Czech Technical University in Prague

Faculty of Electrical Engineering

BACHELOR THESIS

2012

David Hlavatý

Czech Technical University in Prague

Faculty of Electrical Engineering

Department of Cybernetics

Parallel Trajectory Planning on GPU

B.c. Programme: Open Informatics
Specialisation: Computer and Information science

May 2012

Author: David Hlavatý
Supervisor: Ing. David Šišlák, PhD.

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

BACHELOR PROJECT ASSIGNMENT

Student: David Hlavatý

Study programme: Open Informatics

Specialisation: Computer and Information Science

Title of Bachelor Project: Parallel Trajectory Planning on GPU

Guidelines:

1. Find existing methods for parallel trajectory planning.
2. Get familiar with CUDA architecture available on nVidia graphics cards.
3. Design parallelization of A* and AA* algorithms suitable for running on CUDA architecture.
4. Implement designed parallel version of A* running on CUDA.
5. Implement designed parallel version of AA*.

Bibliography/Sources:

[1] E. Burns, S. Lemons, R. Zhou, W. Ruml: Best-first heuristic search for multi-core machines. In Proceedings of the 21st international jont conference on Artifical intelligence, pp. 449– 455, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.
[2] E. Burns, S. Lemons, R. Zhou, W. Ruml: Parallel best-first search: The role of abstraction. In Proceedings of the AAAI-10 Workshop on Abstraction, Reformulation, and Approximation, 2010.
[3] M. Evett, A. Mahanti, D. Nau, J. Hendler, J. Hendler: PRA*: Massively parallel heuristic search. Journal of Parallel and Distributed Computing, 25:133–143, 1995.
[4] A. Kishimoto, A. Fukunaga, A. Botea: Scalable, Parallel Best-First Search for Optimal Sequential Planning. In Proceedings of the International Conference on Automated Scheduling and Planning ICAPS-09, pp. 201–208, Thessaloniki, Greece, 2009.

Bachelor Project Supervisor: Ing. David Šišlák, Ph.D.

Valid until: the end of the winter semester of academic year 2012/2013

L.S.

prof. Ing. Vladimír Mařík, DrSc.
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 9, 2012

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: David Hlavatý

Studijní program: Otevřená informatika (bakalářský)

Obor: Informatika a počítačové vědy

Název tématu: Paralelní plánování trajektorie na GPU

Pokyny pro vypracování:

1. Prostudujte způsoby paralelizace plánovacích úloh ze zadané literatury.
2. Prostudujte architekturu nVidia CUDA.
3. Navrhněte způsob paralelizace plánovací úlohy.
4. Naimplementujte paralelizaci A* na architektuře CUDA.
5. Naimplementujte paralelizaci plánovacího algoritmu AA* pro systém AgentFly.

Seznam odborné literatury:

[1] E. Burns, S. Lemons, R. Zhou, W. Ruml: Best-first heuristic search for multi-core machines. In Proceedings of the 21st international jont conference on Artifical intelligence, pp. 449– 455, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[2] E. Burns, S. Lemons, R. Zhou, W. Ruml: Parallel best-first search: The role of abstraction. In Proceedings of the AAAI-10 Workshop on Abstraction, Reformulation, and Approximation, 2010.

[3] M. Evett, A. Mahanti, D. Nau, J. Hendler, J. Hendler: PRA*: Massively parallel heuristic search. Journal of Parallel and Distributed Computing, 25:133–143, 1995.

[4] A. Kishimoto, A. Fukunaga, A. Botea: Scalable, Parallel Best-First Search for Optimal Sequential Planning. In Proceedings of the International Conference on Automated Scheduling and Planning ICAPS-09, pp. 201–208, Thessaloniki, Greece, 2009.

Vedoucí bakalářské práce: Ing. David Šišlák, Ph.D.

Platnost zadání: do konce zimního semestru 2012/2013

L.S.

prof. Ing. Vladimír Mařík, DrSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 9. 1. 2012

I hereby declare that this bachelor thesis is completely my own work and that I used only the cited sources in accordance with the Methodical instruction about observance of ethical principles of preparation of university final projects.

Prague, May 25, 2012

David Allan

Signature

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etnických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 25. Května 2012

David Allan

Podpis

Acknowledgments

I would like to thank to my supervisor, Ing. David Šišlák, PhD., for his support and help during my bachalor project, and to Štěpán Kopřiva, who brought me to this project.

My thanks also belongs to my family for their love, patience and support, especially to my father Zdeněk who also contributed to this thesis by several drawings.

Abstract

The release of the CUDA architecture made massively parallel computing possible on ordinary desktops and opened a door to enormous computing power of graphics adapters. The trajectory planning for aerial vehicles is one of the tasks that can benefit from it. The sought path must respect all physical limitations of the airplane and avoid all no-flight zones.

The thesis presents two algorithms for trajectory planning on the CUDA architecture: a parallel version of A* algorithm and Accelerated A* algorithm that uses varying planning steps to speed up the planning. The parallelization relies on a distribution of states between individual threads. To implement the proposed algorithms, a block synchronization that is not officially supported in CUDA is required. Two solutions to this problem are given in the thesis. Both algorithms are experimentally evaluated and compared to their sequential version.

Abstrakt

Vytvoření CUDA architektury umožnilo masivně paralelní vypočty na běžných osobních počitačích a otevřelo dvěře k obrovskému výpočetnímu výkonu grafických karet. Plánování trajektorie v letecké dopravě je jedna z úloh, která může být paralelizovaná. Hledaná cesta musí respektovat jednak všechna fyzická omezení letadla a také všechny bezletové zóny.

Tato práce prezentuje dva algoritmy pro plánování trasy na CUDA architetuře: paralelní verzi A* algoritmu a Accelerated A* algoritmu, který používá promněnlivý krok k zrychlení prohledávání. Implementace navržených algoritmů vyžaduje synchronizaci mezi bloky v CUDA architektuře, jenž není oficiálně podporována. V praci jsou navrhnuta dvě možná řešení tohoto problému. Oba algoritmy byly experimentálně vyhodnoceny a porovnány s jejich neparaelizovanými verzemi.

Contents

List of Figures

List of Algorithms

List of Tables

List of Acronyms

AA* Accelerated A*.

ALU Arithmetic Logic Unit.

CDHA* CUDA Hash Distributed A*.

CM Connection Machine.

CPAA* CUDA Parallel Accelerated A*.

CPU Central Processing Unit.

DRAM Dynamic Random Access Memory.

FIFO First-In, First-Out.

GPU Graphics Processing Unit.

HDA* Hash Distributed A*.

LIFO Last-In, Last-Out.

PBNF Parallel Best-N Block-First.

PRA* Parallel Retraction A*.

PSDD Parallel Structured Duplicate Detection.

RA* Retraction A*.

SDD Structured Duplicate Detection.

SFU Special Function Unit.

SIMD Single-Instruction, Miltiple-Data.

SIMT Single-Instruction, Multiple-Thread.

SM Streaming Multiprocessor.

Chapter 1

Introduction

Path finding goes back a long time before the first computers were invented. Whenever someone needed to move from one place to another, they always wanted to use the "best" possible path. The meaning of the "best" path varied according to the nature of the journey. For a group of merchants, the main aspect was the safety of the road. For a king moving his army, the goal was to stay undetected from his enemies. The travel plan had to be prepared in advance and modification required to stop and look at the map.

The invention of computers made planning much easier. Modern computers can go through a large amount of possible solutions and return the desired one in a matter of seconds. A typical usage is in navigation for cars in combination with the GPS system. It calculates a route at the beginning and then it provides directions during the journey. If a wrong turn is taken, it immediately recalculates the route and provides alternative directions.

Nevertheless, the application of the path planning does not have to be limited only to cars. Another domain where it is sought to find a solution quickly is the planning for aerial vehicles. It might look at the first sight that it is not a complicated task, because aircrafts can fly straight towards their goal. However, just as cars are limited by roads and traffic regulations, airplanes must avoid certain areas that they cannot fly in. For example, the United States Airspace contains more than 1400 no-flight zones consisting of areas around government buildings, schools or military objects. A zone can either be valid for the whole area or it can only be restricted to a specific altitude. In addition, some of these zones may be active only for a limited time. The sought trajectory must not only consider all no-flight zones, but it also has to respect the physical limitations of the airplane. That includes the minimum and maximum velocity, turn radius, or flight range. Hence, the trajectory planning for aerial vehicles is a computationally demanding task.

The flight plan planning is solved in the *AgentFly* system [1], that was developed by researchers from the *Agent Technology Center*. The AgentFly is a multi-agent system enabling large-scale simulation of civilian and unmanned air traffic.

Computers have changed a lot since the first computer was invented. For many years, the most reliable source for improvement was to increase the clock speed. However, at the same time, leading industrial companies were building supercomputers by adding additional processors. This was the foundation of *parallel computing*. The most famous supercomputer of today is the *IMB Blue Gene*, which aims to reach 20 *Petaflops* (i.e. $20 \cdot 10^{15}$ floating point operations per seconds) in 2012.

Unfortunately, supercomputers are extremely expensive. Therefore, for many years, parallel computing was only the domain of a few. A breakthrough came with the introduction of the NVIDIA CUDA architecture in 2006, which made massively parallel computing possible on ordinary desktops. CUDA enables computation power of the Graphical Processing Unit

(GPU) to be used for general-purpose computing. Due to the affordability of graphics cards, it soon got the attention of many programmers. Nowadays it is one of the most rapidly developing fields of modern computing.

1.1 Thesis Goals

Both the planning and parallel computing are very popular topics. Several parallel search algorithms were presented in the last decade, but none of them was designed for CUDA architecture. GPUs are specific for their execution model, when multiple threads execute the same instruction, and for their memory hierarchy.

This thesis aims to fill this gap. The goal is to explore the possibilities of parallel planning on CUDA architecture and to use this knowledge to design an algorithm, implement it and evaluate its performance. Parallel planning on the GPU would be especially useful for the AgentFly, where hundreds of trajectories need to be computed and the affordable GPU provides a nice way to boost performance.

1.2 Related work

Evett et al. [2] designed *Parallel Retraction A** (PRA*). It is a parallel version of *Retracting A** (RA*) search that uses a hash function to distribute states among available processors. The algorithm was tailored to run on a *Single-Instruction, Multiple-Data* (SIMD) *Connection Machine* (CM)[1]. Retracting A* is a modification of A* algorithm. It uses the same evaluation function to select the best state, but unlike A* it is not necessary to keep all expanded states in memory. States can be retracted and re-expanded later to save some space. This was an important property for PRA*, because each processor on the CM had only small local memory. PRA* exchanges states between processors by using synchronous communication. In the experiments performed by Evett et all., PRA* showed linear speed up against RA* with respect to a number of processors.

Kishimoto et al. [4] introduced *Hash Distributed A** (HDA*). Hash Distributed A* is based on PRA*. It also uses the hash function to distribute states, but unlike PRA* it uses asynchronous, non-blocking communication. It decreased the overheads caused by communication, since a processor does not have to wait until it is confirmed that the state was received. This makes it possible to used HDA* even on distributed systems, where communication between individual systems has high latency. Another aspect of HDA* is its simplicity. It does not use complicated retraction that was used by PRA*. It is merely a parallel version of A*. Therefore, the implementation is straightforward, which makes it easier to debug it. The execution time of HDA* was compared with A* and it was roughly five times faster on a cluster with eight cores and with 128 cores, HDA* was between 35 to 65 times faster.

Zhou et al. [5] described a new algorithm for graph search, called *Parallel Structured Duplicate Detection* (PSDD). *Structured Duplicate Detection* (SSD) is a graph search that allows an unused portion of the state space to be stored on external memory. It is based on many-to-one projection of the state space. The original state space is mapped to a new *abstract* space. States in the abstract space form an *abstract graph*. An abstract state s_1' is a successor of abstract state s_2' if there are two states s_1 and s_2 in the original state space, s_1 is successor of s_2, s_1 maps to s_1', and s_2 to s_2'. Abstract states, which do not have any common successor, are said to be *disjoint*. States in the original space that map to one

[1] A Connection Machine was a (series of) supercomputer(s) designed by W Hillis. See [3] for more information.

abstract state are referred together as *n-block*. A group of n-blocks of disjoint abstract states is called *free n-blocks*. *PSDD* uses n-blocks to divide the work between available processors (threads). Each processor gets one free n-block. A processor then expands all states from that n-block. Therefore, synchronization is not necessary at this point. Once all states are expanded, another free n-block is selected. PSDD was validated against SSD. PSDD with three threads was approximately three times faster.

Burns et al. [6, 7] further improved PSDD and presented *Parallel Best-N Block-First* (PBNF). Unlike PSDD, it works well with inadmissible heuristics and non-uniform and non-integer move costs. The improvement lies in the way an n-block is selected and expanded. The cost of each n-block is determined by the best state inside it (i.e. the state with the smallest evaluation function). Free n-blocks are ordered by their cost. When a new n-block is requested, the one with the best cost is selected. Then instead of expanding all states, only those with the cost smaller than the cost of the best free n-block are expanded. Once the cost of states inside the n-block gets bigger, the current n-block is released and a new one is acquired. PBNF was tested on several planning problems. Apart from one, it was faster than A* even with only one thread. With seven threads, it was always faster than A* and in most cases it was better than original PSDD.

None of these algorithms were designed to run on the GPU. Bleiweiss [8] described the implementation of parallel path-finding on the CUDA architecture. His work is aimed at crowded game scenes, where thousands of paths need to be found. Each path is planned sequentially by one thread. Parallelism is achieved by running these threads on a CUDA device simultaneously. Bleiweiss tested the algorithm on several graphs with different number of edges. With 2150 edges and 115 600 agents (i.e. threads), the GPU implementation was more than fifty times faster.

