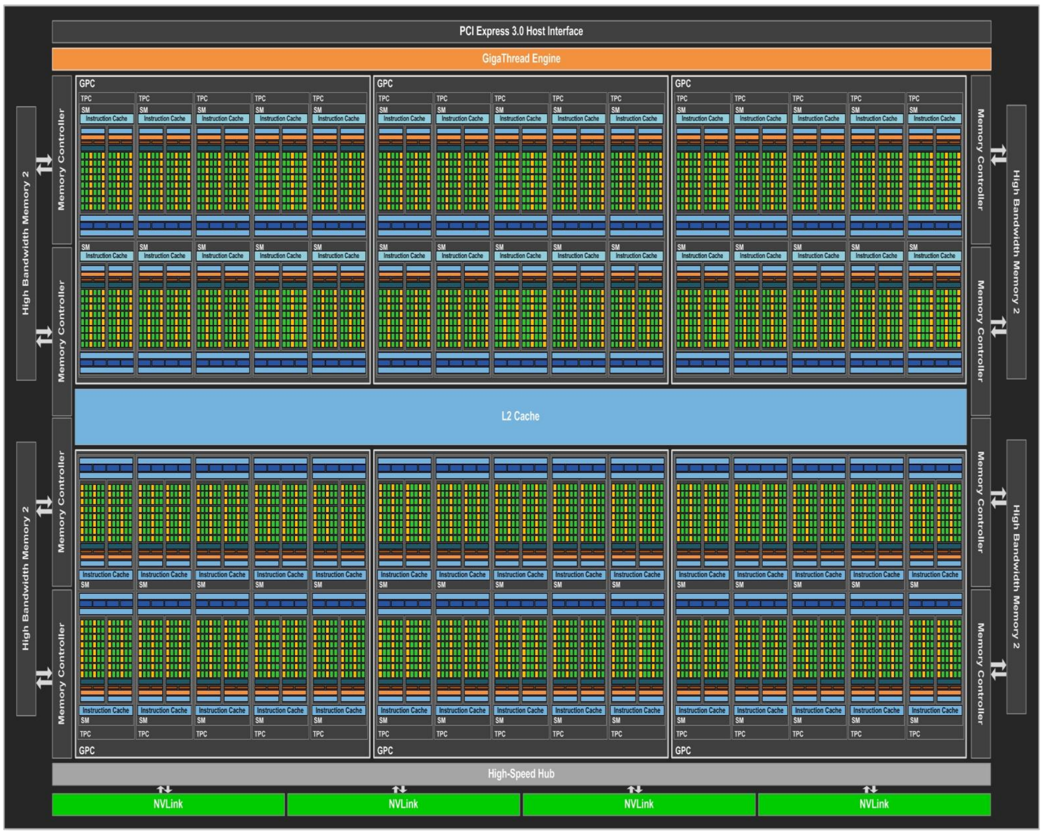


Figure 2.1: Overview of the Nvidia Pascal GP100 GPU with 60 SM units. From Nvidia Corporation [21, Figure 7]. Reprinted with permission.

2.1.1 The Nvidia GPU architecture

Nvidia Corporation is one of the largest GPU designers in the world. In 2008, they introduced the first unified architecture GPU in the GeForce 8800. They have since introduced the Tesla line of products, which targets GPGPU specifically.

Pascal is Nvidia's second most recent microarchitecture, released in 2016, and succeeded by Volta in 2017 [22]. The highest-end Pascal GPU, the GP100, has 60 Streaming Multiprocessor (SM) units, each with 64 Processing Elements (PEs), also known as CUDA processing cores, for a total of 3840 PEs [21]. An overview of the GP100 is shown in Figure 2.1.

The consumer-grade Nvidia TITAN Black GPU is based on the older Kepler microarchitecture from 2012 [23]. It has a total of 2880 PEs [24].

Thread execution

A Pascal multiprocessor can simultaneously schedule and execute 2048 concurrent threads [21]. In order to enable the management and execution of this many threads, they are executed in a SIMT manner. This means that threads are grouped together such that conceptually, each thread in the group either has to execute the same instruction, or some threads must wait, until the branches re-converge. This makes avoiding *branch divergence* important for optimizing performance. Groups of threads executed in this manner are called *warps* and consist of up to 32 threads [20], [21].

In addition to the SIMT execution, memory accesses are also optimized for the warp model [20], [25]. When the threads in a warp access different memory addresses, the memory system coalesces them into a minimal amount of accesses. This means that if the 32 threads in a warp access 32 different 4-byte words aligned in a single 128-byte segment, every memory access can be satisfied by a single physical memory access. However, if the memory accesses are spread out or unaligned, at worst 32 separate serial accesses must be performed. It follows that similarly to branch divergence, avoiding uncoalesced memory accesses is very important to attain good performance.

NVLink

NVLink is Nvidia's GPU and CPU interconnect technology which allows GPUs to access other GPUs' memory and host memory at up to 160 GB/s (Pascal [21]) or 300 GB/s (Volta [22]). NVLink enable GPUs to communicate among themselves and with NVLink-enabled host systems at much higher rates than the conventional PCI Express (PCIe) bus technology. Individual NVLink interconnects can be bonded together in *gangs*, aggregating their bandwidth.

2.1.2 The CUDA programming model

The CUDA programming framework, released by Nvidia in 2007, enables general-purpose programming of CUDA-enabled GPUs [19]. CUDA is an extension to the C and C++ programming languages that provides three key abstractions: hierarchical groups of threads, shared memories, and barrier synchronization.

A CUDA program is written very similarly to a regular C program, except it has a number of *parallel kernels* that are applied to the GPU as threads and executed in parallel. The serial part of a CUDA program runs on a host CPU. Kernels are functions that look like regular, serial code, but are specially

Listing 2.1: Example CUDA code for element-wise squaring an array of integers on a GPU. Some code components omitted for brevity.

```
 1 __global__ void square(int *a, int *b) {
 2     int i = blockIdx.x;
 3     b[i] = a[i]*a[i];
 4 }
 5
 6 int hostA[32];
 7
 8 int main() {
 9     cudaMalloc((void **)&gpuA, 32*sizeof(int));
10     cudaMalloc((void **)&gpuB, 32*sizeof(int));
11
12     cudaMemcpy(gpuA, hostA, 32*sizeof(int),
           cudaMemcpyHostToDevice);
13
14     square<<<1, 32>>>(gpuA, gpuB);
15 }
```

decorated at call sites to specify that they should be executed in parallel. The same code is executed by as many threads as specified when the kernel is called. Listing 2.1 shows an extract of a simple CUDA program, with a kernel definition on lines 1–4 and a kernel invocation launching a single block of 32 threads on line 14. The `__global__` annotation identifies a kernel that is called from the CPU and executed on the GPU.

Threads in CUDA are organized as grids of blocks[2] of threads [26], visualized in Figure 2.2. Blocks are sets of threads that can access shared memory and can cooperate via barrier synchronization. Grids are sets of blocks that may execute independently, or coordinate via global memory [20]. When threads are executed, they have access to their thread identifier, unique within a block, the block identifier, unique within a grid, and a grid identifier, unique across the device. The block and grid identifiers may be one, two or three-dimensional [19].

Note that the blocks and grids are an abstraction, and do not map directly to concepts like warps and multiprocessors. It is, however, crucial for the programmer to know how they both work and relate to each other. Threads within a block are executed as one or more warps, and making block sizes multiples of the warp size can be beneficial to performance. The strength of the grid-of-blocks abstraction is that a CUDA program can execute on a GPU of any size, exploiting whatever amount of parallelism is available.

[2]Also called Cooperative Thread Arrays (CTAs).

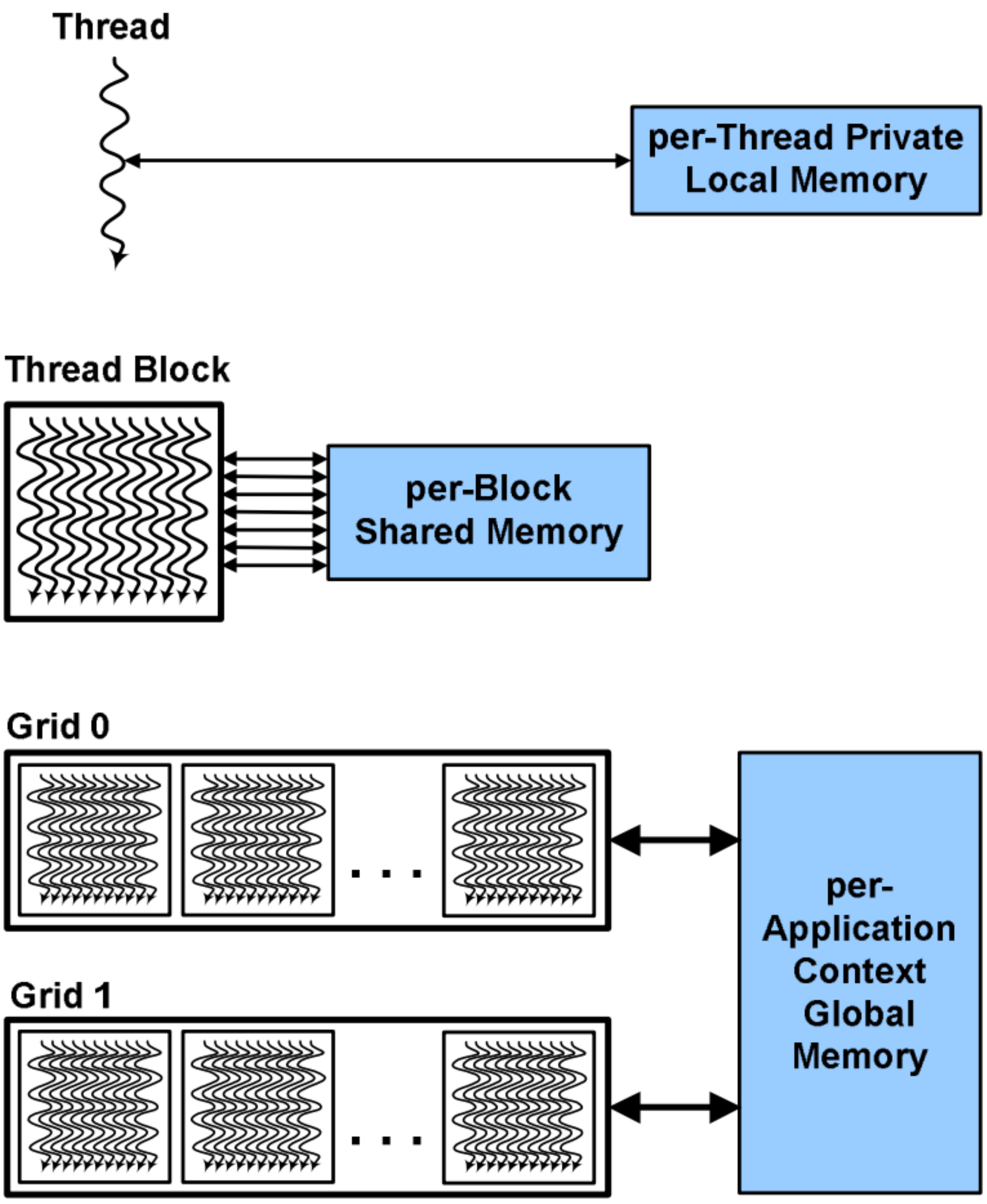

Thread
per-Thread Private Local Memory
Thread Block
per-Block Shared Memory
Grid 0
Grid 1
per-Application Context Global Memory

Figure 2.2: Overview of the CUDA thread model with local and shared memories. From Nvidia Corporation [27, page 6]. Reprinted with permission.

2.1.3 Multi-GPU programming

With Multi-GPU programming, we refer to the programming of a single compute node with multiple GPU devices (as opposed to a cluster of nodes equipped with GPUs). A single compute node can typically handle up to about 25 GPUs, and a challenge arises when an application requires communication between these devices [1]. As noted by Ben-Nun, Sutton, Pai, and Pingali [1], there are two traditional approaches to multi-GPU programming: as multiple processes, each controlling a single GPU, or as a single process using a Bulk Synchronous Parallel (BSP) programming model. Both of these techniques introduce overhead and potential underutilisation of resources, the former due to the use of message-passing, and the latter due to synchronisation. These difficulties may be amplified when working with irregular applications with unpredictable work and communication patterns.