Edelkamp et al. [9] used the GPU to solve the *Nine Men's Morris* game. A common approach in game theory is to use Depth-First search to explore the game states space. However Depth-First search is inherently sequential. Therefore they used Breath-First search. The main algorithm runs on the Central Processing Unit (CPU) and the GPU is used during each iteration to expand all states at a given depth. This improvement helped them to explore every reachable state in less than four days. For comparison, the corresponding CPU implementation was unable to return a result in five days.

1.3 Thesis Organization

Chapter 2 introduces planning in Artificial Intelligence. It formally defines the planning problem and provides the overview of commonly used planning algorithms. The major part of the section describes two search algorithms that were used in this work - the well-known A* and its modification, Accelerated A*, that is used in the AgentFly for trajectory planning.

Chapter 3 presents the used approach for parallel planning on the CUDA architecture. The first part of the chapter explains a parallel version of A* called *CUDA Hash Distributed A** (CHDA*). Beside the basic concept, it comments on the algorithm pseudocode and discusses its optimality. The second part introduces modifications that are necessary in order to run Accelerated A* on the GPU. The modified algorithm is called *CUDA Parallel Accelerated A** (CPAA*).

Chapter 4 gives the implementation details of selected parts. First, the problem of synchronization between blocks is addressed. Two solutions to this problem are given - a synchronized and non-blocking buffer. Then the representation of a priority queue for suggested algorithms is provided. In the last part of the section, the representation of no-flight zones in the AgentFly is described together with its impact on the parallel version of Accelerated A*.

Chapter 5 presents the experimental evaluation of presented algorithms. CHDA* is compared to the implementation of A* in a two-dimensional grid with two movement models: (i) four-way unit cost (ii) eight-way unit cost. The performance of CPAA* is evaluated on real data from the AgentFly system. This chapter also discusses the results of the performed experiments.

Chapter 6 summarizes all the information presented in this thesis and provides a final evaluation of the achieved results. Furthermore, it suggests a possible extension of this work.

Appendix A covers general-purpose computing on the CUDA architecture. First, a brief history of the CPU and GPU is given. Then NVIDIA's Fermi architecture and the CUDA programming model are described. Both of the most limiting factors of modern GPUs are addressed - memory latency and code divergence.

Chapter 2

Planning in Artificial Intelligence

Many tasks, including trajectory planning, can be described as a *planning problem*. This makes it possible to use one formal framework to solve all kinds of problems. Planning problems, where the environment is *observable*, can be solved by one of the several general-purpose *search algorithms* that were designed for solving such problems.

This chapter describes the planning in Artificial Intelligence. Section 2.1 provides a formal definition of a general planning problem. The basic terminology is explained in Section 2.2. The overview of commonly used algorithms for solving planning problems is given in Section 2.3. Last two sections explain search algorithms that were used in this work. Namely it is A* search in Section 2.4 and Accelerated A* search in Section 2.5.

2.1 Problem Statement

S. Russel et al. [10] formally defined planning problem with the observable environment by five components:

1. The *initial* state of the problem (e.q. the start position for a trajectory planning problem).

2. A description of the possible actions available at a particular state. (e.q. for a two-dimensional grid map: (i) go left (ii) go right (iii) go up (iv) go down)

3. A description of what each action does, formally called a *transition model*. It is a function that generates a new state as a result of applying a particular action at a particular state. States that are reachable by a single action from some state s are known as the *successors* of s. (e.q. for a two-dimensional grid map, the successors of state $(1, 1)$ are $\{(0, 1), (2, 1), (1, 0), (1, 3)\}$).

4. The goal test, which determines whether a given state is a goal state.

5. A path cost function that assigns a numeric cost to each *path*. A path is a sequence of states that are connected by actions. If each action has a given cost, then the total path cost is the sum of the cost of individual actions along the path.

The first three components (i.e. the initial state, actions and the transition model) define *state space* that can be represented as a *directed graph*. States accessible from the initial state are *nodes* of the graph and actions are *links* between these nodes. Thus, a problem can be seen as a *graph search* (or a *tree search* if loops are allowed).

Some literature (including [10, 11]) distinguish these two terms. States are unique elements of a state space. Nodes are elements that a search algorithm operates on. Each

node is associated with exactly one state. One state can be referred by several nodes. This perspective is useful if a state can be visited more than once (loops are allowed), because it emphasizes the fact that each visit is associated with a different path (each node is at a different depth of the search tree).

A *solution* is a sequence of actions that leads from the initial state to a goal state [10]. Each solution is characterized by its path cost. An *optimal solution* is the one with the lowest path cost among all possible solutions.

2.2 Basic Terminology

Open List (called the **frontier** in [10]) is a data structure that stores states which are yet to be expanded by a search algorithm. Common representation for an open list is a *queue* [10]. The order in which new elements are inserted and removed, determines the type of the queue. The three most widely used types are: (i) first-in, first-out (FIFO *queue*) (ii) last-in, first-out (LIFO *queue*) (iii) *priority queue*, where the element with the highest priority is removed first. The FIFO queue is also known as the *stack* [12]. The LIFO queue is sometimes referred to simply as the *queue*.

Closed List (called the **explored set** in [10]) is a data structure that stores states which have been already expanded by a search algorithm. A closed list can be used to prevent an algorithm from repetitively expanding a particular state [10]. It is often represented by a *hash table*[1]. , since the hash table allows a fast look-up operation to check for an identical (or in some applications a similar) state.

Binary heap is a tree structure that is usually stored in an array, where each node has the maximum of two children that have bigger (smaller) key [14]. The root of the tree contains the node with the smallest (biggest) key. It is commonly used data structure for a priority queue.

Completeness is a property of a search algorithm. An algorithm is said to be *complete* if it is guaranteed to find a *solution* if it exists [10].

Optimality is a property of a search algorithm. An algorithm is said to be *optimal* if it is guaranteed to find an *optimal solution* [10].

2.3 Overview of Search Algorithms

As was mentioned at the beginning of this section, search algorithms are designed to find a solution for planning problems that fulfill all requirements stated in Section 2.1. All algorithms follow the same basic structure that is provided in Algorithm 2.1.

Individual algorithms vary in the way they select a state from the open list (line 5). This property is known as a *search strategy* [10]. Based on the search strategy, they can be divided into the two categories: (i) *uninformed* search (ii) *informed* search

[1] A hash table is a data structure that effectively implements a dictionary (i.e. an element can be accessed by its key). [13]

Algorithm 2.1 Basic structure of search algorithms (source: [10])

```
function SEARCH(problem)
    OPEN ← s_init
    CLOSED ← ∅
    while OPEN ≠ ∅ do
        s_c ← REMOVE(OPEN)
        if s_c = s_goal then
            return solution
        else
            INSERT(s_C, CLOSED)
            foreach s_n ∈ SUCCESSORS(s_c) do
                if s_n ∉ CLOSED then
                    INSERTORREPLACEIFBETTER(s_N, OPEN)
    return failure
```

Uninformed search only has the information about a state that is provided in a problem definition [10]. That means they can only generate successors, and decide whether a particular state is a goal state or not. Typical strategies are:

- **Breath-first search** always expands all successors of the current state before selecting another one from the open list. The open list is represented by a FIFO queue. Given a sufficient amount of time, this algorithm is *complete*. If all actions have the same cost, it is also *optimal*.
- **Uniform-cost search** selects and expands a state from the open list that has the lowest path cost from the initial state. Before a new state is added into the open list, it is tested to see whether a similar state is already in it or not. If it is, the state with the smaller path cost is used. The open list is represented by a priority queue. Unlike the Breath-first search, this algorithm is *complete* and *optimal* regardless of the cost function.
- **Depth-first search** always expands the deepest state in the open list. The open list is represented by a modified LIFO queue that adds successors in an opposite order than they are generated. This algorithm, however, is not guaranteed to find an optimal solution. In an extreme case, it may not even find a solution at all. Therefore, it is *not complete*, nor *optimal*.

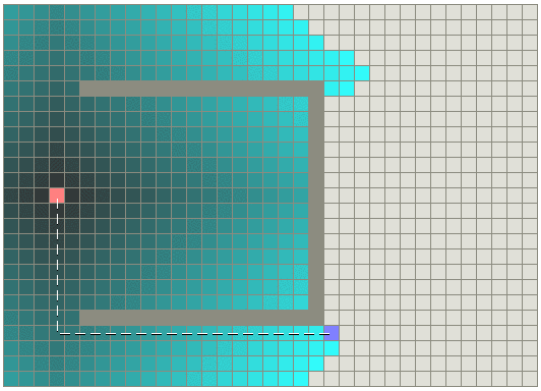
Informed search uses special knowledge about the domain of a problem to select the most promising state [11]. A typical approach is to use this knowledge to compute a *heuristic* (by evaluating a *heuristic* function).The heuristic function is described later, but in general it should estimate the path cost from the current state to the goal state. These type of strategies are known as *Best-first* search [10]. Common examples are:

- **Greedy best first search** uses only the heuristic to expand a state that is closest to the goal state. The open list is represented by a priority queue that always returns and remove a state with the smallest estimated remaining path cost. This algorithm is neither *complete*, nor *optimal*.
- **A* search** combines the heuristic with the real path cost from the initial state. The open list is also represented by a priority queue. If certain requirements on the heuristic function are satisfied, it is *complete* and *optimal*.

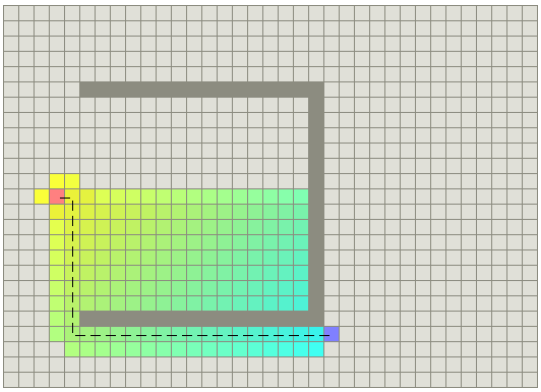
2.4 A* Search Algorithm

A* is one of the most popular search algorithms. It is easy to implement, and yet it provides fairly good performance even on large problems. It is a form of *best-first search* [10].

A* uses an *evaluation function* $f(s) = g(s) + h(s)$ to select a state for expansion, where $g(s)$ is the true path cost from s_{init} to s and $h(s)$ is an estimated cost from s to s_{goal} [11]. During each iteration, it selects a state from the open list that minimizes $f(n)$ (line 5 in Algorithm 2.1). In general, this usually means that less states are expanded and a path is found faster as illustrated in Figure 2.1. Whether this is the case depends on properties of the evaluation function. It should be clear, that the path cost from the initial state is fixed. Therefore, effectiveness of A* relies on the heuristic function.



(a) Breadth-first search



(b) A*

Figure 2.1: States expanded by A* and Breadth-first search. (source: [15])

2.4.1 Properties of the Heuristic Function

The heuristic has a significant effect on properties of A*. It can affect the effectiveness and the optimality as well. Let $h^*(s)$ be a true cost from s to s_{goal}. Then according to the value of the $h(n)$, the *heuristic* function can be divided into the three categories [15]: (i) $h(s) \ll h^*(s)$ (ii) $h(s) \gg h^*(s)$ (iii) $h(s) \approx h^*(s)$. In the first case, only the $g(s)$ is considered and A* turns into the *Uniform-cost search*. The second case is the exact opposite. The heuristic is much bigger than the true path cost. Therefore, the $g(s)$ will have a little influence (if any at all) on the evaluation function and A* reduces to *Greedy best-first search*. Neither the first case nor the second leads to an improvement. Hence, the heuristic should be somewhere around the true path cost.

Let $c(s, s') \geq \varepsilon > 0$ be the true path cost between states s and s', where ϵ is a reasonably small value[2]. Then the heuristic function is said to be [10]:

- *admissible* if $h(s) \leq h^*(s)$
- *consistent* if $h(s) < c(s, s') + h(s')$

The first property means that the path cost will be never overestimated, i.e. the total path cost will be always bigger than the $f(s)$. This is a sufficient condition for the *optimality* if the *closed list* is not used or it is allowed to re-open already closed states. Otherwise it is possible that an intermediate state on the desired path will be expanded and closed with worse $g(s)$ and it will not be re-opened once it is reached again with a better one.

The condition of the second property is nothing else than *triangle inequality*[3] from mathematics, where vertices of the triangle are s, s', and s_{goal} [10]. In other words, the evaluation function cannot get lower when moving to the successor state. Each function that is consistent is also admissible. The admissible heuristic, however, does not have to be consistent. If the heuristic is consistent then A* is optimal. P. E. Hart el al. [11] also proved that with the consistent heuristic, A* will never expand more states than any other search algorithm. The *consistency* is sometimes also called a *monotonicity*.

2.4.2 Selection of the Heuristic Function

Selecting the appropriate *heuristic* function can be tricky. It strongly depends on the problem. The consistency does not mean that the heuristic is good. For example, $h(s) = 0$ is consistent, but it does not add any information beyond the problem definition.

Let h_1 and h_2 be two consistent heuristics. If $h_1(s) > h_2(s)$ for all $s \in \mathbb{V}$, than h_1 dominates h_2 [10]. The heuristic that dominates another one will never expand more states. The condition for the admissible (and therefore consistent as well) heuristic gives the upper bound. Therefore, the ideal heuristic should be as close to h^* as possible.

One way of obtaining a good heuristic is to solve a *relax problem* [10]. A relax problem is a simplified original problem. This is achieved by omitting some constraints. Such a problem is usually easier and faster to solve. Another approach is to have a database of *subproblems*. The heuristic then can be selected as the cost of a subproblem in the database.

When designing a heuristic, time complexity should be considered as well. If it takes too much time to compute the heuristic, the algorithm might give a worse performance than some uninformed search. Sometimes, it might be even feasible to choose an inconsistent heuristic and obtain a sub-optimal solution that can be sufficient for the given problem.

2.4.3 Heuristics for Two-Dimensional Grids

When planning is done on a two-dimmensional grid, these are the most often used heuristics:

- *Manhattan distance*: $h(s) = |x_s - x_{s_{goal}}| + |y_s - y_{s_{goal}}|$
- *Diagonal distance*: $h(s) = \max(|x_s - x_{s_{goal}}|, |y_s - y_{s_{goal}}|)$
- *Euclidean distance*: $h(s) = \sqrt{(x_s - x_{s_{goal}})^2 + (y_s - y_{s_{goal}})^2}$

[2] The meaning of ε is to ensure that each action will increase the path cost. Otherwise, the search might get caught in a loop between two states with a cost that converges to zero

[3] *Triangle inequality* states that the length of any side of the triangle cannot be greater than the sum of the other two.

2.5 Accelerated A* Search Algorithm

Accelerated A* (AA*) was introduced by Šišlák et al. [16]. It was designed to enable a trajectory planning in large-scale environments. It extends A* algorithm by adding an *adaptive sampling* [17]. This brings significant speed-up while preserving a *search precision*.

The distance of a newly generated state from the parent state depends on a *planning step*. The planning step length depends on the distance to the nearest *excluded area* (obstacle or restricted area). If the state is far away from any excluded area, the sampling is sparse and fewer states are generated. The closer planning gets to some excluded area, the smaller step is used so AA* will not miss any solution. The planning step cannot get smaller than a predefined minimum value, known as a *search precision*, otherwise the step will infinitely reduce towards zero for states that are very close to some excluded area [16]. Therefore, AA* will not miss any gap between excluded areas that is larger than the search precision. Figure 2.2 illustrates this concept in an example in two-dimensional space.

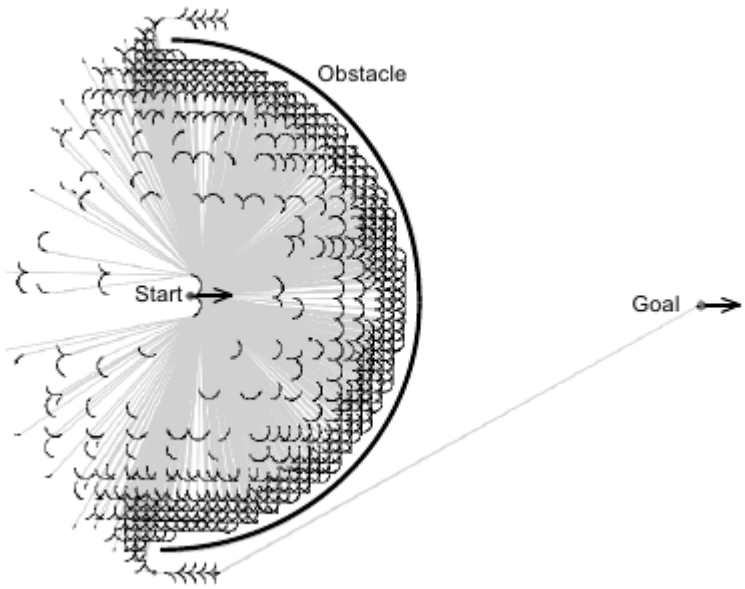


Figure 2.2: The adaptive sampling example in two-dimensional space (source: [18])

The method of varying sampling steps has two drawbacks. First, it can result in generating several states that are almost similar, but that will not pass the ordinary equality test [17]. This will dramatically increase the number of states and reduce the algorithm performance. Hence AA* uses a similarity test instead when operating on open and closed lists. Two states are similar if their euclidean distance and difference of their directions are less than the thresholds derived from the planning step. Secondly, the path can be curved more than it is necessary. In order to avoid it, each state is *smoothed* before it is added into the open list. Briefly, smoothing is a process of finding a new parent that gives a better path cost from the initial state when used instead of the current parent.

Since Accelerated A* is based on A* search algorithm, it also relies on a good heuristic function. AA* uses the length of the actual path that respects all physical constraints on a movement, but that does not consider any excluded zones.

2.5.1 Algorithm Pseudocode Description

The Accelerated A* pseudocode is provided in Algorithm 2.2. The function expects a *start* and a *goal* planning configuration on the input and returns a flight plan (empty if

the search failed). The planning configuration consists of a position and a direction. The flight plan is denoted as fp, the empty flight plan as $\langle\rangle$.

First, it is checked to see whether the *start* and the *goal* are valid (line 2). The configuration is not valid if it is inside some excluded area. Then an initial planning state is created and added into the open list.The planning state consists of a planning configuration, a sampling level, a flight plan history, an *usable* flag that indicates if the flight plan used for the heuristic is valid, an evaluating function cost and a reference to the previous planning state. The closed list is initialized to an empty set (lines 4 - 7).

During each iteration (lines 8 - 26), the best state is removed from the open list and inserted to the closed list (lines 9 and 10). If the *usable* flag is set (line 11), then a goal state is prepared and smoothed. Then a complete flight plan is reconstructed and returned (lines 12 - 15). Otherwise all possible flight plans from this state are generated (lines 16 - 26).

For each flight plan, an end planning configuration is obtained and a planning step is detected (lines 17 and 18). Then it is checked to see if the configuration is already in the closed list (line 19) and if the flight plan is *not* valid (line 21). The flight plan is valid if it does not intersect with any excluded area. If either one of these conditions is *true*, then the rest of the loop is skipped. Otherwise a new state is created, smoothed and inserted into the open list (lines 23 - 26).

If the open list is empty, a trajectory was not found and the empty flight plan is returned (line 27).

Algorithm 2.2 Accelerated A* algorithm pseudocode (source: [17])

1: **function** AASearch(pc_S, pc_G)
2: **if** $\neg$IsValid(pc_S)**or** $\neg$isValid(pc_G) **then**
3: **return** $\langle\rangle$
4: $fp^{end} \leftarrow$ Connect(pc_S, pc_G)
5: $s_S \leftarrow \langle pc_S, \text{DetectSamplingStep}(pc_S), \langle\rangle, \text{IsValid}(fp^{end}), 0, \text{Cost}(fp^{end}), -\rangle$
6: $OPEN \leftarrow s_S$
7: $CLOSED \leftarrow \emptyset$
8: **while** $OPEN \neq \emptyset$ **do**
9: $s_C \leftarrow$ RemoveTheBest($OPEN$)
10: Insert($s_C, CLOSED$)
11: **if** u^{s_C} **then**
12: $s_G \leftarrow \langle pc_G, -, \text{Connect}(pc^{s_C}, pc_G), true, g^{s_C} + h^{s_C}, 0, s_C\rangle$
13: $s_G \leftarrow$ SmoothPath(s_G, pc_S)
14: $fp^{result} \leftarrow$ ReconstructPath(s_G)
15: **return** fp^{result}
16: **foreach** $fp_i \in$ Expand(s_C) **do**
17: $pc_N \leftarrow$ EndConfiguration(fp_i)
18: $\xi_N \leftarrow$ DetectSamplingStep(pc_M)
19: **if** Contains($pc_N, CLOSED, \xi_N$) **then**
20: **continue**
21: **if** $\neg$IsValid(fp_i) **then**
22: **continue**
23: $fp^{end} \leftarrow$ Connect(pc_N, pc_G)
24: $s_N \leftarrow \langle pc_N, \xi_N, fp_i, \text{IsValid}(fp^{end}), g^{s_C} + \text{Cost}(fp_i), \text{Cost}(fp^{end}), s_C\rangle$
25: $s_N \leftarrow$ SmoothPath(s_N, pc_S)
26: InsertOrReplaceIfBetter($s_N, OPEN$)
27: **return** $\langle\rangle$

Chapter 3

Parallel Planning on CUDA Architecture

This chapter presents two parallel algorithms. Section 3.1 explains a parallel version of A* for the CUDA architecture and discusss its properties. This algorithm is called *CUDA Hash Distributed A** (CHDA*). Section 3.2 introduces modifications that are necessary for CHDA* to be usable for *Accelerated A**. The modified parallel algorithm is named *CUDA Parallel Accelerated A** (CPAA*).

3.1 CUDA Hash Distributed A*

CUDA reveals computational power through a large number of threads. In order to take advantage of it, it is necessary to have work for these threads. A simple implementation lets threads operate on a shared open and closed list. Synchronization has to be used to ensure consistency of these structures. Both the open and closed list are accessed quite often. Thus, this approach yields a lot of overheads. The situation is even worse on the CUDA architecture. Recall that threads in a warp execute the same instruction (unless they are divergent). Thus, they access these lists at the same time, which causes their execution to be serialized and consequently it reduces the instruction throughput by a factor of 32.

Another problem is synchronization itself. CUDA supports synchronization only within a single block. The block is assigned on one SM, ergo only a fraction of resources would be used. To fully utilize the GPU, it is necessary to have multiple blocks and to establish communication between them. This is possible due to the support of atomic operations on global memory. Although it is impossible to avoid it, it should be used as little as possible, which is not the case of the open and closed list.

Another approach is to use one's own open and closed list for each thread. With careful design, the access of threads in a warp can be coalesced as long as they request elements with the same relative offset. In this case, synchronization is needed only when states are transferred between blocks. In general, this should occur less often. Evett et al. [2] introduced an algorithm that uses synchronous communication to distribute newly expanded states according to their *hash* value. Kishimoto et al. [4] improved it to use asynchronous communication. The suggested algorithm is a modification of this algorithm, which is known as *Hash Distributed A** (HDA*) [4].

The first part of this section introduces a CHDA* concept. It is followed by detailed algorithm describtion. The last subsection discusses suboptimality of the algorithm and suggests a solution to this problem.

3.1.1 Concept

Work is divided between n threads that are divided into b blocks. Each block contains t threads. All blocks must be guaranteed to run at the same time. Therefore, the size of b and t is limited from above by the number of blocks and threads that can reside on one SM as well as by usage of resources (i.e. registers and shared memory). To avoid wasting available resources, the number of threads in a block should be a multiple of 32. Threads are uniquely identified by their *block id* (bid) and *thread id* (tid) within a block. This identifier is denoted as (bid, tid).

All threads are even. There is no master thread that would control the others. Communication between threads is asynchronous. Each thread has a buffer, where other threads can put states for it. The buffer must support two operations: (i) insertion of new states by all threads (ii) removal of states by the thread it belongs to. Consistency must be guaranteed at all times. Several threads might try to insert new states while the owner tries to remove one. Since buffers are located in global memory, access takes a long time. Ideally, requests should be coalesced. This is impossible to achieve for insertion, but removal of states occurs at the same time within a warp. Therefore, the buffers should be grouped by blocks, which also makes them easily accessible using the (bid, tid) identifier.

Each thread is responsible for expanding states from its own open list. When a new state is generated, a *hash function* is evaluated to obtain a *hash* for a given state. The hash function may be any function that returns an integer with one limitation. It must be deterministic (i.e. it must return the same value for equal states). The hash is used to select a thread the state belongs to by using Equation 3.1. The state is then inserted into the buffer of the desired thread. Figure 3.1 illustrates a distribution of states on a two-dimensional grid map.

$$bid = \frac{(hash \mod n)}{t} \quad , \qquad tid = (hash \mod t) \tag{3.1}$$

Once all successors are generated, the thread checks its buffer for new states. If there is one, the thread takes it and processes it. First, it checks whether the state is already in the closed list. This test is different from the one used in the original A*. In general, due to the nature of all parallel search algorithms, a state may already be closed with a worse path cost. Therefore, it is necessary to allow re-opening of already visited states. If the closed list contains an equal state with better $f(s)$, the new state is discarded. Otherwise it is inserted into the open list (or in the case that the open list already contains the equal state, it is either discarded or the original value is updated and the open list is re-sorted, depending on whether the new state has a better path cost or not).

(1,1)	(1,2)	(2,1)	(2,2)	(1,1)
(1,2)	(2,1)	(2,2)	(1,1)	(1,2)

$$hash = y \cdot width + x$$

Figure 3.1: Example of distribution of states for two blocks with two threads per block in a two dimensional grid map.

The number of states that are removed from the buffer during each iteration can impact performance. If only one state is removed, it is possible that a state with much better $f(s)$ will get stacked in the buffer for a long time (especially if the buffer is implemented as LIFO) and the thread will keep expanding states that do not lead to a solution. On the other hand,

if an algorithm requires that the buffer is empty before moving further, the thread might end up removing states only if other threads keep inserting new ones.

Once a goal state is reached, the other threads are notified through a common variable for all threads (a shared variable) in global memory and the solution is saved. The same applies for a situation when the thread has nothing to do (i.e. the open list is empty). In such a case, the thread notifies others and waits until all threads empty their open list or the goal state is reached. While waiting, it must check its buffer for incoming states. If it receives a new state, the other threads must be notified again that it has some states to process (i.e. the open list is not empty).