Ideally, we would like to develop multi-GPU applications in an asynchronous manner, such that processors and GPUs can work independently and without waiting for other devices. However, programming such asynchronous applications from scratch turns out to be challenging. The heterogeneity of multi-GPU hardware topology, where pairs of GPUs may have different interconnects and communication speeds, only adds to the difficulty of asynchronous programming.

Groute

With these challenges in mind, Ben-Nun, Sutton, Pai, and Pingali [1] introduced Groute, a programming model and runtime environment that enables easier development of asynchronous multi-GPU applications. It takes concepts from computer networking and applies them to multi-GPU systems, introducing abstractions such as Endpoints (hosts and GPUs), Links, and Routers. The Groute framework handles all communication between Endpoints in an asynchronous manner, and low-level networking concepts such as packetisation and pipelining ensure responsiveness and improve computation and communication overlap. Abstract Routers enable effortless programming of communication patterns such as broadcasting to multiple devices, sending to the first available device, or gathering data from multiple devices.

Groute also provides high-level reusable constructs such as a distributed worklist (DWL) implementation. A DWL is essentially a global list of work-items to perform some computation on. Each peer that participates in the DWL is typically both a consumer and producer; a work-item may be extracted from the list, and its processing may produce further work-items, which are pushed back onto the list. Often, in distributed computation, work-items can only be handled by specific devices, like when the search space of a search algorithm is partitioned between devices. This requires all devices to be able to communicate with all others, which the Groute DWL solves by using a ring topology. The DWL implementation takes care of all communication and coordination, leaving it to the programmer only to provide certain callbacks for directing work-items (process locally, send away or drop) and for the actual processing of work-items.

2.2 The A* search algorithm

The A* algorithm is a *best-first* (or *informed*) search algorithm [12]. It uses a heuristic function to estimate the cost of a solution when deciding which nodes to expand during search. This enables A* to explore less of the search space to find a solution, compared to uninformed algorithms like breadth-first or depth-first search [28].

In order to minimise the number of expanded nodes during search, A* evaluates an estimator $\hat{f}(n)$ for each node n, and selects the best candidate for expansion based on the estimator value of each node. The estimated function is usually defined as

$$f(n) = g(n) + h(n)$$

and its estimator as

$$\hat{f}(n) = \hat{g}(n) + \hat{h}(n).$$

The two functions $g(n)$ and $h(n)$ represent the lowest cost to get from the start node to node n, and from node n to the goal node, respectively. In practice, $\hat{g}(n)$ is the cost of the best path to node n found so far, and $\hat{h}(n)$ is the heuristic function, using knowledge of the problem domain to estimate the cost from n to the goal [12].

Algorithm 2.1 lists pseudocode for the A* algorithm. It maintains two sets of nodes, usually called the *open list* and *closed list*, in which nodes to be expanded and nodes that have been expanded are kept, respectively. The open list is actually a priority queue, usually implemented as a heap, while the closed list is an associative array mapping states to search nodes, typically implemented as a hash table. Expanding a node here means iterating over its neighbour nodes, deciding whether to add them to the open list, only update their $\hat{g}$-values, or skip them entirely.

Algorithm 2.1. A* best-first search

```
procedure A*($start, goals$)
    OPEN $\leftarrow \{start\}$, CLOSED $\leftarrow \varnothing$
    while OPEN $\neq \varnothing$ do
        $node \leftarrow$ EXTRACT(OPEN)                ▷ Extract node with lowest $\hat{f}$
        if $node \in goals$ then
            return path from $start$ to $node$
        end if
        CLOSED $\leftarrow$ CLOSED $\cup \{node\}$
        for all $node' \in$ NEIGHBOURS($node$) do
            $g' \leftarrow \hat{g}[node] +$ COST($node, node'$)        ▷ Add edge cost
            if $node' \in$ CLOSED then
                if $g' < \hat{g}[node']$ then                ▷ Reopen if better
                    OPEN $\leftarrow$ OPEN $\cup \{node'\}$
                else
                    continue
                end if
            else if $node' \notin$ OPEN then
                OPEN $\leftarrow$ OPEN $\cup \{node'\}$
            else if $g' \geq \hat{g}[node']$ then          ▷ Avoid updating if worse
                continue
            end if
            $\hat{g}[node'] \leftarrow g'$
            $\hat{f}[node'] \leftarrow g' +$ HEURISTIC($node'$)
        end for
    end while
    return failure        ▷ No path from $start$ to any node in $goals$ exists
end procedure
```

2.2.1 Admissibility and optimality of A*

An algorithm is said to be *admissible* if it is guaranteed to find an optimal solution if one exists. The admissibility of A* is contingent on the heuristic function also being admissible.

The heuristic function is admissible iff for all n, $\hat{h}(n) \leq h(n)$ [12], [28]. In other words, it must be optimistic, and never overestimate the actual cost of a solution path from a given node. Intuitively, if the heuristic function is overestimating, an optimal solution path could end up unexplored while the search finishes with a suboptimal solution.

It is also preferable that the heuristic be consistent [12] (or *monotonic* [28]). That is, the heuristic must adhere to the triangle equality, such that the $\hat{f}$-value of nodes does not decrease along a path from the start node. If a consistent heuristic is used, every node is guaranteed to only be expanded once, and the algorithm implementation can be slightly simplified, replacing Lines 12 to 16 in Algorithm 2.1 with a single **continue** statement [10].

Given an admissible heuristic, A* search is also *optimally efficient* [12], [28]. That is, no other admissible algorithm using the same heuristic is guaranteed to expand fewer nodes than A* search.

Detailed proofs of the admissibility and optimality of A* appear in Hart, Nilsson, and Raphael [12] and Russell and Norvig [28].

2.2.2 The heuristic function

Given the formal requirements imposed on the heuristic function discussed in Section 2.2.1, the question then arises: how do we design a good heuristic function?

The most trivial heuristic, yielding $\hat{h}(n) = 0$ for all n, reduces A* search to an uninformed pseudo-breadth-first search, where nodes are ordered by $\hat{g}(n)$ and the shortest paths are expanded first, similar to how breadth-first search expands nodes by depth level.

In order to expand the least amount of nodes before finding a solution, we must aim for our heuristic function $\hat{h}(n)$ to be as close to $h(n)$ as possible while staying admissible and consistent [12].

2.3 A* on the GPU

A* is not an algorithm that lends itself easily to parallelisation. It is inherently serial, each iteration extracting and expanding a single node from the open list. This is a requirement for A* to be admissible; we have to process nodes in sequential order by their $\hat{f}$-values.

Zhou and Zeng [2] proposed one approach to parallelising A* on GPUs. The first step toward parallelisation is to calculate the heuristic function in parallel; this is generally trivial due to the inherent independence between heuristic calculations. Next, we would like to process multiple nodes simultaneously to exploit the parallelism of the GPU. Here we run into a challenge: the open list is usually implemented as a binary heap, and adding or removing items are sequential operations taking $\mathcal{O}(\log n)$ time. Although concurrent, lock-free priority queues exist, they are not suitable for use on GPUs using SIMT execution.

The solution of Zhou and Zeng [2] is to replace the single open list with one open list per CUDA thread in the GPU implementation, each containing a different, non-overlapping subset of the full open list. This is not a new idea: Evett, Hendler, Mahanti, and Nau [15] introduced this concept with their Parallel Retracting A* algorithm.

Zhou and Zeng call their parallel A* search algorithm GA*, and it differs from the serial A* in Algorithm 2.1 in the following ways:

- On each iteration, each CUDA thread tries to extract a node from its private queue. When a solution is detected, instead of being immediately returned, it is atomically compared against any previously or simultaneously discovered solutions, and stored in a shared variable if the new solution has a lower cost.

- The best found solution is compared against the lowest $\hat{f}$-value of any open list, and if no open node has a lower value, the found solution is known to be optimal and the algorithm terminates.

- If no optimal solution can be guaranteed, non-solution nodes are expanded, and newly created nodes added to a shared list. Instead of adding new nodes directly to the open list, duplicates are removed once all threads have expanded their nodes.

- After node deduplication, heuristics for the remaining nodes are calculated, and the nodes distributed to the various open lists. It is important to note that they do not push expanded nodes to the same open list as their parents; this ensures good nodes are distributed among multiple threads and increases the amount of useful work being done.

One major detail still missing is how to perform node deduplication on GPUs. Zhou and Zeng [2] solve this with a hash table scheme they call Parallel Hashing with Replacement [29, Algorithm 3], which is a simplification of Cuckoo Hashing [30]. Similar to Cuckoo Hashing, it may use multiple hash functions, and primarily differs in the way conflicts are handled. If an item is being inserted and maps to an occupied slot, the existing item is dropped. This greatly simplifies implementation on the GPU. The effect of using this technique for detecting duplicate nodes in A* search is that we may miss some duplicates and expand some nodes more than once. This does not affect the admissibility of A*, and so it is safe to use.

2.3.1 Related work

Other authors have worked on A* search problems using the GPU.

Inam [13] applied A* to grid path finding using a simple Manhattan distance heuristic, showing what is possibly the first implementation of A* on the GPU. They begin with a basic, sequential implementation of A*, and enhance it to improve performance on GPUs. One of these is to parallelise the expansion of nodes, allowing 8 CUDA threads to work on a single node (because each node in the grid has potentially 8 valid successor nodes). Only a single node is extracted from the open list in each iteration, so the parallelism is limited by the out-degree of search nodes.

Hiroki, Naoaki, and Hirokazu [9] presented an iterative version of Inam [13], focusing on permutation puzzle solving on the GPU, in particular the solving of Rubik's cube. They implement a slightly specialized version of Iterative Deepening A* (IDA*) for solving these puzzles, using pattern databases and domain knowledge to solve cube puzzles.

Horie and Fukunaga [14] introduced Block-Parallel Iterative Deepening A* (BPIDA*), which assigns search subtrees to CUDA blocks, aiming to reduce the warp divergence and load imbalance which occurs when assigning subtrees to individual threads during IDA* search.

2.4 Distributed A* search

In today's compute landscape, we must often look to scale horizontally. A single CPU or GPU can only yield a certain amount of processing power and memory capability before improving performance becomes prohibitively expensive. Thus, it is interesting to explore algorithms and techniques for distributing computation across multiple devices. This is particularly interesting for the A* algorithm, which is often limited by its high memory usage.

Parallel Retracting A* (PRA*) [15] its derivative Hash Distributed A* (HDA*) [6] are two closely related techniques that use a hash function of search nodes' state and a modulus operator to divide the search space between different threads or computation devices. When a thread expands a node, it calculates the hash value of its state and determines whether it should be processed locally or sent to a different thread. HDA* is an improvement over PRA* on the communication part of this scheme.

Clearly, the choice of hash function affects the performance of PRA* and HDA*. If a uniform hash function is used, for N threads or devices, on average only $\frac{1}{N}$ of nodes generated will be processed locally, while $\frac{N-1}{N}$ will be transferred out. If we can increase the fraction of search nodes that are processed locally, we can reduce the Communication Overhead (CO) of these algorithms. Jinnai and Fukunaga [16] define the CO as the fraction

$$\frac{\text{nodes generated}}{\text{nodes sent away}}.$$

Burns, Lemons, Ruml, and Zhou [3] introduced APRA* and AHDA* as variants of these algorithms that use state space abstraction instead of simple hashing to divide the search space among threads. They divide the search space into blocks, and assign a subset of the blocks to each processing thread. Generated nodes then have a high probability of falling in the same block as their parent nodes, and being assigned to the same thread.

While APRA* and AHDA* mitigate the issue of communication overhead, they introduce a new problem: load balancing. Because of the way the search space is distributed, some computation threads may spend time searching in "useless" partitions of the search space, or perhaps not searching at all due to no nodes being assigned to them. Jinnai and Fukunaga [16] define Search Overhead (SO) as the fraction

$$\frac{\text{nodes expanded in parallel search}}{\text{nodes expanded in sequential search}}.$$

If the parallel A* implementation does no useless work and expands the same amount of nodes as a sequential implementation, the SO will be close to or even equal to 1. However, if a single processing thread becomes a bottleneck, other threads will spend time expanding worse nodes, resulting in a higher SO.