The termination is possible only under two circumstances: (i) the goal state has been reached (ii) all threads' open lists and buffers are empty. In the first case, a path has been found. In the second case, a path does not exist. It is especially important to check that there are no states on the way, otherwise the algorithm might end earlier and miss a solution.

Algorithm 3.1 CUDA Hash Distributed A* pseudocode

Require: $finished = false$ **and** $emptyCounter = 0$
Ensure: *solution* contains a solution to the problem if it exists.

```
 1: function CHDASearch(problem, buffer, finished, emptyCounter, solution)
 2:     emptyFlag ← false
 3:     OPEN ← ∅
 4:     CLOSED ← ∅
 5:     (bid_{s_init}, tid_{s_init}) ← GetSectorId(s_init)
 6:     if (tid_{s_init}, tid_{s_init}) = (bid, tid) then
 7:         InsertOrReplaceIfBetter(s_init, OPEN)
 8:     while ¬finished and (emptyCounter < n or ¬IsEmpty(buffer)) do
 9:         if ¬IsEmpty(OPEN) then
10:             s_C ← RemoveTheBest(OPEN)
11:             Insert(s_C, CLOSED)
12:             if s_C = s_goal then
13:                 solution ← ReconstructSolution(s_C)
14:                 finished ← true
15:             else
16:                 foreach s_N ∈ Successors(s_C) do
17:                     (bid_{s_N}, tid_{s_N}) ← GetSectorId(s_N)
18:                     Push(buffer_{(bid_{s_N}, tid_{s_N})}, s_N)
19:         while HasNext(buffer_{(bid,tid)}) do
20:             s_C ← Pull(buffer_{(bid,tid)})
21:             if ContainsBetter(s_C, CLOSED) then
22:                 continue
23:             else
24:                 InsertOrReplaceIfBetter(s_C, OPEN)
25:         if IsEmpty(OPEN) then
26:             if ¬emptyFlag then
27:                 AtomicInc(emptyCounter)
28:                 emptyFlag ← true
29:         else
30:             if emptyFlag then
31:                 AtomicDec(emptyCounter)
32:                 emptyFlag ← false
```

3.1.2 Algorithm

A pseudocode for a CHDA* kernel function is provided in Algorithm 3.1. Besides a problem description and buffers, it expects three other arguments that must be placed in global memory and shared by all threads: (i) a `finished` determines whether a solution has been found yet (ii) an `emptyCounter` stores how many threads have an empty open list (iii) a `solution` is used to return a found solution (if it exists).

First, an *empty flag*, open and closed list are initialized (lines 2 - 4). The empty flag is used to determine whether a thread has notified others that its open list is empty. The *initial* state is inserted only to the open list of a thread that it belongs to.

The main loop continues until either a solution is found or all states have been expanded (lines 8 - 32). The iteration can be divided into three stages:

- **Stage 1**: expansion of a state from the open list (lines 9 - 18)
- **Stage 2**: processing of states from the buffer (lines 19 - 24)
- **Stage 3**: notification of other threads if the open list is empty or not (lines 25 - 32)

In the first stage, there is check done to see whether the open list has some more states (line 9). If it does not, the expansion of the best state from the open list is skipped. Otherwise, the best state is removed and closed (line 10 and 11). It is then compared with the goal state (line 12). If they are equal, a solution is reconstructed and $finished$ is set to `true` (lines 13 and 14). Otherwise, successors are generated and written on an appropriate buffer (lines 16 - 18).

As long as there is only one goal state, synchronization is not required while writing a solution, since only one thread will ever write it due to the deterministic hash function. However, if there are more goal states, it is possible that individual goal states will be assigned to different threads. In such a case, a proper synchronization must be used to guarantee consistency.

In the second stage, each thread processes states from its buffer. This pseudocode expands all states in the buffer, but as was suggested in Section 3.1.1, it might not always be the best solution. Once a new state is pulled from the buffer (line 20), it is checked to see if the closed list contains an equal state with better path cost (line 21). If it does not, the new state is inserted into the open list (line 24).

The last stage checks if the thread has a state in the open list. If it does not, the empty counter is incremented if it was not done in previous iterations (lines 26 - 28). Similarly, if the thread has some states to process and other threads were notified, the empty counter is decremented (lines 30 - 32). To ensure consistency, the counter is incremented/decremented by using an *atomic increment/decrement* operation.

3.1.3 Optimality

It is easy to observe that the algorithm *is not optimal*, even if $h(s)$ is *admissible* and *consistent*. A* optimality is guaranteed (under the condition that $h(s)$ is admissible and consistent), because an expanded state has the lowest $f(s)$ (it is the best state) among all states that have not been closed yet. In the suggested algorithm, n threads expand states from their own open list. Therefore, besides the best state, $n-1$ other states are expanded as well. As a matter of fact, the best state might not be expanded at all during the iteration, because it might be in some thread's buffer. Since some threads might have less work to do, it is possible that a suboptimal solution will be found first. Hence, without any modification, the algorithm is not guaranteed to find the best solution.

To ensure optimality, the search must continue even after the solution is found. The modified algorithm can be divided into two phases:

- **Phase 1**: find any solution
- **Phase 2**: continue search until it is proven that there are no better solutions

Phase 1 consists merely of the algorithm suggested in Section 3.1.1. When a solution is found, the cost of the solution is stored in a shared variable in global memory.

Phase 2 adds few modifications. First, states with the $f(s)$ worse than the best solution are removed from the open list (the open list is pruned). Secondly, only states that have the $f(s)$ smaller than the best solution are added into the open list. The rest is discarded. Whenever a new solution is found, its cost is recorded and the other threads are notified of this change so they can prune their open list. The search continues as long as some thread has states in the open list or in the buffer.

Depending on the data structure used for an open list, pruning can be an expensive operation. The most commonly used implementation for the open list (and the one used in this work, see Section 4.2) is a *binary heap*. Removing all nodes with a key value bigger than a certain threshold would require traversing the whole tree and removing nodes that do not satisfy this condition. Generally, the cost for such an operation is $O(n \cdot log(n))$, which is too high, especially on CUDA architecture due to high memory latency and code divergence that will undoubtedly occur.

In the case that the cost for pruning is too high, it might be better to use a slightly different approach. Instead of insisting on removing all states that are worse than the best solution, only a portion of those states, which can be removed without adding too much overhead, is removed. Furthermore, every operation on the open list (i.e. remove and return smaller, insert, update) is given the best path cost as a *limit*. If a state with the $f(s)$ worse than the limit is accessed, it is immediately removed if the cost for such an operation is small. A negative of this approach is that operations on the open list are generally slower, because the open list contains more states. Nonetheless, in some situations, this might give a better result.

3.2 CUDA Parallel Accelerated A*

*Accelerated A** extends A* algorithm. Basically, the algorithm suggested in Algorithm 3.1 can also be used for AA*. Nevertheless, without any modification, a state distribution would lead into expanding a lot of similar states. To avoid it, states that are close to each other should be assigned to the same thread. A solution to this problem is described in Section 3.2.1.

Unfortunately, this requirement introduces a new problem - a *load balancing*. CHDA* relies on the fact that each successor will be, at least with very high probability, assigned to a different thread. As a result of that, most of the threads have some work to do. This, however, contradicts the condition that similar states should be processed by the same thread. This would not be a problem if all threads had some states to expand from the beginning. However, since only one thread can own the initial state, it can take a long time (if at all) before all threads are occupied. Fortunately, unlike A*, AA* is computationally demanding, especially during the smoothing. Hence, it can be solved by making two modifications: (i) let several threads work on one state (ii) add another stage that would allow threads to work on a state that does not belong to them. These two improvements are described in Section 3.2.2 and Section 3.2.3.

3.2.1 States Distribution

To avoid extensive overhead as a result of duplicate states, similar states must be assigned to the same thread as often as possible. Thus, states are distributed according to their real position in a search space rather than by a hash function. Assume without loss of generality that a search is done in a two-dimensional setup[1]. Then search space can be divided exactly into a two-dimensional grid of n sectors, with b sectors in the x-dimension and t sectors in the y-dimension. Each thread is responsible for states in one sector. As a consequence, duplicate states are undetected only on the border of two sectors. Figure 3.2 illustrates this concept.

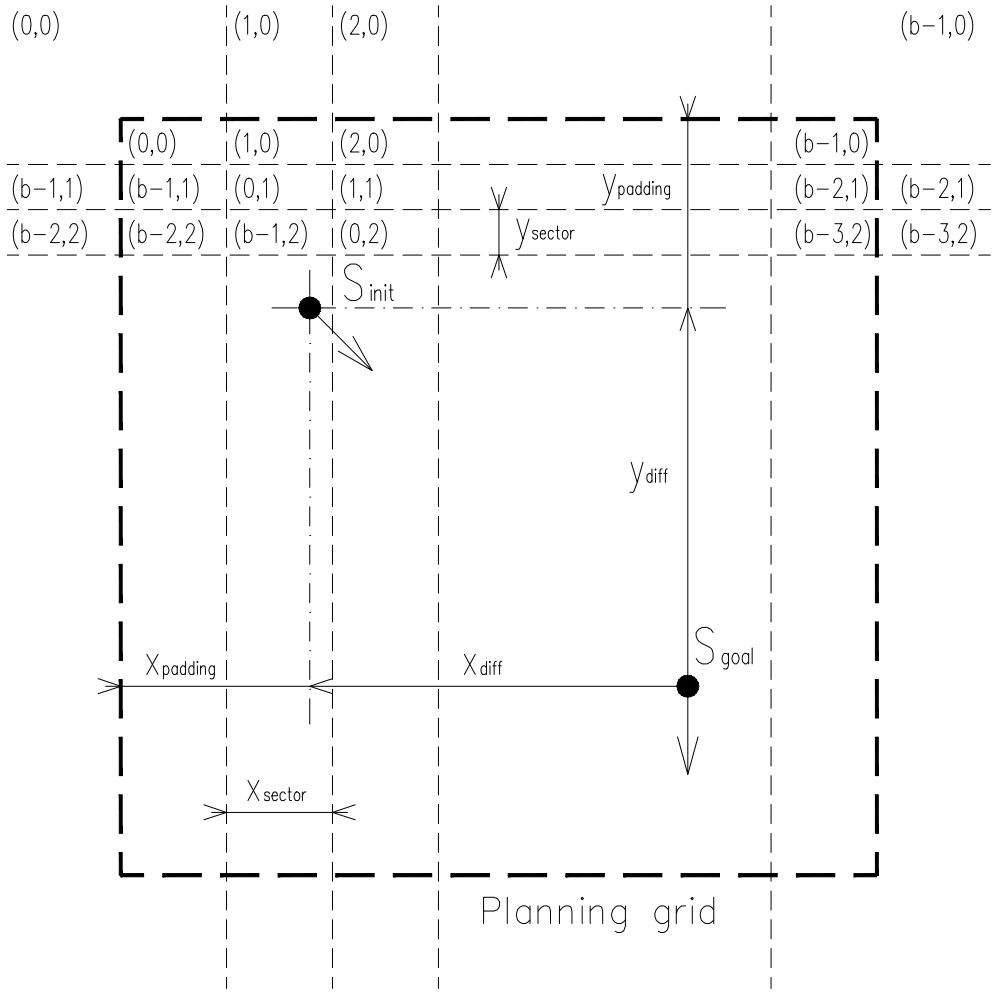


Figure 3.2: Partitioning of search space into the sectors in a two-dimensional setup.

Remember that AA* plans a path between two states: s_{init} and s_{goal}. Intuitively, the sought path should be as straight as possible. Therefore, most of the expanded states should lie in (or somewhere around) a space between s_{init} and s_{goal}. Let's denoted this area as a *planning grid*. Let x_{diff} be the difference between x-axis coordinates of the s_{init} and s_{goal} and y_{diff} be the difference between y-axis coordinates. Let $x_{padding}$ be the distance in the x-axis dimension from a rectangle formed by the s_{init} and s_{goal} where expanded states are considered to occur. Let $y_{padding}$ be the same distance as $x_{padding}$ only in the y-axis dimension. Then planning grid can be expressed as a rectangle by its upper-left corner v_{ul} and bottom-right corner v_{br} (3.2). The dimensions of the grid are $(width, height)$ (3.3).

[1] A three-dimensional setup can be transformed to a two-dimensional for a need of states distribution by omitting one of the dimensions.

$$
\begin{aligned}
v_{ul} &= (\min(x_{s_{init}}, x_{s_{goal}}) - x_{padding}, \min(y_{s_{init}}, y_{s_{goal}}) - y_{padding}) \\
v_{br} &= (\max(x_{s_{init}}, x_{s_{goal}}) + x_{padding}, \max(y_{s_{init}}, y_{s_{goal}}) + y_{padding})
\end{aligned} \tag{3.2}
$$

$$
width = x_{diff} + 2 \cdot x_{padding} \quad , \qquad height = y_{diff} + 2 \cdot y_{padding} \tag{3.3}
$$

The planning grid is divided into b sections in the x-dimension and t sections in the y-dimension. This gives *one-to-one mapping* between sectors of the search space and sections of the planning grid. Therefore, each sector is assigned exactly one section. The dimensions of each sector are (x_{sector}, y_{sector}) (3.4). This also improves hashing, because a position of each state on the planning grid can be first transformed to an interval from $(0, 0)$ to (x_{sector}, y_{sector}) and then used to compute a hash. This way, all hashes are reachable for a given thread and space in memory is not wasted.

$$
x_{sector} = \frac{width}{b} \quad , \qquad y_{sector} = \frac{height}{t} \tag{3.4}
$$

The described partitioning has one hiccup. It supposes that all expanded states will be inside the planning grid. This, however, cannot be guaranteed and some states (or if the planning grid is chosen wrongly most of them) can be outside the grid. A fast and an easy solution is to extend the corner sectors to infinity. Since occurrence of these states is still assumed to be rare, they can be assigned some default hash value. That way, a position transformation can still be used to improve hashing for states inside the planning grid.