Ideally, we would like the best of both worlds: the load balancing of uniform hashing, and the communication efficiency of state space abstraction.

2.4.1 Abstract Zobrist Hashing

Abstract Zobrist Hashing (AZH) (Jinnai and Fukunaga [16]) combines Zobrist hashing, a hashing method introduced by Zobrist [17] intended for game playing, and the abstraction of Burns, Lemons, Ruml, and Zhou [3] to create a work distribution technique that has both low communication overhead and low search overhead.

Zobrist hashing

Zobrist hashing [17] is a hashing method suitable for hashing game states for board games such as checkers, chess, Go, and the sliding tile puzzle. It satisfies two desired properties of hash functions: a good random distribution, and fast computation. It works by generating a table of random integers in advance, denoted S, where each integer in S represents a possible "feature" of the data being hashed. In the domain of game playing, such features may "knight on f6" (for chess) or "tile 11 in position 4" (for the 15-puzzle). The Zobrist hash $Z(x)$ of a certain input set of such features $x = \{x_1, x_2, \ldots, x_n\}$ then becomes the exclusive or (XOR) sum of all its features:

$$Z(x) = \bigoplus_{i=0}^{n} S[x_i].$$

Because the XOR of uniformly distributed random integers is also uniformly and randomly distributed, so is the Zobrist hash.

Due to some other useful properties of XOR, viz. commutativity ($x \oplus y = y \oplus x$), associativity ($x \oplus [y \oplus z] = [x \oplus y] \oplus z$), self-inversing ($x \oplus x = 0$) and the identity ($x \oplus 0 = x$), incrementally generating hashes for game states becomes very simple. Suppose we want to calculate the hash of a new state in a game of chess, where we moved a knight from position f6 to position d5. Due to the self-inversing and identity property, we can simply XOR in the features "knight on f6" and "knight on d5" to the parent state's hash code to get the hash of the new state. The commutativity and associativity ensures that if we reverse the move, we go back to the parent state's hash code.

Zobrist hashing thus gives us an efficient method of hashing search space states and their successor nodes, particularly for games and puzzles.

Applying state space abstraction

Jinnai and Fukunaga [16] uses Zobrist hashing in combination with state space abstraction as used by Burns, Lemons, Ruml, and Zhou [3] to form a hybrid technique that is a compromise between the load balancing of uniform hashing and the low communication overhead of abstraction. It introduces no additional overhead during runtime; the only difference from regular Zobrist hashing is the way the pre-generated integer table S is created.

Let's imagine we are hashing states of a game where we have tiles numbered 1 through 8 on a 3 by 3 board with positions numbered 1 through 9 (i.e. the 8-puzzle, discussed further in Section 2.5). The feature set S consists of integers $s_{t,p}$ where t identifies a tile and p its position. If the integers in S are all random, we have regular Zobrist hashing. However, if we group some of these features together, similar board states in our game will hash to the same value. To see this, imagine we partition our 3 by 3 board into rows, thus letting $s_{t,1} = s_{t,2} = s_{t,3}$ for all tiles t, and similar for the second and third rows. Then, moving a tile laterally would result in the child state having the same hash as its parent. For example, if we move a tile t from position 2 to position 1, then

$$Z(child) = Z(parent) \oplus t_{t,2} \oplus t_{t,1} = Z(parent) \oplus 0 = Z(parent).$$

This means similar states will be grouped together, resulting in lower communication overhead if used for state space distribution in A* search. We also benefit from good load balancing, because whenever the hash value does change, it is randomly and uniformly distributed.

2.5 The sliding tile puzzle

The sliding tile puzzle is a puzzle game where $(n \times n - 1)$ tiles and a blank space are to be rearranged on an $n \times n$ board from a scrambled state to a given solution state. The puzzles are often named by their number of tiles; e.g. the 15-puzzle refers to the 4×4 variant, and the 24-puzzle to the 5×5 variant. A visualization of the 15-puzzle is shown in Figure 2.3.

The puzzle is solved by "sliding" orthogonally adjacent tiles into the blank space repeatedly until the tiles have been rearranged to the solution configuration (Figure 2.3b). Solving the puzzle optimally means finding a shortest-possible sequence of moves that achieves this.

The sliding tile puzzle is simple to define, but difficult to solve algorithmically (the 15-puzzle has about 10^{13} possible configurations; the 24-puzzle almost 10^{25} [10], [31]). For this reason, it is often used as a benchmark for search algorithms, including the A* algorithm [10].

2.5.1 Applying A* to the sliding tile puzzle

The sliding tile puzzle maps naturally to a state-space problem as described in Fukunaga, Botea, Jinnai, and Kishimoto [10]. Each unique puzzle configuration is a state in the problem space, and the possible tile moves represent transitions between states. We define the cost of each transition to be 1, such that the total cost of a path through the state space is the number of moves performed.

	12	10	13
15	11	9	14
7	3	6	2
4	8	5	1

(a) Scrambled state

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

(b) Solution state

Figure 2.3: Instances of the 15-puzzle.

Manhattan distance

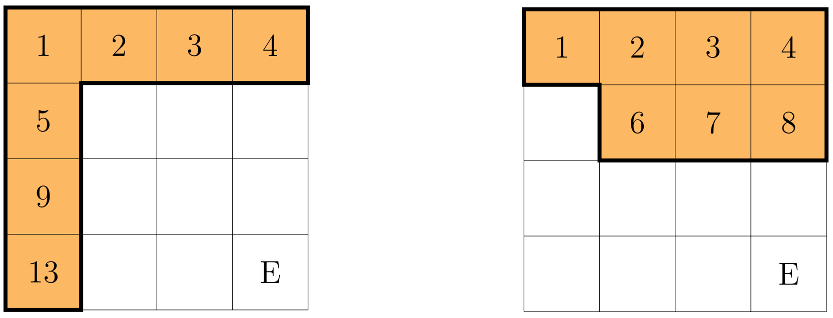
As is customary with A*, the challenge lies in defining the heuristic function $\hat{h}(n)$. One of the simplest and most well-known heuristics for the sliding tile puzzle is the Manhattan distance heuristic [31], [32]. To calculate it, we sum up the Manhattan distance from each tile to its goal position. The Manhattan distance is the sum of a tile's horizontal and vertical displacement from its goal position. This is an admissible heuristic, because in order to reach the goal state, each tile has to move at least this distance, and each move only affects a single tile, making it is a lower bound on the total number of moves required $h(n)$.

The Manhattan distance heuristic is easy to calculate, but not particularly accurate. It does not account for interactions between tiles, and assumes each tile can move "directly" to its goal position. For a more accurate heuristic on the sliding tile puzzle we can use pattern databases.

Non-additive pattern databases

The main weakness of the Manhattan distance heuristic is that it does not take into consideration different tiles' interaction with each other when estimating the number of moves required to get a tile to its goal position. Pattern databases improve upon this by considering groups of tiles together, and how many moves are required to get each of them into their goal position, while disregarding tiles not part of the group. A *pattern* refers to a specific configuration of a group of tiles, and a pattern database records the number of moves required so solve each permutation of a pattern [33]. Generating such a database turns out to be quite simple, and amounts to a breadth-first search backwards from the goal state to each permutation of the pattern.

Figure 2.4 shows two tile patterns used by Culberson and Schaeffer [33]



(a) “Fringe” pattern (b) “Corner” pattern

Figure 2.4: Tile patterns in their goal positions.

for the 15-puzzle. Note that the tiles not part of the patterns (blank) and the empty tile (E) are distinguished. With seven tiles plus the empty tile tracked, the number of moves needed to solve each permutation of the patterns can be stored in 495 MB of memory or less [32].

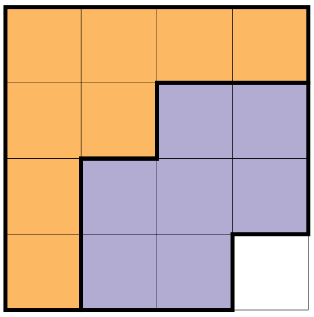
Using pattern databases as a heuristic for A* search is quite simple. For any given puzzle state, we take the pattern of the tiles in the tile group and the empty tile, and look up the number of moves required to solve them in the pattern database. Clearly, this is a lower bound on the amount of moves required to solve the entire puzzle. If we have multiple patterns and databases, we can take the maximum value among each of them to get an even better heuristic.

These types of pattern databases are *non-additive*. That is, we can’t use the sum of two pattern databases as our heuristic, even if the patterns are non-overlapping (as in Figure 2.5), as it would not be admissible. This is because the pattern databases not only count the moves of tiles in the pattern, but the other tiles as well.

It turns out, however, that adding together multiple databases of a slightly different type, namely additive pattern databases, is possible.

Statically-partitioned additive pattern databases

Statically-partitioned additive pattern databases [31] (originally introduced by Korf and Felner [32] as disjoint pattern databases) are created with disjoint, or non-overlapping patterns, as exemplified in Figure 2.5. In contrast with the non-additive pattern databases discussed above, additive databases only count the moves of tiles that are part of the pattern. In essence, they treat all non-pattern tiles plus the empty tile as if they are all empty. Because of this, we are able to add the values for each database together, forming an even better heuristic than non-additive pattern databases [32].


(a) 7-8 partitioning of the 15-puzzle.

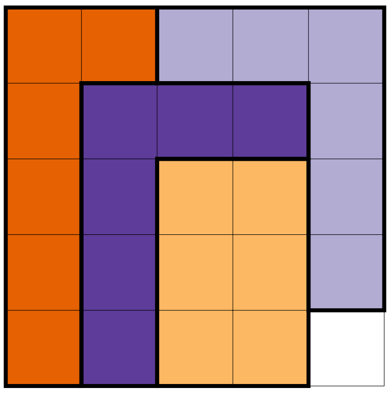

(b) 6-6-6-6 partitioning of the 24-puzzle.

Figure 2.5: Disjoint patterns for the 15- and 24-puzzle.

The Manhattan distance heuristic turns out to be a trivial example of an additive pattern database heuristic where each database pattern contains a single tile [31], [32]. In practice, we use patterns like those in Figure 2.5, and are mainly limited by the memory the databases occupy. Because we disregard non-pattern tiles, the 7-tile pattern database of Figure 2.5a will contain $16!/(16-7)! = 57657600$ entries, and the 8-tile database $16!/(16-8)! = 518918400$ entries. If each entry occupies a single byte, this amounts to about 577 MB of memory. The 6-tile databases in Figure 2.5b similarly require a total of 510 MB of storage.

Felner, Korf, and Hanan [31] introduced dynamically-partitioned additive pattern databases, which are more memory-efficient than statically-partitioned ones, but yield less effective heuristics. They conclude that the statically-partitioned databases described above are preferable if sufficient memory is available.

Chapter 3

Multi-GPU A*

In this chapter we describe how we develop and implement a multi-GPU A* sliding tile puzzle solver.

We use the GA* algorithm [2] as the base algorithm with statically-partitioned additive pattern databases [32] as the heuristic, Abstract Zobrist Hashing [16] for search space partitioning, and Groute [1] for asynchronous inter-GPU communication and distributed worklist coordination.

3.1 GA* with distributed worklist

Our implementation uses the Groute DWL system as its foundation. Figure 3.1 illustrates the DWL implementation with the main CUDA kernels and data structures present on each GPU device, and shows how work items flow through the system and to and from other GPUs.

Each GPU runs the GA* algorithm with its own open and closed lists. However, instead of being pushed directly back to open lists, search nodes expanded by GA* are fed to the Groute distributed worklist implementation, which decides whether to process the nodes locally or send them to a different device by consulting user-implemented callbacks. The callbacks are essentially parallel CUDA kernels that use the Abstract Zobrist Hashing of nodes as well as the local closed set to determine which action to take. At each iteration of the algorithm, nodes are also consumed from the Groute distributed worklist and distributed among the per-thread open lists.

In Figure 3.1, the local worklist is a circular queue of work items that either originate from a different GPU (via the SplitReceive kernel), or from previous iterations of our algorithm (via the SplitSend kernel). The work items in our implementation are A* search nodes, which contain a compact representation of the puzzle board state, a pointer to the parent node, the $\hat{f}$-

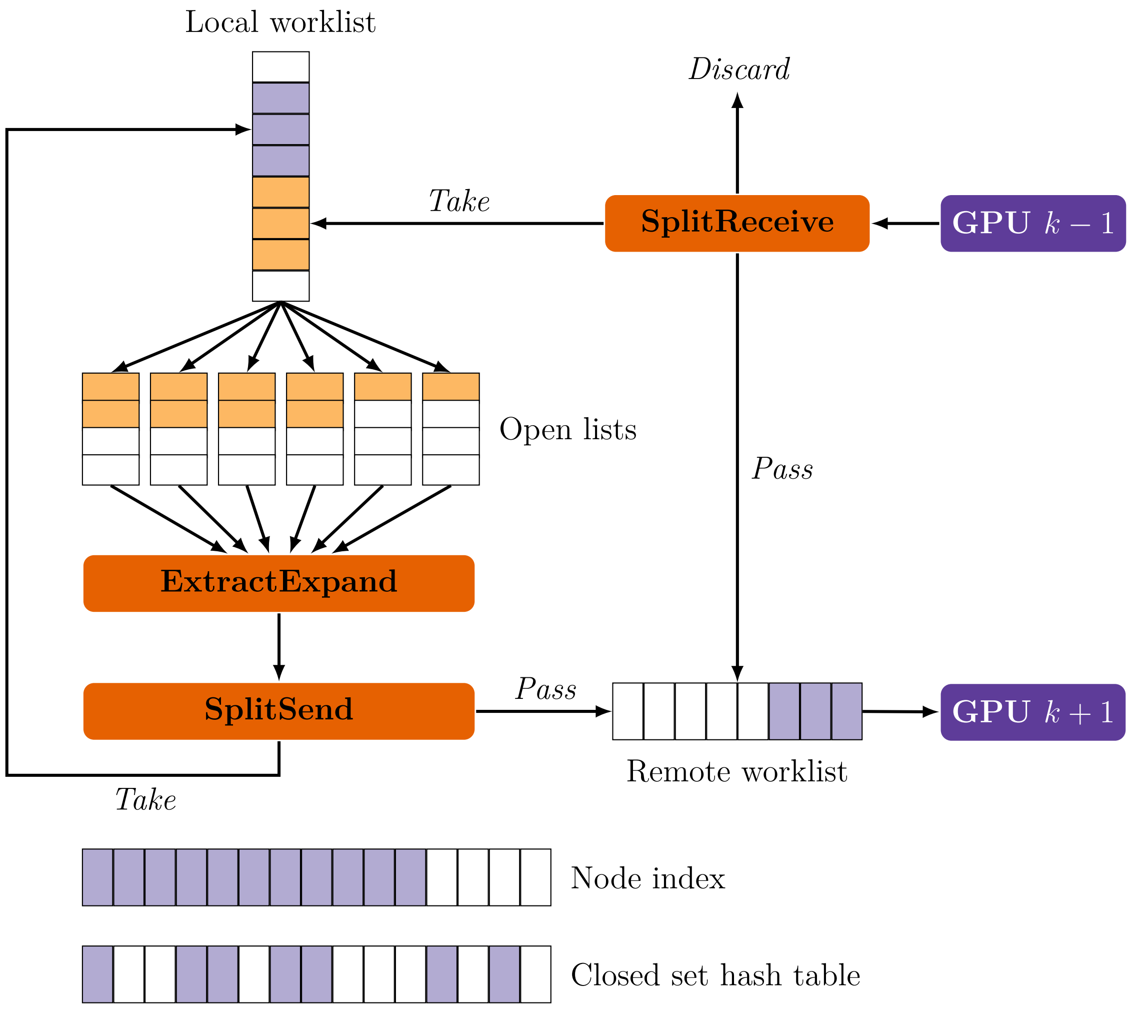


Figure 3.1: Overview of the Groute DWL implementation and per-thread open lists from GA*. Based on Ben-Nun, Sutton, Pai, and Pingali [1, Figure 9].

and $\hat{g}$-values of the node, and its abstract Zobrist hash. Nodes in the local worklist are put into the node index, which keeps all discovered search nodes owned by each device. Pointers into the node index are distributed to and pushed onto the open lists, of which there are one per CUDA thread. The open lists are implemented as simple binary heaps.

The ExtractExpand kernel performs the main A* search work, namely extracting nodes from open lists, expanding and deduplicating them, and passing them on to the SplitSend kernel. We implement the GA* algorithm [2], using statically-partitioned additive pattern databases [32] for our heuristic. Our use of pattern databases is expanded upon in Section 3.1.1.

The SplitSend and SplitReceive kernels are responsible for sorting through work items, deciding whether to keep them in the current device *(Take)*, pass them on to the next device *(Pass)*, or to filter them out *(Discard)*. They use callbacks to decide what to do with search nodes. The callback implementations are quite simple, and their CUDA code is listed in Listings A.1 and A.2. The first step is to determine which device owns the incoming work item. This is deduced with a modulo operation on the item's AZH (elaborated on in Section 3.1.2). Any items owned by a different device has to be passed along; because the local device has no knowledge of closed nodes in other partitions of the search space, no filtering can be done.

If a work item is owned by the current device, we check if it exists in the closed set. For this, we use a hash table that maps board values to pointers into the node index. Note that the hash used for the closed set is *not* the same hash used for determining node ownership (AZH). The AZH's use of abstraction means many nodes that are close in the search space will have the same hash value. This is unfitting for a hash table, so for the closed set we use the compact representation of the puzzle board as the hash value instead. This is the same method used by Zhou and Zeng [2]. When a node's slot in the hash table is occupied, we compare its board with that of the closed node, to avoid false positives.

If a node owned by the current device does exist in the closed set, it is simply discarded. If not, it is added to the local worklist.

When a solution node is discovered, its index is atomically stored in a global variable on the GPU that found it. The host checks the contents of these variables between each iteration and distributes its values to all other GPU devices when it is updated. We can not exit immediately when a solution is discovered, because a better solution may exist unexpanded in open lists or communication buffers on any device. However, once a solution has been found, we can begin discarding all nodes with an $\hat{f}$-value higher than that of the solution. This is possible because our heuristic is admissible. Once all nodes have been discarded or processed, we can exit and return the optimal solution.

3.1.1 Pattern database heuristic

When utilising statically-partitioned additive pattern databases, the main question becomes: how do we partition the puzzle boards? The general rule is that we want to group tiles that are near each other in the goal state together, due to the fact that they interact more with each other than other tiles [32]. The tile groupings used by our implementation are illustrated in Figure 2.5, and are the same as used by Zhou and Zeng [2].

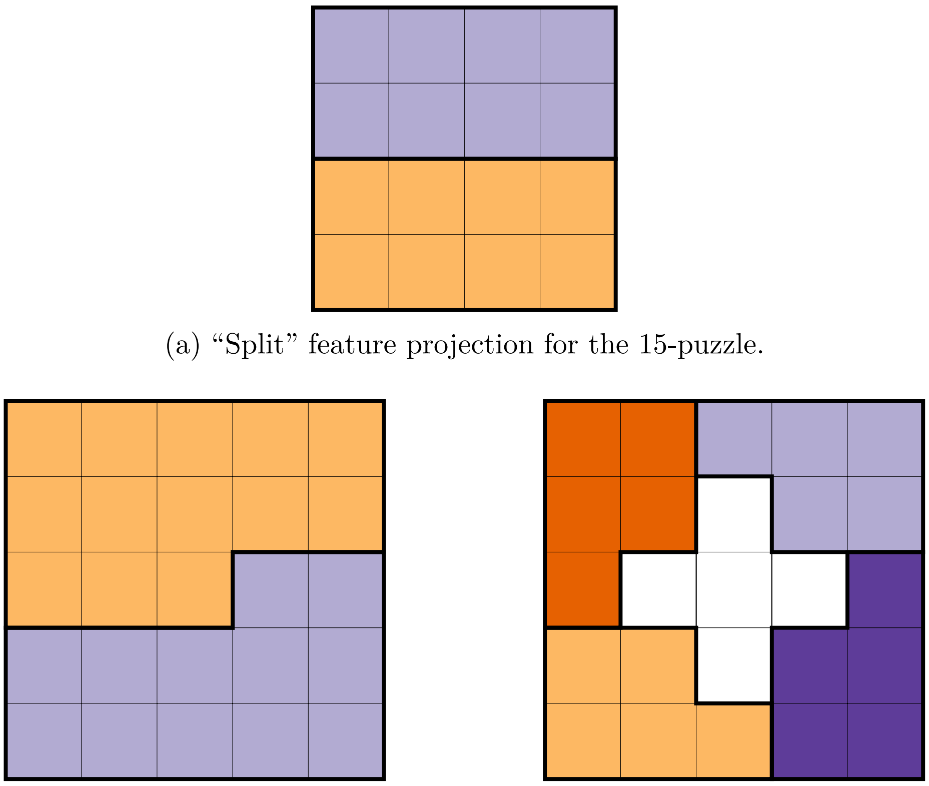
The pattern databases are generated on the host prior to running the main solver program. As the databases are reused for each run, the cost of generation is amortized. When the main solver runs, each database for the particular problem (e.g. 15-puzzle, 24-puzzle) is loaded into shared memory on each GPU device.

3.1.2 Abstract Zobrist Hashing

Similarly to pattern databases, a key question with AZH is how to partition the search space. We achieve this by using feature projections that have a high probability of giving derived states the same hash as their parent. The projections used are the same as those by Jinnai and Fukunaga [16], in addition to a "split" pattern for the 24-puzzle, and are shown in Figure 3.2. They can be interpreted as each tile in the puzzles having only two (Figures 3.2a and 3.2b) or five (Figure 3.2c) possible positions. That is, a single tile will have the same feature value within each shaded area. Thus, a board state will only produce a new hash value if a tile crosses the boundaries of the feature projections, and this gives a good trade-off between search overhead and communication overhead when distributing nodes among GPUs [16].

In order to set up for AZH, we create an $N \times (N+1)$ array of integers S (for the N-puzzle). This corresponds to the set S of features discussed in Section 2.4.1, and gives us a random integer for each tile $1 \leq t \leq N$'s possible position $1 \leq p \leq (N+1)$. The hash of the initial search node is then retrieved as $\bigoplus_{i=1}^{N} S[t_i][p_i]$. Deriving a hash after moving a tile t from position p to p' requires us to XOR two values into the parent hash: $S[t][p]$ and $S[t][p']$. We pre-calculate all these values used for incremental hashing in an $N \times (N+1)^2$ array S', where $S'[t][p][p']$ is the value to XOR when a tile t was moved from p to p'.

On the GPUs, we only need the incremental array S', because we calculate the hash of the initial node on the host. With 32-bit hash values, this array


(a) "Split" feature projection for the 15-puzzle.

(b) "Split" feature projection for the 24-puzzle.

(c) "Block" feature projection for the 24-puzzle.

Figure 3.2: AZH feature projections for the 15- and 24-puzzle.

requires 15.36 kB of memory for the 15-puzzle, and exactly 60 kB for the 24-puzzle. We store S' in CUDA constant memory, which has a limit of 64 kB.

3.1.3 Memory allocation

A* is a memory-hungry algorithm. We would like to make optimal use of the memory available on the GPU devices, so that no buffer or data structure becomes the bottleneck (i.e. becomes prematurely, forcing the algorithm to exit). This is challenging, because dynamically (re)allocating memory and growing data structures is not a trivial task on GPUs [34].

It is difficult to know a priori how much memory is required for each of them, because it depends on problem size, hardware topology, and other factors. We are faced with the task of pre-allocating all memory our algorithm will use throughout its lifetime. The principal structures are the node index, the open lists, the closed set, and Groute buffers for the local, incoming, outgoing and pass-through worklists.

For these reasons, we define the sizes of the principal data structures and buffers as fractions of total free memory on the GPU (after allocating fixed-sized memory like pattern databases), and allow them to be tweaked as part of the program input.

3.2 Benchmarking

In order to to evaluate our implementation and identify how parameters such as hardware, characteristics of the problem being solved, and variations in the algorithm affect its performance, we run several benchmarks on two different hardware platforms.