3.2.2 Threads Cooperation

Accelerated A* has some parts that, with the current approach, do not comply with the CUDA architecture. Talk is about an intersection test and planning step test (called a *point-level test*[2]).

To determine whether a path intersects, it must be tested against all excluded areas. This requires loading a large amount of data from memory. There would not be a problem if the path did not intersect for all threads, because they would request the same address in the memory. A problem occurs when some paths intersect. Given the distribution of states, the path of individual threads is likely to intersect with a different excluded area. Each time some paths intersect, the warp becomes more divergent. In the worst possible case, a test may fail for all threads except one on the first excluded area. In such a situation, these threads are forced to wait until the path of the remaining one either passes or fails the test and the throughput is dramatically reduced.

This problem becomes even more serious during the point-level test. The point-level is detected by using a *binary search* [19]. First, the middle planning step is tested. If it fails, the distance is decreased by one half. Otherwise it is increased by one half. A point is then tested against this newly obtained distance. This continues until a correct point level is found. To determine that a planning step is valid, it must be tested against all excluded areas. In this case, however, individual threads might be testing different steps, hence requesting completely different data. Therefore, besides the code divergence, a problem with scattered memory access arises.

[2]It is called a point-level test, because the planning step is determined by testing the state position (point) against *inflated excluded areas*. The excluded areas are inflated by the tested distance. It is then tested whether the point is inside any inflated excluded area or not. If it is, an excluded area is closer than the tested distance.

Both of these problems can be solved by having all threads in a warp to collaborate on solving tasks that exhibit by natural data parallelism such as intersection and a point-level test. Each thread will perform the test on a portion of excluded areas. This has positive impact on the performance in several ways: (i) the test can stop as soon as one thread fails (ii) memory access can be coalesced, because an interleaved memory pattern can be used and successive threads can test successive excluded areas (iii) a warp can be fully utilized even if only one thread has states in the open list (i.e. it improves the load-balancing).

3.2.3 Task Scheduler

Thread cooperation solves the load-balancing only partially. It merely optimizes the execution within a warp. With a simple modification, the whole block can collaborate. Nevertheless, this would still not solve the occupancy issue of other blocks. A simple analysis of Algorithm 2.2 reveals an interesting property. The iteration of AA* can be divided into three steps: (i) expand a state from the open list and generate its successors (ii) smooth the path of each successor and detect its point-level (iii) try to add it into the open list. The second step is the most expensive one. AA* tries to avoid it by detecting duplicate states of already closed states in the first step. However, this is not possible for CPAA*. Because the algorithm must allow re-opening of already closed states, smoothing must be computed for every expanded state, otherwise it is not possible to decide whether a closed state should be re-opened or not.

Observe that the second step does not require a write access to the open list, nor to the closed list. All it needs is a newly generated state and read access to states that lies on the path. Therefore, there is no reason to insist that the second step has to be executed by a thread (or if thread cooperation is used a warp/block) that the state belongs to. Instead of that, a smoothing and point level detection can be done by a thread (warp/block) that has available resources (i.e. has no states to process). For the rest of this section, it is assumed that the cooperation of all threads within a block is used.

This effect can be achieved by adding a *task scheduler*. The task scheduler is an ordinary concurrent buffer. The only difference between the one used for distribution of states is that it is shared by all blocks. Every block has to have write and read access to it. When a new state is generated, it is added to the task scheduler. After all successors are generated (or immediately if none of the threads in a block has a state to process), one or more states are pulled from the task scheduler. Threads in a block then collectively smooth the path and detect the point level of individual states. Then a thread that state belongs to is determined and the state is added into the appropriate buffer. After that, the algorithm continues as described in Section 3.1.1 - each thread checks its buffer for incoming states and processes them. This whole concept is shown in Figure 3.3.

It can be further improved to provide even better control over the load-balancing. Assume that only a few blocks have states for expansion. Once a block finishes generation of new states, it tries to pull a state from the task scheduler. But since there are plenty of blocks with available resources, this is not necessary at all. As a matter of fact it is even a counter productive, because other blocks will have nothing to do. A better solution is to skip this step and rather process received states from the buffer and expand the next state from the open list. That way, most of the blocks can stay occupied until they have their own states to expand.

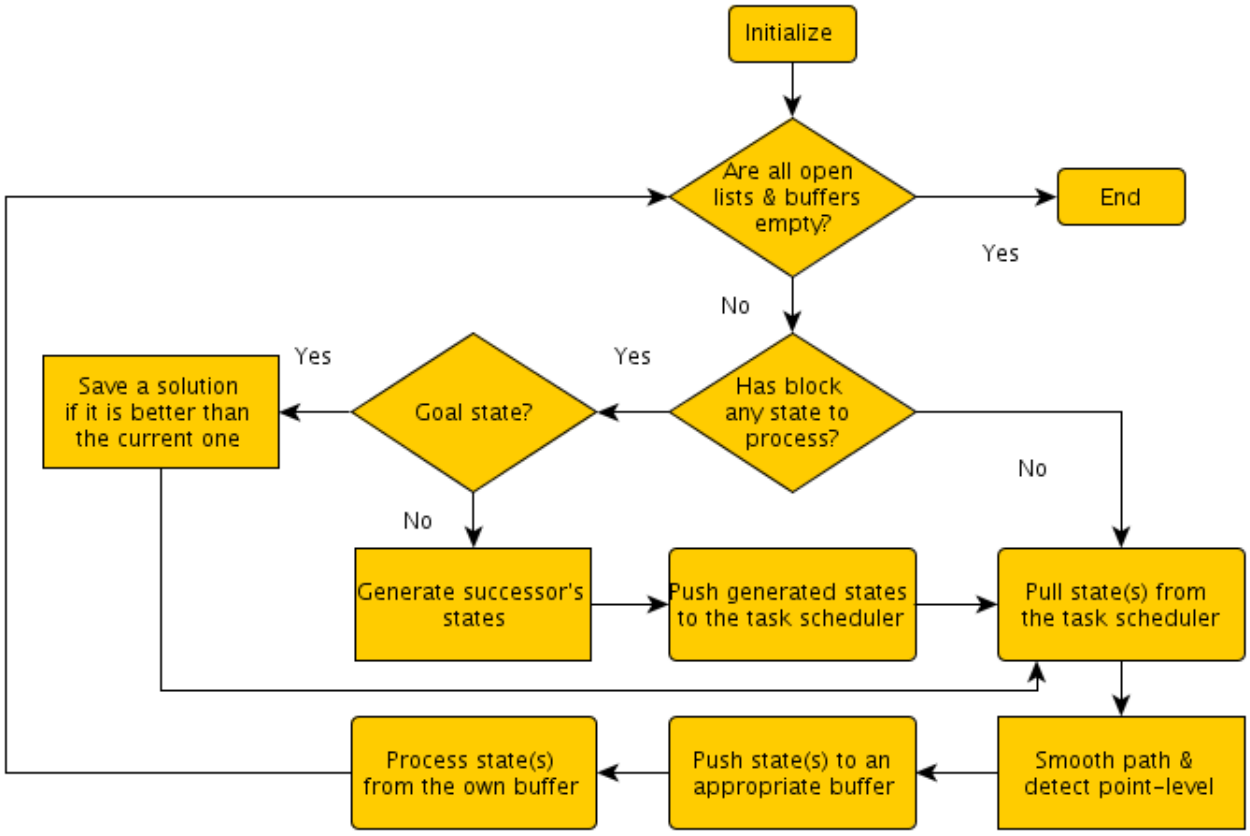


Figure 3.3: A flowchart of CUDA Parallel Accelerated A* algorithm.

3.2.4 Algorithm

CPAA* algorithm is a combination of AA* algorithm, CHDA* algorithm that was described in Section 3.1, and modifications introduced in this section. It can be divided into four stages:

- **Stage 1**: expansion of a state from the open list, successors are added to the task scheduler
- **Stage 2**: processing states from the scheduler (i.e. smoothing and point-level detection)
- **Stage 3**: processing states from the buffer (i.e. either add them into the open list or discard them)
- **Stage 4**: notification of other threads whether the open list is empty or not

For further information, refer to a pseudocode of the kernel function in Algorithm B.1. Note that the given algorithm is *not optimal*. The same as for CHDA*, it is necessary to continue search even after a valid path is found to ensure that the optimal path is returned. See Section 3.1.3 for more details.

3.2.5 Rejected Approaches

This section discusses two approaches that were originally considered and tested, but that proved to be wrong.

Distribution of No-Flight Zones

Frequent operation in Accelerated A* is a test, whether a point is inside any (inflated) zone. Logically, the point can be only inside those zones that extend to the sector the point belongs to. Therefore, all zones can be divided between sectors before the search starts and then the point can be tested against zones inside its sector. Furthermore, zones can be re-organized in memory in such a way that consecutive zones belong to the same sector.

Unfortunately, this "improvement" had almost no effect on performance, but the time needed for the distribution of zones was long. Therefore it was abandoned.

Larger Number of Sectors

Originally, there was an attempt to solve the problem with load balancing by adding more sectors, while the number of threads and the size of the planning grid remained the same. Each thread was responsible for more sectors. Each sector had its own buffer and closed list, but the open list was shared by all other sectors that belonged to the same thread.

This helped a little bit when the number of sectors per thread was four, but it was still not enough to occupy all blocks, especially when the smallest planning step was used. When the number of sectors was further increased, the performance reduced [3]. Therefore, this approach was also rejected.

[3] This is the problem with the distribution of states in general. See Section 5.2 for more details.

Chapter 4

Implementation Details

This chapter provides a detailed description of selected problems. Section 4.1 discusses the problem of synchronization between blocks. First it explains concurrence support on the CUDA architecture. Then the in-depth description of a synchronized and non-blocking buffer follows. Section 4.2 introduces a representation of binary heap on the CUDA architecture. Section 4.3 describes representation of excluded areas and differences between the generation of inflated edges on the CPU and GPU.

4.1 Synchronization

Both algorithms, CHDA* and CPAA*, relay on exchange of states between blocks. CUDA only supports synchronization between blocks by separated kernel launches. Neither CHDA* nor CPAA* can use this technique, because it would be necessary to launch a new kernel after each iteration. On top of that it is not known in advance how many iterations would be needed. Fortunately, CUDA supports atomic functions that perform a *read-modify-write* operation in a single transaction. Therefore, it is possible to enforce block synchronization by implementing a buffer in global memory that supports concurrent read and write requests. According to the way the simultaneous requests are handled, concurrent buffers can be divided into two categories: (i) synchronized buffer (ii) non-blocking buffer.

Furthermore, two challenges must be overcome while sharing data between blocks. The first one is the result of a long memory latency and optimizations done by a compiler. The compiler might decide to reorder some memory requests. This can result in a situation when a control variable is set sooner than expected, before all changes to global memory are visible by other blocks. As a consequence, a different block can read inconsistent data. This can be avoided by using a *memory barrier*.

The second one is based on the fact that on Fermi, each SM contains an L1 cache. The contents of individual caches is not synchronized. Thus, the L1 cache can have obsolete values. It is up to a programmer to make sure that correct values are read. This can be achieved by using a *volatile keyword*.

4.1.1 CUDA Concurrence Support

This section describes three components that are needed to implement communication between blocks on the CUDA architecture: (i) atomic functions (ii) memory barrier (iii) volatile qualifier.

Atomic Functions

CUDA supports atomic functions from a compute capability 1.1 [20]. Atomic functions read an address, perform a desired modification and write the result back without other threads making any changes to that address.

A typical example used in books is an increment of a single variable [21]. This operation is denoted as `x++` in `C/C++`. Although it may seem that only a single instruction is needed, it takes three instructions to increase the value of variable x by one: (i) read the value in x (ii) add one to that value (iii) write the result back

Let's assume that initially $x = 3$ and two threads want to increment it. The expected result after performing an increment operation by both threads is $x = 5$. If threads reside on the same SM, it will happen if a warp scheduler issues first all three instructions of one thread and then of the other thread. This is not likely to happen due to the long memory latency and the way the warp scheduler selects instructions for execution. Bigger chances are that both threads will first read the value, then modify it and finally write the result ($x = 4$) back. If threads are in the same warp, a correct result cannot be obtained in principle, because the threads will execute all three instructions exactly at the same time. In such a case, which thread will write the result back is undetermined. If threads are on a different SMs, they are not even controlled by the same warp scheduler and the result may or may not be correct. The only way to ensure the correctness is to use an atomic function (aptly named `atomicInc()`).

The most useful atomic function available in CUDA C is `atomicCAS()` [20]. *CAS* stands for *Compare and Swap*. It takes three arguments: (i) an address in a memory (ii) a comparison value (iii) a new value. It reads the value at the address, stores it in a temporary variable `old`, and compares it with the comparison value. If they are equal, the value at the address is replaced by the new value. The function returns `old`.

Memory Barrier

Very often, a program is written in such a way that some changes in memory are expected to be visible to other threads before moving to another phase in computation. By default, this intention is not known to a compiler. The compiler may reorder instructions to speed up the computation. Therefore, if a result depends on the instructions order, a *memory barrier* must be used. In *CUDA C*, the memory barrier, represented by `__threadfence()` function, stalls a thread until changes, made by the calling thread, in shared memory are visible by all threads in a block and changes in global memory are visible by all threads in the device [20]. Furthermore, it prevents the compiler from moving memory requests, which occur before the barrier, after the barrier and vice versa. Note, however, that requests on the same side of the barrier can still be reordered, even two atomic functions if they operates on different addresses.

To enforce that changes are visible only to threads in a block, the `__threadfence_block()` function can be used. The same result can be achieved by `__syncthreads()` (a barrier synchronization). Unlike the barrier synchronization, the memory barrier does not have to be executed by all threads in a block. The thread can continue as soon as changes are visible.

Volatile Qualifier

Even with the memory barrier, a thread can still get incorrect data due to the inconsistent L1 caches on Fermi. Another problem may occur when working with shared memory within a warp [22]. Since execution of threads in a warp is naturally synchronized, it is

common practice to omit synchronization. However, on Fermi, CUDA compiler may decide to accumulate results in a register instead of writing it directly back to shared memory.

Both of these problems can be solved by declaring a variable as a `volatile` [20]. The compiler assumes that the volatile variable can be changed or used anytime by a different thread. Therefore, the variable is never temporarily stored in a register, neither is it cached. Every reference to that variable results in actual read or write requests.

The semantic is the same as in a standard C. Basically, three cases are possible [23]: (i) `volatile int a` (ii) `volatile int *a` (iii) `int *volatile a`. The first case is straightforward. It means that variable a should be accessed in a volatile context. The remaining two are more interesting. The trick is to realize that the compiler parses it from right to left. Hence, in the second one, the value that a points to is going to be treated as volatile, but not the pointer itself. The last case is an exact opposite. The pointer a is volatile, but the value it points to is not and access to it can be optimized. These two cases can be combined together. Consider the following declaration: `volatile int *volatile a`. In this case, access to both, the pointer and the value it points to, compile to an actual load or store instruction. If an object is declared as volatile, all its members are considered to be volatile. It is even possible to declare only a method of the object as volatile. The result is that the object (i.e. `this` pointer) is treated as volatile during the execution of that function.

4.1.2 Synchronized Buffer

The easiest way to implement a buffer that can be shared by several threads is to allow only one thread to read from or write to it. This can be done using a *mutual exclusion lock* (mutex) [24]. A mutex is an object that allows only one thread at a time to acquire control over it. Once it does, the other threads that wish to use the mutex have to wait until the owner's thread releases it.

In CUDA C, a simple mutex can be implemented using a variable in global memory and an atomic compare and swap instruction. Algorithm 4.1 provides a pseudocode for two functions to acquire and release the control over the mutex. The mutex is used by a thread if its value is *true*. Recall that `atomicCAS()` returns the original value. Therefore, the loop at line 2 will not finish until the control over the mutex is acquired. For `RealeseLock()` to work, *mutex* must be declared as volatile, otherwise line 4 may be optimized out.

Algorithm 4.1 Mutual exclusion lock pseudocode (based on [24])

1: **function** AcquireLock($mutex$)
2: **while** atomicCAS($mutex, false, true$) **do continue**

Require: *mutex* was previously acquired by the calling thread
3: **function** ReleaseLock($mutex$)
4: $mutex \leftarrow false$

The suggested mutex has one pitfall. If more threads in a warp try to acquire the lock, the warp will get caught in an infinite loop. Once one thread in that warp acquires control over the mutex, it will exit the loop. However, as a result of SIMT architecture, it will be unable to proceed, because the other threads will be still trying to acquire the lock. That is impossible since it is already acquired. The owner's thread must first release it. Unfortunately, that is impossible too, because it is waiting for threads that are trying to acquire it. This situation is called a *deadlock*.

A solution to this problem is to not insist on obtaining the control over the mutex, but rather try it once and if it fails, try it again a little bit later. It can be even immediately if

none of the threads in a warp has acquired the lock. The point is to give a thread that did acquire it enough time to release it again. Algorithm 4.2 gives the modified `AcquireLock()` function.

Algorithm 4.2 Try acquire lock pseudocode

```
1: function TryAcquireLock(mutex)
2:     return atomicCAS(mutex, false, true)
```

Even with this modification, a dead lock can still occur. Assume without loss of generality that a block consists of 32 threads and `TryAcquireLock()` is used to obtain the lock. For simplification, let's say that a block tries to acquire the lock if at least one thread in the block tries to acquire it. Imagine a situation when all blocks that reside on one *SM* try to acquire the lock. Once one block succeeds, it continues with the execution of a protected part of the program (called a *critical part*) and releases the lock as soon as it finishes it.

Algorithm 4.3 LIFO synchronized buffer pseudocode

```
 1: function Pull(array, lastIndex, mutex)
 2:     locked ← true
 3:     while locked do
 4:         locked ← TryAcquireLock(mutex)
 5:         if ¬locked then
 6:             if lastIndex > 0 then
 7:                 lastIndex ← lastIndex − 1
 8:                 retVal ← array[lastIndex]
 9:                 __threadfence()
10:             else
11:                 retVal ← null
12:             ReleaseLock()
13:     return retVal
14: function Push(newElement, array, lastIndex, size, mutex)
15:     locked ← true
16:     while locked do
17:         locked ← TryAcquireLock(mutex)
18:         if ¬locked then
19:             if lastIndex < size then
20:                 array[lastIndex] ← newElement
21:                 lastIndex ← lastIndex + 1
22:                 result ← true
23:                 __threadfence()
24:             else
25:                 result ← false
26:             ReleaseLock()
27:     return result
```

CUDA does not use a *preemptive scheduling*[1]. It merely selects a warp that has available instructions for execution. Therefore, it might happen that the scheduler will issue only

[1] A preemptive scheduling allows to temporarily interrupt one task (thread) and give computational resources to another one.

instructions of warps that are trying to acquire the lock and the block that owns it will never get a chance to release it. Unfortunately there is nothing that can be done to prevent it at this moment. Although the chance that this situation will occur is extremely small[2], in general it can happen and it must be considered when designing an algorithm.

The implementation of synchronized buffer does not differ much from the ordinary one. Only a few changes are necessary. The extra work that has to be done is to acquire the lock at the beginning of each method and to release it at the end. Before the lock is released, the memory barrier must be called to ensure that all changes are visible to other threads. At last, all the buffer's variables that can be changed must be declared as volatile.

The synchronized buffer that was tested in this work is LIFO type. Internally, it is implemented by an array that is used to store buffer elements and an index on the first free position after the last element in the buffer. To minimize time that is spent in the critical part, only pointers to objects are stored in the buffer. Since an object's variables may be set by one thread and read by another one, the object itself must be also declared as volatile. Therefore, the buffer array must be declared as `volatile T *volatile *array`, where `T` is the object type. If the first volatile is omitted, a thread might read incorrect values from the object. If the second one is omitted, a thread might read incorrect pointer from the buffer array. Algorithm 4.3 provides a pseudocode for `push()` and `pull()` functions.

As was suggested in Section 3.1.1, insertion of states happens at random, but all threads in a warp remove states from the buffer at the same time. With that in mind, the synchronized buffer can be improved to speed up removal of elements. Instead of having t separated buffers for t threads in a block, they are all grouped under one buffer. The last indices and mutexes are stored in a `lastIndices` and `mutexes` arrays. Last index and mutex of a specific thread can be accessed through its unique thread id. This way, when all threads in a warp read or write the last index, the request can be fully coalesced. Elements are stored in one big array. Individual elements of one thread are *interleaved* by the number of threads in a block (i.e. for a thread with the thread id tid, its elements are located in the array on indices $\{tid, tid + t, tid + 2 \cdot t, \ldots, tid + (lastIndices[tid] - 1) \cdot t\}$). This concept is illustrated in Figure 4.1

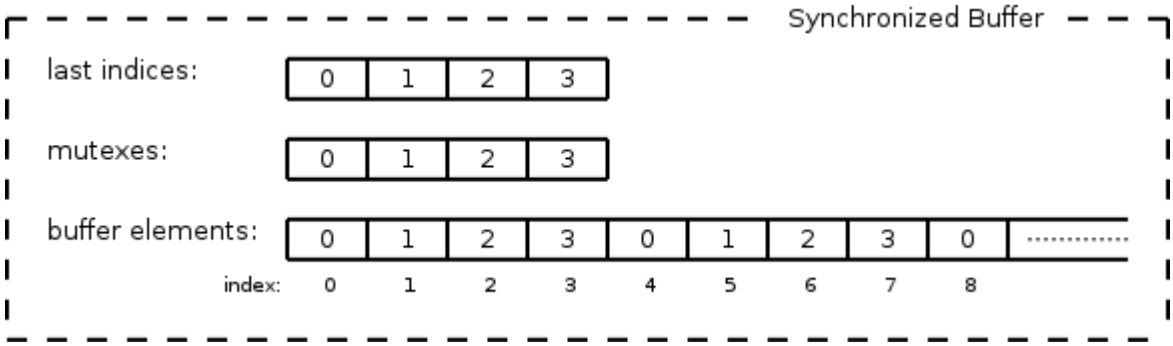


Figure 4.1: A synchronized buffer for $t = 4$

4.1.3 Non-Blocking Buffer

A locking is an expensive operation. It adds unnecessary overheads even if only one thread is trying to access a protected object, because the mutex must be first acquired and then released. If occurrence of concurrent access by multiple threads is rare (or at least adequately small), it might be better to try to insert (or remove) the element into (from) the buffer without locking it and occasionally, if the buffer were modified by another thread

[2]It never happened during the experiments.

in the meantime, repeat the action until it succeeds. An algorithm that allows concurrent access but does not required locking is called a *non-blocking* algorithm.

Treiber [25] proposed a non-blocking *LIFO* buffer. The buffer is internally represented by a *linked-list*[3]. Algorithm 4.3 gives the pseudocode of `Push()` and `Pull()` functions. It is almost the same as for an ordinary linked-list. The trick is on lines 7 and 14. The head is updated only if it was not changed by another thread. If it was, `atomicCAS()` would not update the head and the loop termination condition would not be satisfied. Hence the whole procedure would repeat until the head is updated (or the buffer is empty in case of removal). Since always at least one thread must update the head, a deadlock can never happen, not even if all threads try to pull or push the element at the same time.

Algorithm 4.4 LIFO non-blocking buffer pseudocode (based on [25])

```
 1: function PULL(head)
 2:     repeat
 3:         retVal ← head
 4:         if retVal = null then
 5:             return null
 6:         next ← GETNEXT(retVal)
 7:     until ATOMICCAS(head, retVal, next) = retVal
 8:     return retVal

 9: function PUSH(newElement, head)
10:     repeat
11:         oldHead ← head
12:         SETNEXT(newElement, oldHead)
13:         __THREADFENCE()
14:     until ATOMICCAS(head, oldHead, newElement) = oldHead
```

Note that only an update to memory inside the `Pull()` function is done through the atomic operation. Therefore, a memory barrier is not necessary. On the other hand it must be used inside the `Push()` function, because the reference to the next element is updated. In addition, both the head pointer and the object it points to must be declared as volatile, otherwise it is possible that either a wrong head or reference to the next element will be read (i.e. it must be declared as `volatile T *volatile head`). Similarly as for the synchronized buffer, it can be improved by grouping buffers of threads in a block together.

It was mentioned in Section 3.1.1 that the number of states that are pulled during each iteration can impact the performance. A non-blocking buffer offers a nice and easy solution to this problem. Note that all states in the buffer can be removed by simply removing the first one and setting head to *null*. Therefore, all states can be obtained from the buffer at the beginning and then they can be processed without accessing the buffer again. A possible drawback of this solution is that if one thread in a warp has much more states in the buffer than others, the remaining threads will be doing nothing, even if some new states are added into their buffer. This can be fixed by calling an ordinary `Pull()` function that removes only one state. An alternative solution is to use a modified function for removal of a state from the buffer that does not contains the loop. Instead, if `atomicCAS` fails it returns *null*. Let's call this function `TryPull()`. In both cases, the reference to the next state must be set to *null*.

[3] A list is a data structure consisting of nodes that are connected [26]. In linked-list, each node contains a reference to another node. Therefore, only the reference to the first node in the group need to be stored. The first node is called the *head* of the linked list.

4.2 Priority Queue

A data structure used for a priority queue in this work is a *binary heap*. A binary heap can be represented as an array [14], which makes it possible to at least partially optimize access of individual threads in a warp with respect to memory latency. The way individual nodes are mapped into the array is shown in Figure 4.2. The indices of parent and children nodes from the current node are computed by using Equation 4.1.

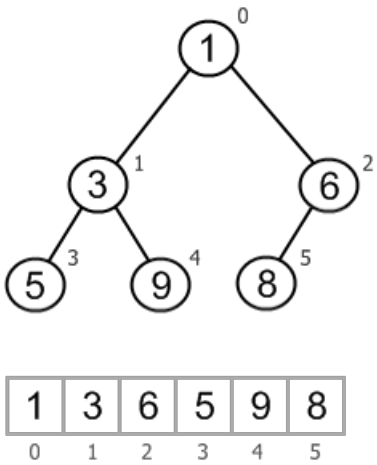


Figure 4.2: Array representation of a binary heap (source [27])

$$
\begin{aligned}
parent &= \frac{current - 1}{2} \\
leftChild &= (index + 1) \cdot 2 - 1 \\
rightChild &= (index + 1) \cdot 2
\end{aligned}
\tag{4.1}
$$

A simple implementation, when the heap of each thread is stored in a separated array, would not provide good performance, because access would not be coalesced even if threads in a warp requested an element with the same array index. Therefore, a better solution is to use the same technique as the one used for buffers. Priority queues of threads in a block are grouped together and stored in one big array. Individual elements are interleaved.