We benchmark our implementation by solving ten different sliding tile puzzle board configurations, five being 15-puzzles and five 24-puzzles. The puzzles in question are listed in Table 3.1. They were all generated by randomly shuffling the puzzle and hand-picking ten boards that were of an appropriate difficulty; that is, they are not all too easy to solve, and the implementation was able to successfully find optimal solutions in at least a few cases. For the 24-puzzles, we benchmarked using both of the AZH feature projections in Figures 3.2b and 3.2c, while for 15-puzzles we used only the split projection in Figure 3.2a.

When measuring the runtime of our program, we do not include the time used for setup. We measure the wall-clock runtime from when the algorithm starts working and expanding search nodes, discounting any time spent allocating memory and initialising data structures such as pattern databases.

3.2.1 Hardware

We benchmarked our implementation on two systems, which we refer to as *Minsky* and *Yoda*.

The Minsky system is an IBM Power System S822LC (code-named "Minsky") [35], which has two 10-core POWER8 processors and four Nvidia Tesla P100 GPUs. The Minsky system has NVLink interconnects between some pairs of GPUs and CPUs, as shown in Figure 3.3. Note the potential bottleneck at the system bus: it is the only way for GPUs on the left to communicate with GPUs on the right. Each P100 has 16 GB of memory. On the Minsky system, our code was built and executed with CUDA 8.0 and gcc 5.4.0.

Yoda[1] is an EXXACT supercomputer cluster with 6-core Intel Xeon E5-

[1]TOP500 entry: `https://www.top500.org/system/178670`

Table 3.1: Benchmarked board configurations with optimal solution lengths and sequential node expansions. The number in board names refer to the number of random moves performed in order to generate them. Tiles are listed from left to right, top to bottom, with 0 representing the empty space.

Name	Moves	Sequential expansions	Configuration
4x4-300	48	129 177	9, 10, 12, 13, 1, 0, 4, 2, 6, 14, 11, 8, 5, 7, 3, 15
4x4-1200	62	67 141 768	4, 3, 10, 1, 12, 7, 11, 0, 9, 14, 6, 5, 2, 8, 15, 13
4x4-1400	60	4 088 545	0, 11, 5, 1, 12, 10, 2, 6, 4, 9, 7, 13, 8, 15, 3, 14
4x4-1600	56	2 586 809	15, 5, 0, 8, 4, 14, 7, 11, 10, 1, 2, 13, 12, 6, 9, 3
4x4-1900	56	5 743 215	11, 9, 4, 8, 15, 5, 13, 10, 0, 2, 12, 7, 14, 3, 6, 1
5x5-100	38	11 447	2, 6, 9, 3, 4, 12, 7, 1 15, 5, 11, 8, 10 13, 19, 16, 17, 14 0, 20, 21, 22, 18, 23, 24
5x5-200	64	22 023 118	7, 1, 13, 19, 0, 2, 6, 14, 8, 9, 4, 5, 17 10, 20, 12, 11, 18, 15, 3, 16, 21, 22, 23, 24
5x5-300	66	154 996 768	11, 2, 7, 13, 9, 1, 4, 8, 20, 6, 12, 23, 3 15, 10, 17, 0, 22, 5, 14, 21, 16, 18, 19, 24
5x5-400	80	239 691 683	1, 12, 9, 19, 20, 13, 5, 10, 4, 23, 11, 7, 0 8, 3, 6, 24, 15, 21, 2, 16, 22, 14, 17, 18
5x5-500	78	247 401 269	7, 16, 2, 9, 8, 22, 0, 12, 21, 10, 13, 1, 5 4, 18, 19, 23, 15, 6, 3, 11, 17, 24, 14, 20

2620 v2 processors and eight Nvidia TITAN Black GPUs per node, communicating over PCIe. Due to some unresolved issues with some nodes, we were only able to benchmark with up to 7 GPUs. Each GPU on this system has 6 GB of memory. On Yoda, we compiled and executed our code with CUDA 7.5 and gcc 4.8.4.

Due to the differences in the hardware capabilities of the two systems, we use different memory allocation schemes when benchmarking. We try to allocate data structures and communication buffers proportional to the utilisation at runtime, such that a single data structure does not fill up long before the others becoming a bottleneck. By trial and error, we ended up with the proportions shown in Figure 3.4. The slices are shaded to distinguish between GA* memory and Groute memory. Note that on Yoda we had to allocate much more memory to Groute communication buffers, due to the slower interconnects and increased communication.

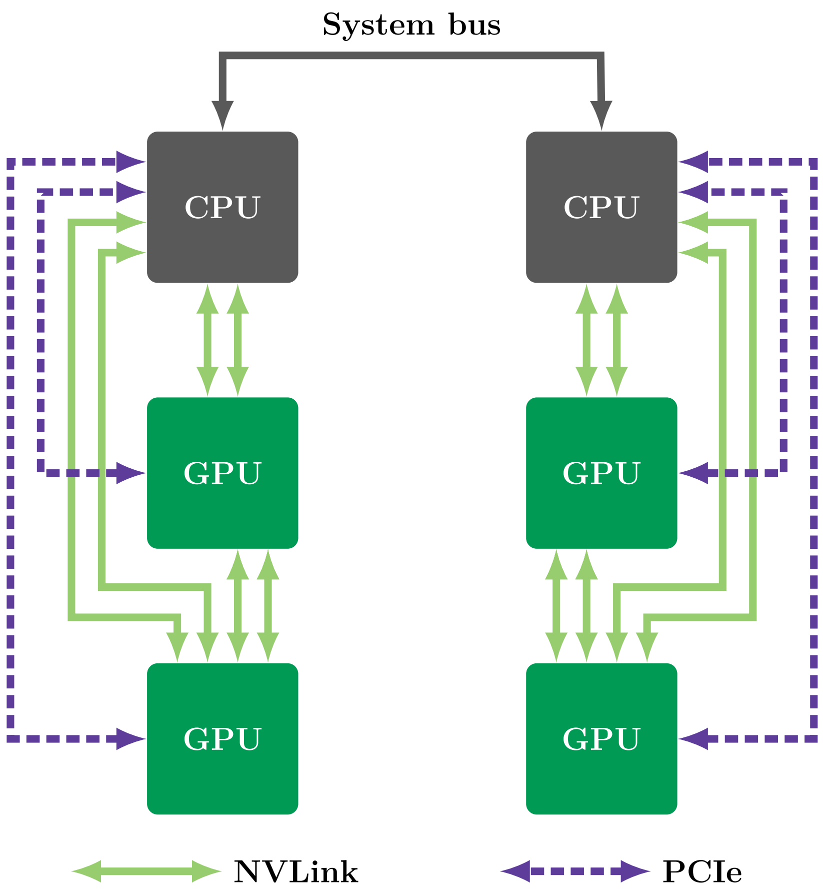
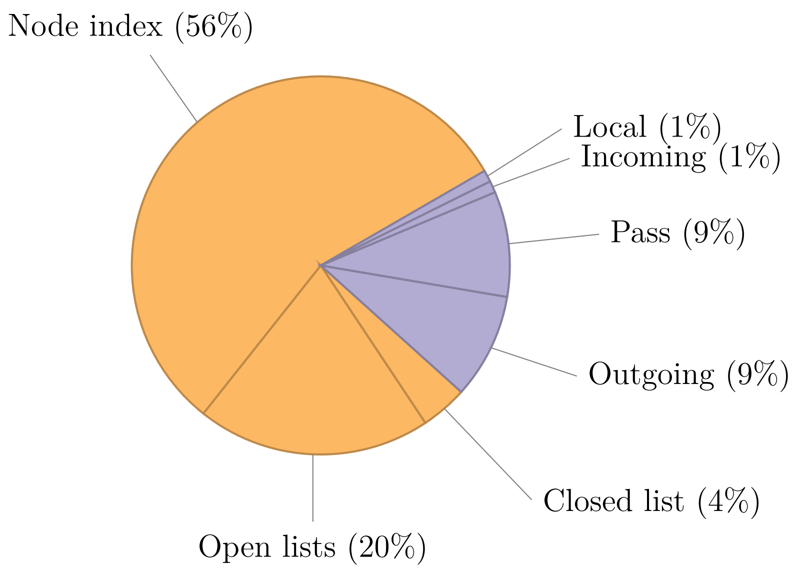
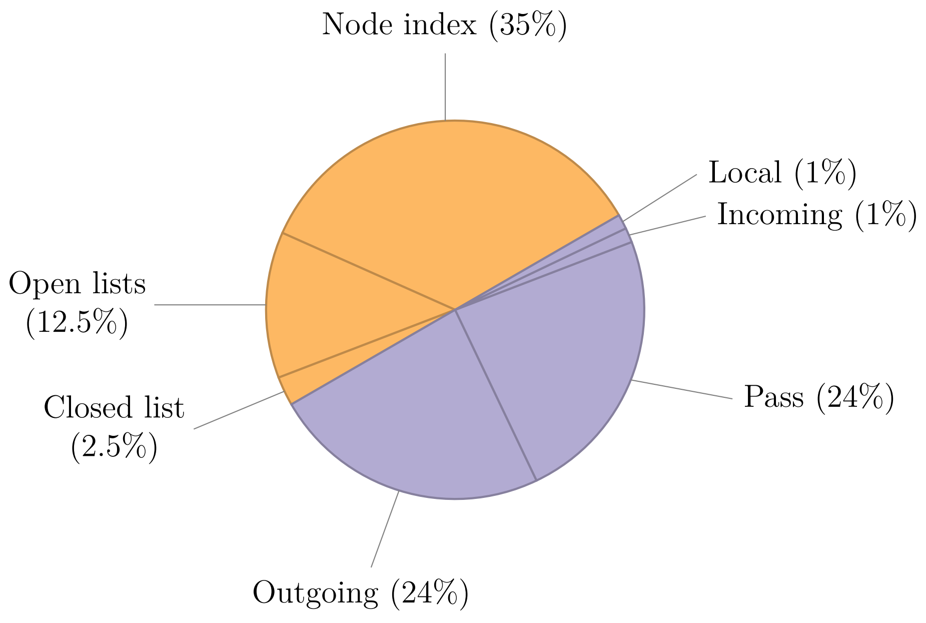


Figure 3.3: Minsky system overview. Note that NVLinks are bonded together in gangs of 2. Based on Caldeira, Haug, and Vetter [35, Figure 1-12].



(a) Minsky



(b) Yoda

Figure 3.4: Memory provisioning used for benchmarking on the Minsky and Yoda systems.

Chapter 4

Results and Discussion

Here we present the results of our benchmarks described in Section 3.2, and analyse and discuss the findings. We especially look at wall-clock runtime, node expansion rate, and the search- and communication overhead of our program runs, as well as discussing a type of overhead we call Termination Overhead.

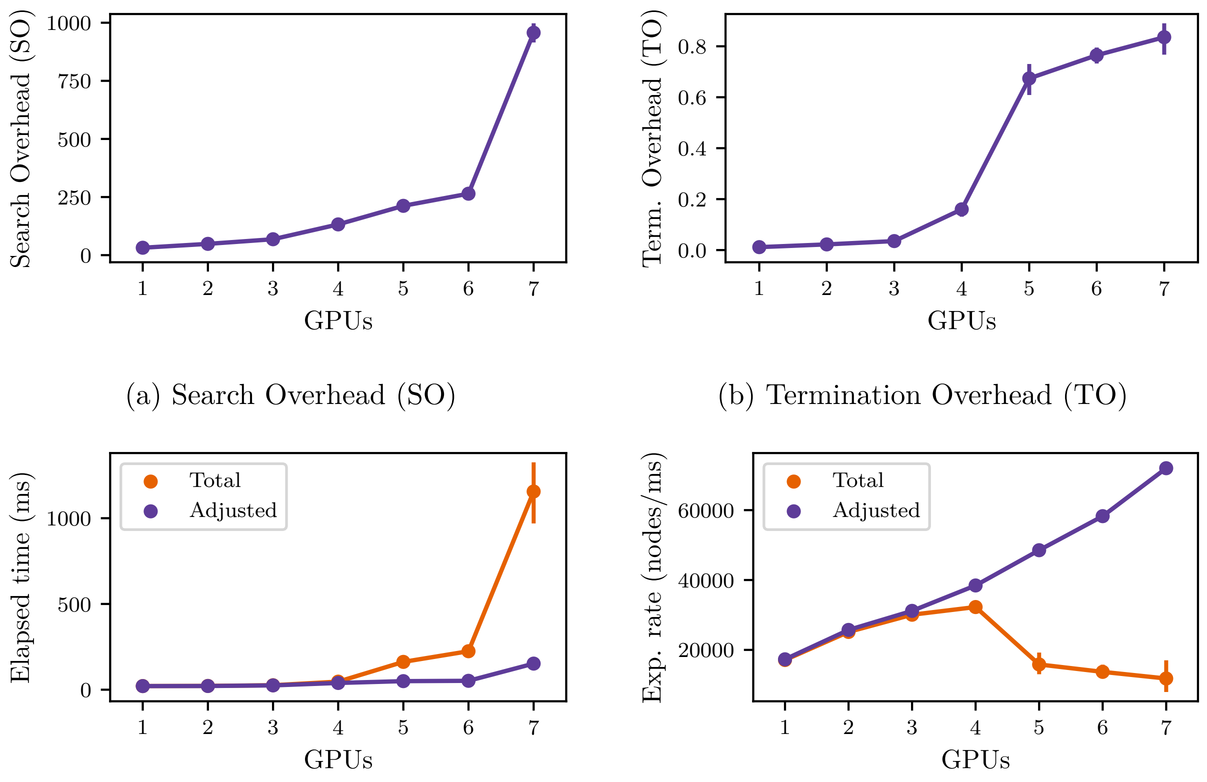
4.1 Benchmark results

The results of the 24-puzzle benchmark runs are listed in Tables 4.1 and 4.3. Because we only list results where the program was able to successfully find an optimal solution at least once, some results are missing. Unsuccessful runs are in virtually all cases caused by the algorithm running out of memory before finding a solution.