This changes computation of indices of parent and children nodes a little bit (4.2). A new formula can be divided into three parts: (i) transform a current index to normal scale (ii) compute a result using Equation 4.1 (iii) transform the result back.

$$
\begin{aligned}
parent &= \frac{\overbrace{\frac{current - tid}{t}}^{\text{transform current to normal scale}} - 1}{2} \cdot \overbrace{t + tid}^{\text{transform result back}} \\
leftChild &= \left(\left(\frac{current - tid}{t} + 1 \right) \cdot 2 - 1 \right) \cdot t + tid \\
rightChild &= \left(\left(\frac{current - tid}{t} + 1 \right) \cdot 2 \right) \cdot t + tid
\end{aligned}
\tag{4.2}
$$

4.3 Accelerated A* Excluded Area

Every computationally intensive part of Accelerated A* is related to excluded areas. Therefore, suitable representation is crucial for quick run of the algorithm. This section describes representation that is used in the AgentFly system and discusses a problem with inflated zones generation that arises on the GPU.

4.3.1 Representation

Each excluded area consists of three or more edges and the minimum and maximum altitude. For simplification, any excluded area will be referred simply as a zone. Each edge can be represented by two vectors with their origin in the Earth center. Let's call these edges ray_1 and ray_2. They define a plane of the edge that can be expressed by its normal vector, denoted as $normalPlane$. In general, the normal vector can have two directions: it can either be oriented to the zone or away from it. Which one is used is not important, but it must be unified for all edges, otherwise it will be impossible to work with it. Hence, it is defined that the $normalPlane$ is oriented away from the zone.

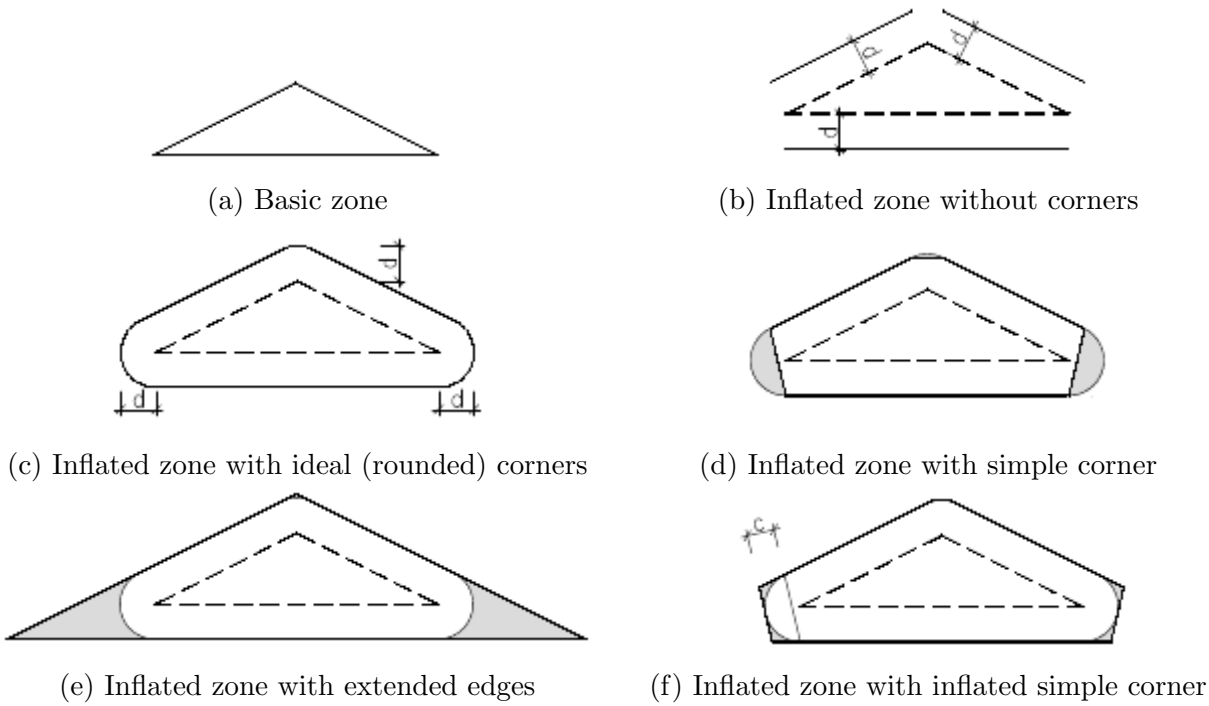
All zone's edges must form a convex shape. This condition simplifies many tasks that are often used in AA* such as a test if a point is inside the zone. To determine it, only a *dot product* with each edge must be computed. If any dot product is bigger than zero, the point does not lie inside the zone.

4.3.2 Inflated Zones Generation

Remember that a planning step is determined by testing the distance to the nearest zone. One way of doing it would be to construct a circle with a radius equal to the tested distance, and to test whether any zone is inside or intersects with the circle. However, since the zones stay the same but it is necessary to find out the planning step for a different state many times, it would not be a good solution. The effective solution is the one that performs all the hard work only once and then uses it to perform an inexpensive test many times.

This is the approach used in the AgentFly. When a particular distance needs to be tested, the zones are *inflated* by the tested distance. The zone is inflated by rotating its edges by the angle that is proportional to the tested distance. The result is that each zone also covers the tested distance. The edges of the inflated zone are called *inflated edges*. The test itself is nothing more than testing the state whether it is inside any inflated zone or not. If it is, then that zone is closer than the tested distance. The inflated zone can be saved so if it is requested again, no additional work is needed. Furthermore, it can be computed only on demand (i.e. the first time it is needed), therefore no unnecessary overhead is created.

This is not possible on CUDA. The inflated zones must be pre-computed for all tested distances. Fortunately, it can be easily parallelized - each thread can generate one inflated edge. The only problem is with the generation itself, which, without any modifications, would not run well on the *CUDA* architecture. Figure 4.3a shows a cut of a zone with three edges. This zone is inflated by a distance d. The result is given in Figure 4.3b. It is easy to see that something is wrong. The rotation created an empty space between adjacent edges, which is unacceptable. The intuitive (and optimal) solution would be to connect them with a rounded corner that respects the tested distance, i.e. the distance between the vertex formed by the original adjacent edges and the corner is (at least) d. Unfortunately, this is not possible to do in the selected representation. However, it can be approximated by several edges. This solution is the one used on the CPU.



(a) Basic zone

(b) Inflated zone without corners

(c) Inflated zone with ideal (rounded) corners

(d) Inflated zone with simple corner

(e) Inflated zone with extended edges

(f) Inflated zone with inflated simple corner

Figure 4.3: A cut of a zone with three edges.

The reason it would not work well on the GPU is that it is not known in advance how many elements will be needed to approximate each corner. As result, memory would have to be allocated dynamically for each corner. Therefore, write would not be coalesced and it would require some post processing to sum up all edges and moved them into a single array so they can be effectively accessed later during the search. Also a warp divergence would further slow down the generation. Hence, another solution must be used.

The easiest solution is to add one edge between adjacent edges that will fill the space (Figure 4.3d). It is fast and it adds only one additional corner. Therefore it is not a problem to pre-allocate memory for all corners so the access can be coalesced. Another option is to simply extend adjacent edges until they intersect with each other (Figure 4.3e). This is a little bit less effective since intersection must be computed. On the other hand, it does not add any corner edge. Hence, fewer edges must be checked during the search. Both solutions add some error (a cross-hatched area in figures). The first one underestimates the size of the zone. Generally, this type of error is unacceptable, since it can result in a situation when a longer planning step is selected and a solution to a problem is missed. The second one overestimates it. This cannot affect the solution, therefore it can be used. Nonetheless, it has another drawback. If the angle between edges is very small, the error will be extremely big even if the zone is very small. As a result, a smaller step will be selected and consequently more states will be generated.

Let's call the first type of error an error of type A and the second one an error of type B. The goal is to find a solution that does not have an error of type A and which minimizes an error of type B. Such a solution is given in Figure 4.3f. It uses the same simple corner as in Figure 4.3d, but it inflates it by distance $c = d - v$, where v is the distance of the vertex of the original zone to the simple corner. The newly generated corner is tangent to the ideal rounder corner. Therefore, its error is always of type B and it is guaranteed to be adequately small.

Chapter 5

Experimental Evaluation

This chapter experimentally evaluates two parallel planning algorithms for CUDA architecture presented in this work. First, the execution time of CUDA Hash Distributed A* is compared to the original A* in Section 5.1. Then, the empirical evaluation is used to inspect the properties of CUDA Parallel Accelerated A* algorithm in Section 5.2.

Individual tests were run on the following configurations:

A* Intel(R) Core(TM)2 Duo CPU T8300 @ 2.40GHz, 4 GB DDR 667 MHz, Gcc 4.4

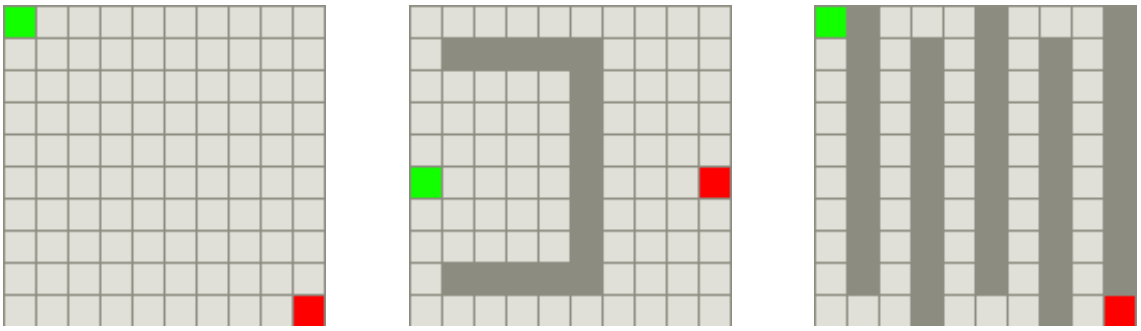
AA* Intel(R) Core(TM)2 Duo CPU T8300 @ 2.40GHz, 4 GB DDR 667 MHz, Java 1.6 OpenJDK

CHDA* GeForce GTX 460 SE, 1 GB GDDR5 1700 MHz, CUDA Toolkit 4.2, Driver Version: 295.41

CPDA* GeForce GTX 460 SE, 1 GB GDDR5 1700 MHz, CUDA Toolkit 4.2, Driver Version: 295.41

5.1 CUDA Hash Distributed A*

This section evaluates the performance of CUDA Hash distributed A* by comparing it with the original A* search. Algorithms were tested in a two-dimensional grid setup with two movement models: (i) four-way unit cost (ii) eight-way unit cost. The experiments were performed on two types of grid: (i) selected grids. (ii) randomly generated grids.



(a) Free grid (b) Obstacle (c) Maze

Figure 5.1: Selected two-dimensional grid planning setups. A green square denotes the start position and a red square is the goal position.

Selected grids that were used for testing are provided in Figure 5.1. The first setup (a) is an *empty grid* without any obstacle. The start is in the left-upper corner and the goal

in the right-bottom corner. This task complies well with A*, since preferable states are on the path from the start to the goal state. The second task (b) is more challenging. A *u-shaped obstacle* in the middle of the grid forces A* to expand less favored states to find a solution. The last one (c) is a *maze* without any cross-roads. In this scenario, the heuristic is completely useless. Therefore, an ordinary breath-first search returns a solution faster than A* since it does not have to compute the heuristic.

Randomly generated grids were tested with three different probabilities of a cell being blocked. The used probabilities are: (i) 10 % (ii) 20 % (iii) 30 %. Each probability was used to generate ten tasks (i.e. a random grid, start and goal position). The execution time for individual tasks was summed-up to evaluate the performance for the given probability.

Each task was tested twenty times to avoid imprecision due to unforeseen events which a processor must handle. The result for each task is the average value of repetitive tests. All planning scenarios were tested with four different grid dimensions: (i) 100×100 (ii) 1000×1000 (iii) 2000×2000 (iv) 5000×5000.

In addition, all configurations were tested with three levels of optimization: (i) no optimization [-O0 -g -G] (ii) level 0 [-O0] (iii) level 2 [-O2]. The first one is used, because the CUDA compiler makes some optimizations even if -O0 flag is set. The only way to ensure that optimizations are not used at all is to compile it with *debugging symbols*. Each section first discusses the results without any optimization and then assesses the benefits of the optimizations.

5.1.1 CHDA* Implementation Properties

The main aspect of CHDA* implementation was to evaluate its speed. Memory requirements were not the subject of interest. Therefore, a closed list was implemented simply as an array with the same size as the tested grid. This is a memory inefficient solution if a grid is sparse (i.e. it contains a lot of blocked cells), but allows it to keep implementation simple, since a hash table with dynamic memory allocation does not have to be used. In addition, a hash table would be inefficient by itself, because access to individual elements happens randomly in nature and threads in a warp would most likely access non-successive addresses.

To decrease the amount of memory requests, the grid and the closed list are stored in one common *float* array. Each cell contains one of the following values: (i) -1 if a cell is blocked (ii) ∞ if a cell has not been visited yet (iii) $g(s)$ if a cell has been already expanded. When a new state is expanded, its path cost is compared with the value in the array. If it is smaller then the state is added into the open list. Otherwise it is discarded (either the position is blocked or it has been visited with a better path cost). Queue indices are stored in a similar way. Each cell contains a positive index in the open list or -1 if a state is not in it.