For 15-puzzles, results are listed in Table 4.2. We will focus mainly on the benchmark results for 24-puzzle solves, as these are the hardest (in terms of sequential node expansions) problems to solve, and are better at stress-testing the multi-GPU system.

In Tables 4.1 and 4.3 some patterns immediately stand out. Firstly, the “split” AZH pattern seems to successfully solve 24-puzzle problems at a higher rate than the “block” pattern. Further, the number of successful solves generally decrease as GPUs are added. We also see that some problems can’t be solved by a single GPU using our implementation, like 5x5-300, 5x5-400 and 5x5-500.

We can also glimpse an alarming pattern in the elapsed time for runs in both Tables 4.1 and 4.2: it appears to rise by a significant amount when more GPUs are used! This is illustrated for the easiest 24-puzzle on the Yoda system in Figure 4.1c, and for the four hardest 24-puzzle problems on



(a) Search Overhead (SO)

(b) Termination Overhead (TO)

(c) Elapsed time

(d) Node expansion rate

Figure 4.1: Various metrics of solves of the 5x5-100 on the Yoda system. “Total” is the overall node expansion rate, while “Adjusted” is the rate from the start of the run until the optimal solution is discovered (i.e. TO is discounted). Error bars where present represent 95% confidence intervals.

the Minsky system in Figure 4.2.

4.2 Termination Overhead

By monitoring the amount of work in communication buffers, we find the cause of increased runtime by the addition of GPUs. On double-GPU benchmarks on the Minsky system, the inter-GPU communication is direct between the two devices, and not a lot of overhead is introduced. Additionally, the communication happens over NVLink (Figure 3.3). Using further GPUs means some communication becomes indirect, going through the ring topology of the DWL. Additionally, data now has to flow through the system bus

Table 4.1: Benchmark results for 24-puzzles on the Minsky system.

Problem	AZH	GPUs	Successful runs (of 15)	Elapsed time (ms)	Adjusted time (ms)	TO	SO	CO
5x5-100	block	1	15	17	17	0.01	39.26	0.00
5x5-100	block	2	15	28	27	0.06	63.99	0.29
5x5-100	block	3	15	575	105	0.82	403.96	0.35
5x5-100	block	4	2	56 847	3917	0.93	21 115.06	0.43
5x5-200	block	1	15	3204	3202	0.00	4.43	0.00
5x5-200	block	2	15	3503	3501	0.00	7.11	0.30
5x5-200	block	3	1	11 795	7013	0.41	16.72	0.36
5x5-300	block	2	4	11 541	11 539	0.00	3.23	0.28
5x5-400	block	2	3	10 484	10 481	0.00	2.00	0.24
5x5-500	block	2	9	10 110	10 107	0.00	1.84	0.24
5x5-100	split	1	15	17	17	0.01	39.26	0.00
5x5-100	split	2	15	21	20	0.02	61.01	0.09
5x5-100	split	3	15	51	41	0.19	133.32	0.13
5x5-100	split	4	15	105	71	0.33	355.93	0.13
5x5-200	split	1	15	3766	3764	0.00	5.20	0.00
5x5-200	split	2	15	3309	3306	0.00	6.68	0.11
5x5-200	split	3	15	9148	5812	0.36	14.46	0.14
5x5-200	split	4	14	18 481	8669	0.53	28.17	0.12
5x5-300	split	2	9	9760	9758	0.00	2.77	0.10
5x5-300	split	3	9	16 347	10 735	0.34	3.78	0.15
5x5-300	split	4	5	18 476	8452	0.54	3.84	0.12
5x5-400	split	2	2	11 661	11 659	0.00	2.19	0.08
5x5-400	split	3	2	22 339	11 668	0.48	2.81	0.14
5x5-400	split	4	1	28 857	7824	0.73	2.42	0.10
5x5-500	split	2	8	10 615	10 613	0.00	1.92	0.10
5x5-500	split	3	5	20 093	10 795	0.46	2.43	0.14
5x5-500	split	4	7	46 254	9174	0.80	2.63	0.14

Table 4.2: Benchmark results for 15-puzzles on the Minsky system.

Problem	GPUs	Successful runs (of 15)	Elapsed time (ms)	Adjusted time (ms)	TO	SO	CO
4x4-300	1	15	19	18	0.01	4.13	0.00
4x4-300	2	15	26	25	0.03	6.69	0.09
4x4-300	3	15	89	54	0.39	18.54	0.13
4x4-300	4	15	484	128	0.74	64.75	0.15
4x4-1200	1	15	6226	6224	0.00	3.13	0.00
4x4-1200	2	15	4180	4178	0.00	2.84	0.12
4x4-1200	3	15	35 321	9170	0.74	7.59	0.13
4x4-1200	4	6	114 935	10 368	0.91	10.96	0.14
4x4-1400	1	15	238	236	0.01	1.97	0.00
4x4-1400	2	15	248	247	0.01	2.65	0.12
4x4-1400	3	15	958	365	0.62	4.64	0.14
4x4-1400	4	15	5663	883	0.84	15.23	0.13
4x4-1600	1	15	161	159	0.01	2.12	0.00
4x4-1600	2	15	156	153	0.02	2.51	0.12
4x4-1600	3	15	474	124	0.74	2.35	0.12
4x4-1600	4	15	1930	89	0.95	2.10	0.14
4x4-1900	1	15	283	281	0.01	1.69	0.00
4x4-1900	2	15	350	348	0.00	2.73	0.12
4x4-1900	3	15	826	335	0.59	3.02	0.13
4x4-1900	4	15	4653	612	0.87	7.60	0.13

Table 4.3: Benchmark results for the Yoda system.

Problem	AZH	GPUs	Successful runs (of 15)	Elapsed time (ms)	Adjusted time (ms)	SO	CO	TO
5x5-100	split	1	15	21	21	31.84	0.00	0.01
5x5-100	split	2	15	22	22	48.51	0.09	0.02
5x5-100	split	3	15	26	25	68.41	0.14	0.04
5x5-100	split	4	15	47	39	132.29	0.13	0.16
5x5-100	split	5	15	162	50	212.12	0.18	0.69
5x5-100	split	6	15	224	52	264.37	0.17	0.77
5x5-100	split	7	15	1156	152	957.09	0.17	0.87
5x5-100	block	1	15	21	21	31.83	0.00	0.01
5x5-100	block	2	15	21	21	39.06	0.29	0.02
5x5-100	block	3	15	29	28	63.25	0.38	0.04
5x5-100	block	4	15	4014	2238	8278.20	0.43	0.44
5x5-100	block	5	5	19 007	4813	21 009.73	0.42	0.75
4x4-300	split	1	15	25	24	3.62	0.00	0.02
4x4-300	split	2	15	25	24	4.90	0.09	0.02
4x4-300	split	3	15	35	32	7.19	0.13	0.08
4x4-300	split	4	15	201	117	40.15	0.14	0.42
4x4-300	split	5	15	335	71	28.51	0.17	0.79
4x4-300	split	6	14	1915	132	68.26	0.17	0.93
4x4-300	split	7	14	752	90	51.81	0.17	0.88

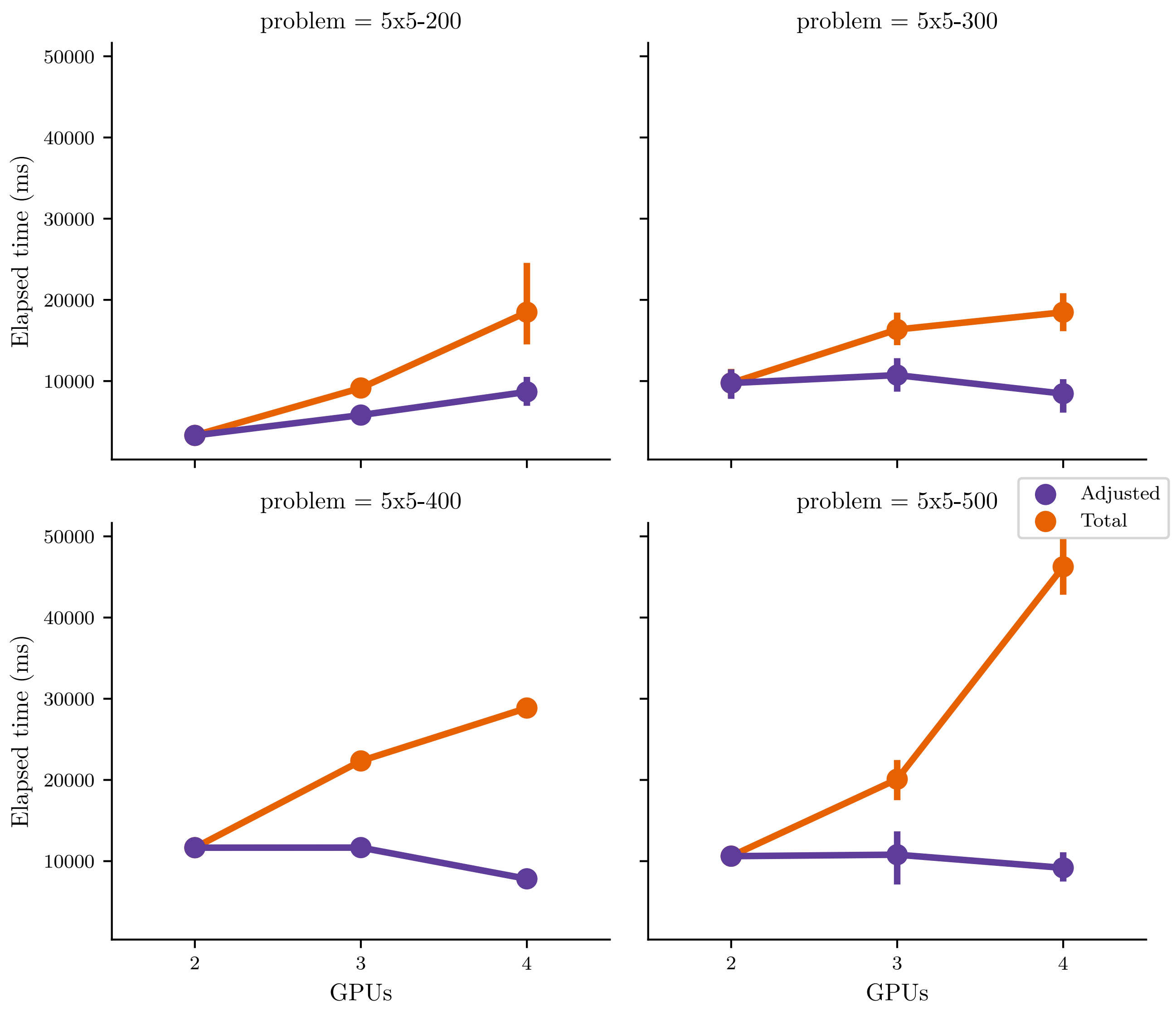


Figure 4.2: Elapsed time for 24-puzzle problems on the Minsky system. "Total" shows the total elapsed time for the algorithm to finish, while "Adjusted" shows the adjusted elapsed time, which disregards Termination Overhead (TO). Error bars indicate 95% confidence intervals.

of the Minsky, instead of NVLink, forming a bottleneck.

The result is that, on 3- and 4-GPU runs on the Minsky, Groute communication links become backed up with search nodes to be transferred to other devices. This means a time increase if search nodes on optimal paths spend a long time queued for transfer, but the main source of increased runtime appears *after* the optimal solution has been discovered. In order to achieve admissibility, we have to guarantee that a discovered solution is optimal. To do this, we have to examine every search node in the open lists, including those in communication buffers, and make sure no node exists with an $\hat{f}$-value lower than the cost of the discovered solution.

Groute's DWL allows us to discard incoming and outgoing search nodes in the SplitReceive and SplitSend kernels respectively. However, once a node has passed SplitSend and is queued for transfer, the next opportunity to discard it is after it has been moved to the next GPU in the ring. This is the main source of what we call termination overhead: waiting for queued search node transfers, only to discard the majority of them immediately. If we imagine we can deal with this overhead as efficiently on multiple GPUs as on a single GPU, we can subtract the elapsed time between finding the optimal solution and program exit, resulting in the *adjusted time* in Table 4.1. We measure the fraction of time spent waiting for communication after finding the solution of the full runtime, and call it the TO.

Figure 4.1b shows how the TO changes with added GPUs on the Yoda system. We see a massive jump going from four to five GPUs, which can be attributed to the topology of the system, similarly to the jump in TO when going from two to three GPUs on the Minsky system. On Yoda, while all inter-GPU communication happens over PCIe, it is not symmetric. Some pairs of nodes can communicate traversing only a single PCIe switch, while communication between the first and second quartet of GPUs passes through the system bus, creating a similar bottleneck on Yoda as on Minsky.

The TO-adjusted time is more favourable in terms of runtime, and shows that for more difficult problems (like 5x5-300, 400 and 500), the 4-GPU run discovers the optimal solution in the shortest amount of time. This is visualised in Figure 4.2. Simultaneously, we see that TO increases along with the number of GPUs.

The question becomes: how can we get rid of the termination overhead? We would like to simply sift through all search nodes in outbound communication buffers and discard the majority of them, instead of being bottlenecked by their transmit times. The hindrance here is practical: Groute DWL communication buffers are highly optimised and are not exposed by the framework to the higher-level algorithms implemented using it. With some modifications to the framework, effectivating A* search termination should be feasible; unfortunately, we were not able to explore this properly in the scope of this work.

4.3 Node expansion rate

While our implementation does not provide a speed-up in terms of wall-clock elapsed time, we can examine the *node expansion rate*. That is to say, how many search nodes is our implementation able to expand per time unit, and how does this scale with the number of GPUs? This is also measured by Zhou and Zeng [2] in their single-GPU implementation of the GA* algorithm.

Similarly to the elapsed time metric, the node expansion rate suffers from Termination Overhead (TO). Very few new nodes are expanded after the optimal solution has been detected. Thus, we also examine the adjusted node expansion rate, where we only measure the expansion rate before the optimal solution was discovered, in order to see how it would scale if we did not have TO.

The node expansion rates for the four hardest 24-puzzles on the Minsky system are shown in Figure 4.3. The expansion rates for the easiest 24-puzzle on the Yoda system are shown in Figure 4.1d. Again, similarly to the elapsed times shown in Figure 4.2 and Figure 4.1c, the overall rates scale poorly, and become worse as more GPUs are added. However, if we disregard the TO, the node expansion rate does improve with additional GPUs. It scales approximately linearly with added devices, adding around 9000 nodes/ms per device, or a little over 50% of the performance of a lone GPU. This indicates that if we can indeed eliminate Termination Overhead, we can achieve good scaling in terms of node expansion rates.

4.4 Search and Communication Overhead

To gain further insight about our implementation benchmarks, we consider the Search Overhead (SO) and Communication Overhead (CO), as defined in Section 2.4, of the algorithm runs. In order to calculate SO, we need to know how many nodes are expanded by a sequential implementation of the A* algorithm with the same heuristic. We used a straightforward CPU implementation of Algorithm 2.1, and its node expansions are listed in Table 3.1. CO is calculated by keeping track of how many generated search nodes are kept locally on the same device, and how many are transferred to a different device. The SO and CO of our benchmark runs are included in Tables 4.1 and 4.2.

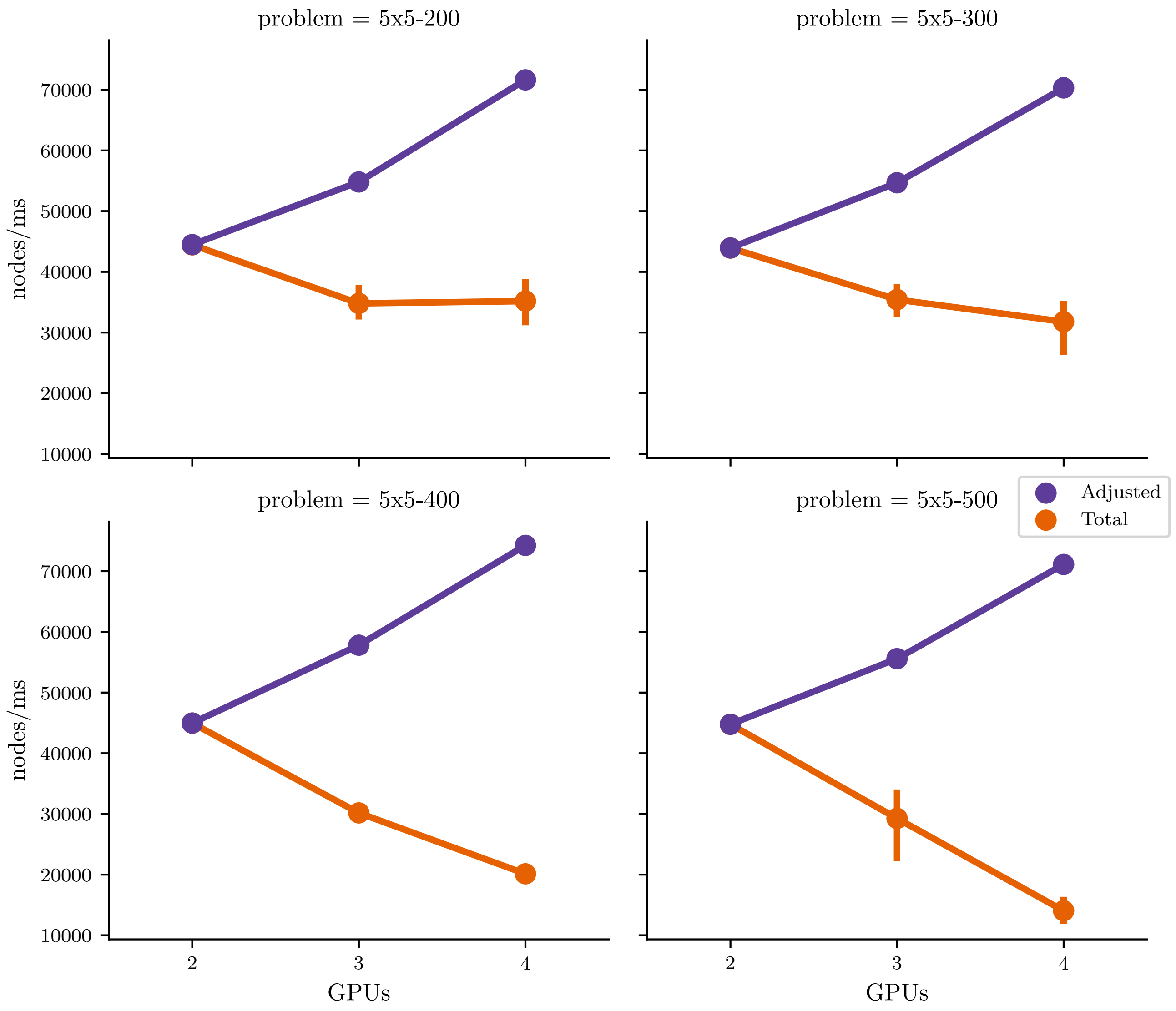


Figure 4.3: Node expansion rates for 24-puzzle problems on the Minsky system. “Total” is the overall node expansion rate, while “Adjusted” is the rate from the start of the run until the optimal solution is discovered (i.e. TO is discounted). Error bars indicate 95% confidence intervals.

Having reaffirmed in the previous section that too much inter-GPU communication is detrimental to performance, it is clear that we should aim to minimise CO. It is useful to observe SO and CO in conjunction. There is an intuitive relationship between them: a low CO means we spend little resources communicating, but could lead to higher SO due to load imbalance [16]. We also have to consider the number of devices when studying CO; on a two-device run we would expect a CO of 50% using a naive uniform hash function, but we expect it to be 75% for a four-device setup.

In Table 4.1, we can compare the CO of the block and split feature projections used by AZH to distribute search nodes. For two GPUs, the block abstraction sends between 24% and 30% of generated nodes to the other device. While this seems good compared to the baseline 50%, it is clearly not good enough, as the implementation struggles to solve most problems using this abstraction. The runs using the split abstraction present better numbers: 15% or fewer of nodes generated are sent to other devices – even on 4-GPU runs where the baseline is 75%. The improvement is evident, as using the split abstraction allows us to solve more difficult problems with a higher number of GPUs without overflowing communication buffers.

However, as already stated, a reduced CO comes at a cost: we also have to evaluate the accompanying SO. In Figure 4.4, the SO and CO of our 2-GPU runs for 24-puzzles on the Minsky system are plotted. The SO for the easiest 24-puzzle on Yoda is also shown in Figure 4.1a. We can see how the SO and CO change for each puzzle with the two different AZH abstractions. The CO in all cases are more than halved, and nearly divided by three in some. The SO, however, does not increase by a proportional amount; in one case, it even decreases. This tells us that the split abstraction is probably preferable, and this is backed up by the number of successful solves seen in Table 4.1.

The decrease of both SO *and* CO for the 5x5-200 board can likely be attributed to the block abstraction solver spending time expanding useless nodes while the optimal path(s) are stuck in communication buffers.

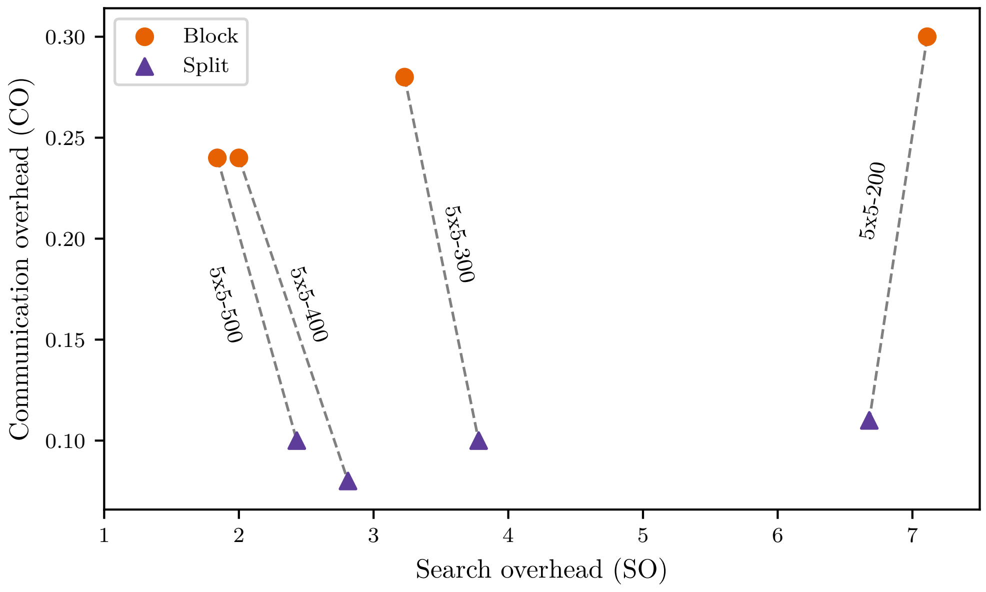


Figure 4.4: Search and communication overheads of 2-GPUs solves of four 24-puzzles with two different AZH abstractions. Solves of the same puzzle are connected and labelled.

Chapter 5

Conclusions and Future Work

A* informed search is an interesting algorithm with applications in computational biology, natural language processing, pathfinding, puzzle solving, and more. We have implemented a version of it that utilises multiple GPU devices on a single host for performing the search. Multi-GPU compute nodes are common in today's high-performance computing landscape due to the massive amount of raw compute power they offer. Programming such systems, however, is challenging, particularly for irregular algorithms such as A*.

We have implemented a multi-GPU-distributed variant of the GA* search algorithm [2] for sliding tile puzzle solving, using the Groute asynchronous multi-GPU programming framework [1] to implement inter-GPU coordination and communication, and Abstract Zobrist Hashing [16] for work distribution and load balancing. This is the first multi-GPU variant of the A* algorithm, as far as the authors are aware.

We have shown that multi-GPU A* search can solve certain problems that are too difficult to solve using the same algorithm on a single GPU. We have also shown that the node expansion rate during search before an optimal solution has been discovered can scale approximately linearly with the number of GPUs utilised, and that each additional GPU can increase the node expansion rate by at least 50% of the performance of a single GPU. This, combined with the increased memory capacity of multiple GPUs, indicates that multi-GPU systems may be a good candidate for solving difficult search problems efficiently.

Our implementation suffers from overhead induced by inter-GPU communication buffers filling up, something that manifests itself clearly as Termination Overhead. We have also seen that hardware topology has a significant impact on performance, and that an asymmetric layout can produce significant bottlenecks in a ring-based distributed worklist application. Much of the overhead can conceivably be solved by more careful programming and implementation with the lower level abstractions of Groute, rather than using out-of-the-box solutions like the DWL implementation.

Our implementation has identified many challenges and hurdles in distributing A* across GPU devices, but shows potential if these can be overcome. It serves as a proof-of-concept and jumping-off point for further multi-GPU A* research.

5.1 Future Work

The largest issue identified with our implementation is the Termination Overhead (TO) it introduces and how it means adding more GPUs can increase the runtime of the algorithm drastically, instead of decreasing it. By modifying the Groute distributed worklist implementation [1], or using an ad-hoc implementation with Groute's lower-level abstractions, there is a possibility that TO can be reduced drastically.

Further, it is quite possible that the ring topology is a suboptimal choice for the DWL, especially considering the typical hardware layout of multi-GPU systems, where communication between all pairs of devices is not symmetric. It would be interesting to look into alternative schemes for implementing the DWL. If more efficient methods can be found, it could reduce the accumulation of search nodes in communication buffers, which would reduce TO, improve responsiveness of the system, and let us reduce the memory dedicated to communication buffers, freeing memory for use by the central A* algorithm.

Another possible optimisation is to improve the receiving of search nodes from other devices. Using Groute, nodes are first placed into a local worklist, before being copied to open list heaps (Figure 3.1). It may be possible to move search nodes more efficiently from the SplitReceive and SplitSend kernels to the A* open lists, eliminating the intermediate local worklist storage.

We have implemented the GA* algorithm of Zhou and Zeng [2], however, other work on GPU A* algorithms exist [9], [14]. Exploring other A* algorithm variations and techniques in a multi-GPU setting is unquestionably also a source of future work.

Bibliography

[1] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali, "Groute: An asynchronous multi-gpu programming model for irregular computations," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ACM, 2017, pp. 235–248.

[2] Y. Zhou and J. Zeng, "Massively parallel a* search on a gpu.," in *AAAI*, 2015, pp. 1248–1255.

[3] E. Burns, S. Lemons, W. Ruml, and R. Zhou, "Best-first heuristic search for multicore machines," *Journal of Artificial Intelligence Research*, vol. 39, pp. 689–743, 2010.

[4] D. G. Spampinato, A. C. Elster, and T. Natvig, "Modelling multi-gpu systems.," in *PARCO*, 2009, pp. 562–569.

[5] R. Barzilay and L. Lee, "Bootstrapping lexical choice via multiple-sequence alignment," in *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10*, Association for Computational Linguistics, 2002, pp. 164–171.

[6] A. Kishimoto, A. S. Fukunaga, A. Botea, *et al.*, "Scalable, parallel best-first search for optimal sequential planning.," in *ICAPS*, 2009, pp. 201–208.

[7] H. Lien, "Procedural generation of road for use in the snow simulator," Department of Computer Science, NTNU, Specialization project, 2011.

[8] Y. Zhou, W. Xu, B. R. Donald, and J. Zeng, "An efficient parallel algorithm for accelerating computational protein design," *Bioinformatics*, vol. 30, no. 12, pp. i255–i263, 2014.

[9] H. Hiroki, I. Naoaki, and M. Hirokazu, "Gpu-acceleration of optimal permutation-puzzle solving," in *Proceedings of the 2015 International Workshop on Parallel Symbolic Computation*, ACM, 2015, pp. 61–69.

[10] A. Fukunaga, A. Botea, Y. Jinnai, and A. Kishimoto, "A survey of parallel a*," *arXiv preprint arXiv:1708.05296*, 2017.

[11] T. Cazenave, "Approximate multiple sequence alignment with a-star,"

[12] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[13] R. Inam, "A* algorithm for multicore graphics processors," *Master's thesis, Chalmers University of Technology*, 2010.

[14] S. Horie and A. Fukunaga, "Block-parallel ida* for gpus," in *Proceedings of the Tenth International Symposium on Combinatorial Search, Edited by Alex Fukunaga and Akihiro Kishimoto*, 2017, pp. 16–17.

[15] M. Evett, J. Hendler, A. Mahanti, and D. Nau, "Pra*: Massively parallel heuristic search," *Journal of Parallel and Distributed Computing*, vol. 25, no. 2, pp. 133–143, 1995.

[16] Y. Jinnai and A. Fukunaga, "Abstract zobrist hashing: An efficient work distribution method for parallel best-first search," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[17] A. L. Zobrist, "A new hashing method with application for game playing," *ICCA journal*, vol. 13, no. 2, pp. 69–73, 1970.

[18] J. D. Owens, M. Houston, D. Luebke, *et al.*, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[19] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[20] J. Nickolls and W. J. Dally, "The gpu computing era," *IEEE micro*, vol. 30, no. 2, 2010.

[21] Nvidia Corporation, *NVIDIA Tesla P100*, `https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf`, Accessed: 2017-12-13, 2016.

[22] ——, *Nvidia Tesla V100 GPU Architecture*, `http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`, Accessed: 2018-07-29, 2017.

[23] ——, *NVIDIA's Next Generation CUDA Compute Architecture: Kepler TM GK110*, `https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`, Accessed: 2018-07-29, 2012.

[24] ——, *GeForce GTX TITAN Black | Specifications | GeForce*, `https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-black/specifications`, Accessed: 2018-07-29, 2014.

[25] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on gpus," in *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, IEEE, 2012, pp. 141–151.

[26] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," in *ACM SIGPLAN Notices*, ACM, vol. 47, 2012, pp. 117–128.

[27] Nvidia Corporation, *Fermi*, `https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`, Accessed: 2017-12-14, 2009.

[28] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach.* Malaysia; Pearson Education Limited, 2016.

[29] Y. Zhou and J. Zeng, *Massively parallel a* search on a gpu: Appendix*, `http://iiis.tsinghua.edu.cn/~compbio/papers/aaai2015apx.pdf`, 2014.

[30] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[31] A. Felner, R. E. Korf, and S. Hanan, "Additive pattern database heuristics," *Journal of Artificial Intelligence Research*, vol. 22, pp. 279–318, 2004.

[32] R. E. Korf and A. Felner, "Disjoint pattern database heuristics," *Artificial intelligence*, vol. 134, no. 1-2, pp. 9–22, 2002.

[33] J. C. Culberson and J. Schaeffer, "Pattern databases," *Computational Intelligence*, vol. 14, no. 3, pp. 318–334, 1998.

[34] M. Steinberger, M. Kenzel, B. Kainz, and D. Schmalstieg, "Scatteralloc: Massively parallel dynamic memory allocation for the gpu," in *Innovative Parallel Computing (InPar), 2012*, IEEE, 2012, pp. 1–10.

[35] A. B. Caldeira, V. Haug, and S. Vetter, *Ibm power system s822lc for high performance computing introduction and technical overview*, `https://www.redbooks.ibm.com/Redbooks.nsf/RedbookAbstracts/redp5405.html`, Accessed: 2018-07-24, 2016.

Appendix A

Multi-GPU A* code

Here we list some of the code from our multi-GPU implementation of A* search. The full implementation exists as a Git repository on GitHub under `acelster/Groute-Astar`[1], and contains instructions on building and running the code.

Listings A.1 and A.2 show the `on_receive` and `on_send` callbacks used by the Groute distributed worklist system. Note that in Listing A.2, a compile-time directive `MEASURE_CO` can be enabled or disabled in order to measure the Communication Overhead (CO) of the implementation.

Listing A.3 shows the principal components of the main work loop, which runs in a separate thread for each GPU device. Note the heavy use of C++ templating in Groute components, making it flexible and easy to adapt for different algorithms. The loop continually processes incoming work items, passing them to `WorkKernel` (which corresponds to ExtractExpand in Figure 3.1). It also checks for new solutions on each iteration, and distributes any improved solution to all devices.

[1]Full URL: `https://github.com/acelster/Groute-Astar`

Listing A.1: The `on_receive` callback.

```
__device__ groute::SplitFlags on_receive(remote_work<N> const & work) {
    /* drop work that is definitely worse than the current best solution */
    if (work.f_value >= packed_solution_length(d_best_solution)) {
        return groute::SF_None;
    }

    if (device_number == work.hash % num_devices) {
        if (closed_set[work.board.hash_value() % closed_set_capacity] != UINT32_MAX && nodes[closed_set[work.board.hash_value() % closed_set_capacity]].board == work.board) {
            /* node is owned by this device, and in the closed set; drop it */
            return groute::SF_None;
        } else {
            /* node is owned by this device, take it */
            return groute::SF_Take;
        }
    } else {
        return groute::SF_Pass;
    }
}
```

Listing A.2: The `on_send` callback.

```
__device__ groute::SplitFlags on_send(local_work<N> work) {
    if (device_number == work.hash % num_devices) {
        if (MEASURE_CO)
            atomicAdd(nodes_kept, 1);
        return groute::SF_Take;
    } else {
        if (MEASURE_CO)
            atomicAdd(nodes_sent, 1);
        return groute::SF_Pass;
    }
}
```

Listing A.3: The main work loop host code, with some elements omitted for brevity.

```
void Work(groute::IDistributedWorklist<TLocal, TRemote> &dwl,
          groute::IDistributedWorklistPeer<TLocal, TRemote,
              DWCallbacks> *const peer,
          groute::Stream &stream,
          const WorkArgs &... args)
{
    auto &input_worklist = peer->GetRemoteInputQueue();
    auto &workspace = peer->GetLocalQueue(0);
    DWCallbacks callbacks = peer->GetDeviceCallbacks();

    while (dwl.HasWork())
    {
        auto input_segs = peer->WaitForInputWork(stream);
        for (auto subseg : input_segs)
        {
            groute::WorkKernel
            <groute::dev::WorkSourceArray<TLocal>, TLocal,
                TRemote, DWCallbacks, TWork, WorkArgs...>
            <<<grid_dims, block_dims, 0, stream.cuda_stream>>>(
                groute::dev::WorkSourceArray<TLocal>(subseg.
                    GetSegmentPtr(), subseg.GetSegmentSize()),
                workspace.DeviceObject(),
                callbacks,
                args...
            );

            input_worklist.PopAsync(subseg.GetSegmentSize(),
                stream.cuda_stream);
        }

        auto output_seg = workspace.GetSeg(stream);
        peer->SplitSend(output_seg, stream);
        workspace.ResetAsync(stream.cuda_stream);

        /* update solution if we found one */
        dwl.update_best_solution(get_gpu_memory(device_data->
            solution)));
    }
}
```