CHDA* kernel was launched with 32 threads in a block and 32 blocks.

5.1.2 Four-Way Unit Cost

In the four-way unit cost movement model, four actions are possible: (i) go left (ii) go right (iii) go up (iv) go down. The cost for each action is one. A commonly used heuristic with this model is *Manhattan distance*[1].

Selected Setups

Results of experiments are provided in Table 5.1. A* beats CHDA* in all configurations.

[1] See Section 2.4.3.

	No Optimization		Level 0		Level 2	
Setup	A*	CHDA*	A*	CHDA*	A*	CHDA*
a	1.68	32.21	1.31	24.27	0.32	25.40
b	4.01	40.59	4.27	27.04	1.05	27.63
c	1.42	3 542.47	1.68	3 324.27	0.41	3 323.69

(a) 100×100

	No Optimization		Level 0		Level 2	
Setup	A*	CHDA*	A*	CHDA*	A*	CHDA*
a	41.20	632.98	41.64	446.24	18.19	445.16
b	556.59	906.55	537.63	555.31	118.17	564.44

(b) 1000×1000

	No Optimization		Level 0		Level 2	
Setup	A*	CHDA*	A*	CHDA*	A*	CHDA*
a	82.80	1 820.68	83.23	1 693.90	46.09	1 697.38
b	2 442.66	5 472.90	2 442.66	2 391.33	489.57	2 360.61

(c) 2000×2000

	No Optimization		Level 0		Level 2	
Setup	A*	CHDA*	A*	CHDA*	A*	CHDA*
a	493.31	7 609.12	483.75	3 917.71	293.94	3 887.80
b	15 383.38	17 421.73	14 892.99	18 413.75	3 164.41	18 654.94

(d) 5000×5000

Table 5.1: The execution time in milliseconds of A* and CHDA* for selected setups with a four-way unit cost movement model for different map dimensions and different levels of optimization.

The most noticeable gap is in the maze (c). CHDA* is 2495 times slower in the smallest map. Since the maze has only one possible path, only one thread has a state for expansion in the open list at any given time. Therefore, during each iteration, 31 blocks have nothing to do and in the last one, only one thread has a state in the open list. Hence, the algorithm expands the same states as A*, but with additional synchronization overhead. This behavior remains the same for larger maps which results in even slower execution. The CHDA* was unable to return a solution within five minutes for a 1000×1000 grid.

CHDA* still cannot compete with A* in the free grid (a), but the difference gets smaller on larger maps (20 times slower on a 100×100 against 15 times on a 5000×5000). This is a consequence of the fact that it takes some time before all threads have some states in the open list and that it also takes some time to copy and prepare data on the device and to launch the kernel. It is a common property of all CUDA programs that when a problem is small it is better to solve it on the CPU.

The single obstacle scenario (b) turned out best for CHDA*. It was still ten times slower than A* for the smallest grid but for the largest one, it was almost just as good as A* (only 1.13 times slower). The difference between results for scenarios (a) and (b) is caused by the obstacle in the second one. In general, CHDA* expands more states than A*. If there are no obstacles, all additional states expanded by CDHA* are useless. However, the obstacle forces A* to visit some of these states too. Therefore, while A* is still checking states that do not lead to a solution, some thread of CHDA* may be already on the right path.

The obvious differences between A* and CPAA* are also in the impact of optimizations on execution time. As expected, A* shows a little increase in speed when debugging symbols are turned off, but with the second level, it is approximately five times faster. CHDA* exhibits different behavior. In most cases, the speed up is between one to two when debugging symbols are turned off and no additional speed-up is achieved with further optimizations. The initial speed-up is mostly caused by improving the usage of registers (i.e. storing temporary results in the registers). The exception is the configuration (b) for the largest grid, where optimizations slowed down the algorithm.

Random Grids

Table 5.2 provides run-times for randomly generated grids. The same as for selected setups, A* is faster than CHDA*.

On random maps with a 10 % probability of a cell being blocked CHDA* had the worst performance in the smallest configuration (66 times slower). As the grid size increased, the relative difference became smaller. On the largest map, A* was approximately only four times faster. CHDA* had a similar progress with the second configuration where the probability was 20 % (79 times slower for 100×100, eight times for 5000×5000). In both cases, the performance on a grid of size 2000×2000 was only slightly worse than the one achieved on the largest grid map.

The last configuration, where a chance of a wall was 30 %, was the only exception. CHDA* was thirty times slower for 100×100, but the relative difference between CHDA* and A* was best for 2000×2000 when it was six times slower. Nevertheless, on 5000×5000 it was 6.3. Hence, this different behavior could just as well be the result of a random generation.

The results of all random setups suggest one conclusion. The grid size must be at least 2000×2000 to fully utilize CHDA*.

Similarly as for selected setups, A* speeds up when the second level of optimizations is used while the CHDA* benefits from the optimization only when debugging symbols are turned off.

Prob. of	No Optimization		Level 0		Level 2	
blocked cells	A*	CHDA*	A*	CHDA*	A*	CHDA*
10 %	3.28	216.47	3.33	191.35	0.87	192.48
20 %	3.85	303.580	4.24	255.53	0.97	261.13
30 %	9.27	280.49	9.57	264.55	2.62	261.60

(a) 100×100

Prob. of	No Optimization		Level 0		Level 2	
blocked cells	A*	CHDA*	A*	CHDA*	A*	CHDA*
10 %	328.95	5 345.57	288.39	3 003.04	128.66	2 999.51
20 %	332.00	5 348.47	304.22	2 968.42	127.70	2 960.37
30 %	514.35	6 760.79	402.38	3 784.30	162.34	3 777.22

(b) 1000×1000

Prob. of	No Optimization		Level 0		Level 2	
blocked cells	A*	CHDA*	A*	CHDA*	A*	CHDA*
10 %	1 876.07	11 531.25	1 309.48	3 869.21	558.43	7 571.33
20 %	1 814.86	20 491.65	1 664.95	12 380.90	656.67	12 397.52
30 %	4 721.60	28 723.71	4 425.55	17 923.01	1 375.87	18 096.15

(c) 2000×2000

Prob. of	No Optimization		Level 0		Level 2	
blocked cells	A*	CHDA*	A*	CHDA*	A*	CHDA*
10 %	6 089.63	25 375.57	5 697.42	19 672.76	2848.30	19 849.96
20 %	10 006.38	78 249.84	9 645.92	58 613.88	4044.16	58 207.52
30 %	13 458.91	85 637.70	12 357.02	59 466.30	4960.97	59 216.03

(d) 5000×5000

Table 5.2: The execution time in milliseconds of A* and CHDA* in random grid maps with a four-way unit cost movement model for different map dimensions and different levels of optimization.

5.1.3 Eight-Way Unit Cost

The eight-way unit cost movement model extends the four-way unit cost movement model by adding four actions: (i) go diagonally left up (ii) go diagonally right up (iii) go diagonally left down (iv) go diagonally right down. The cost of these additional actions is $\sqrt{2}$. Because of the diagonal movement, a Manhattan distance can overestimate the real path cost. Therefore, it is a non-admissible heuristic for the eight-way movement model. An admissible heuristic that gives the best estimate is *Euclidean distance*[2].

Selected Setups

Table 5.3 summarizes run times for individual setups and levels of optimization. The same as for all previous experiments, A* is faster in all cases.

	No Optimization		Level 0		Level 2	
Setup	A*	CHDA*	A*	CHDA*	A*	CHDA*
a	0.28	120.97	0.24	116.83	0.06	118.37
b	5.06	106.90	5.24	93.02	1.17	87.94
c	2.43	10 584.31	2.58	2 835.55	0.59	2 661.65

(a) 100×100

	No Optimization		Level 0		Level 2	
Setup	A*	CHDA*	A*	CHDA*	A*	CHDA*
a	15.26	1 196.63	15.23	1 483.13	10.32	1 522.00
b	724.05	2 654.97	659.93	2 215.19	162.83	2 190.50

(b) 1000×1000

	No Optimization		Level 0		Level 2	
Setup	A*	CHDA*	A*	CHDA*	A*	CHDA*
a	57.04	4 230.27	58.35	9 055.44	39.33	9 225.50
b	3 202.49	15 472.90	2 865.03	11 545.12	740.01	11 767.87

(c) 2000×2000

	No Optimization		Level 0		Level 2	
Setup	A*	CHDA*	A*	CHDA*	A*	CHDA*
a	314.24	17 445.35	331.17	88 859.38	245.68	88 915.86
b	21 272.71	99 690.19	19 562.52	132 366.45	5 882.78	130 601.15

(d) 5000×1000

Table 5.3: The execution time in milliseconds of A* and CHDA* for selected setups with a eight-way unit cost movement model for different map dimensions and different levels of optimization.

[2]See Section 2.4.3.

For the first two configurations, the same conclusions apply as for the four-way unit cost movement model with the difference that the gap between execution times of A* and CHDA* is even bigger. Briefly, in the maze (c), CHDA* is 4 350 times slower. For the empty map (a), the ratio between CHDA* and A* is 432 for 100×100 and 56 for 5000×5000.

Unlike for the four-way movement, in the (b) CHDA* provided the best performance on the configuration of size 1000×1000. It was 3.7 times slower, while for 5000×5000 the slowdown was 4.6.

The large gap is the result of additional actions and the fact that it must be possible to re-open already closed states. Therefore, it cannot be discarded right after the expansion, but it must be sent to the appropriate thread which can decide whether there is a better state in the closed list or not. The additional slowdown in the maze scenario is a consequence of the map representation which is unified with the closed list.

The effect of optimizations stayed unchanged for A* and for CHDA* in the configuration (b). However, the (c) and (a) had a different behavior. In the maze, a significant speed up was achieved. On the other hand, for (a) the effect was the opposite, and from 1000×1000 it slowed down the algorithm. For a grid of size 5000×5000, CHDA* with optimizations was five times slower than without it. This behavior makes the effect of optimizations for CUDA kernels difficult predictable.

Random Grids

The measured run-times are provided in Table 5.4. Again, A* performed better than CHDA* for all configurations.

For all setups, the ratio between A* and CHDA* for a grid of size 100×100 was worse than the one with the four-way movement model (72 times slower for the first one, 112 times for the second one, and 99 times for the last one). Nevertheless, it did better for 5000×5000. The ratio was 2.4 for a grid with a probability of 10 % of blocked cells, 4.2 for a grid with 20 % and lastly, 3.9 when the chance of a wall was 30 %. Also the performance on a 2000×2000 grid did not differ much. Therefore, it further supports the argument, that at least a map of size 2000×2000 is required, otherwise CHDA* is highly ineffective.

In addition, the effect of optimization on CHDA* is even more vague since it speeds up all tasks for all grid dimensions except the largest one. For 5000×5000, only the last configuration had better performance with optimizations. The remaining two run slower.

5.1.4 Prepare Arrays Kernel

Before the search kernel is launched, the map must be converted into the format that was specified in Section 5.1.1 and individual cells of queue indices array must be set to -1. This can be done in a single kernel. The number of array cells that must be set for each array is $n = width \cdot height$. Each thread can process one or more elements. Figure 5.2 shows the execution time of *prepare arrays kernel* for different number of threads per block and different number of elements processed by one thread.

Sooner or later, all values oscillate around 0.036 milliseconds. If each block contains at least 512 threads, it does not matter how many elements the thread will process. If it is less, the minimum number of elements that each thread must process depends on the number of threads per block. Logically, if there are less threads in a block, more elements must be processed by one thread.

This is necessary due to the fact that only a limited number of blocks can reside on one SM. As a consequence, the SM stays unoccupied, which would normally result in a situation when a warp scheduler has no warps with available instructions. However, since the values

Prob. of	No Optimization		Level 0		Level 2	
blocked cells	A*	CHDA*	A*	CHDA*	A*	CHDA*
10 %	8.54	617.55	8.04	580.13	1.91	579.94
20 %	8.05	983.33	8.05	788.01	1.94	796.91
30 %	7.63	753.26	6.67	702.31	1.68	699.07

(a) 100×100

Prob. of	No Optimization		Level 0		Level 2	
blocked cells	A*	CHDA*	A*	CHDA*	A*	CHDA*
10 %	957.52	8 167.61	884.35	5 705.32	269.80	5 621.30
20 %	640.14	7 410.61	580.50	4 961.38	199.83	4 976.95
30 %	666.52	8 440.63	628.41	6 477.74	243.24	6 439.16

(b) 1000×1000

Prob. of	No Optimization		Level 0		Level 2	
blocked cells	A*	CHDA*	A*	CHDA*	A*	CHDA*
10 %	4 758.99	23 233.84	3 869.21	22 170.88	1 251.64	21 969.13
20 %	5 495.353	34 280.015	4 873.18	24 369.42	1 545.58	24 506.91
30 %	3 023.69	16 981.80	2 774.43	11 815.41	1 095.08	11 768.87

(c) 2000×2000

Prob. of	No Optimization		Level 0		Level 2	
blocked cells	A*	CHDA*	A*	CHDA*	A*	CHDA*
10 %	28 340.55	68 685.33	25 115.29	115 197.10	8 959.94	114 941.88
20 %	32 483.51	137 706.01	28 275.34	145 833.33	9 833.89	148 131.31
30 %	21 973.61	86 170.50	20 287.06	43 578.01	8 334.94	44 863.05

(d) 5000×5000

Table 5.4: The execution time in milliseconds of A* and CHDA* in random grid maps with an eight-way unit cost movement model for different map dimmensions and different levels of optimalization.

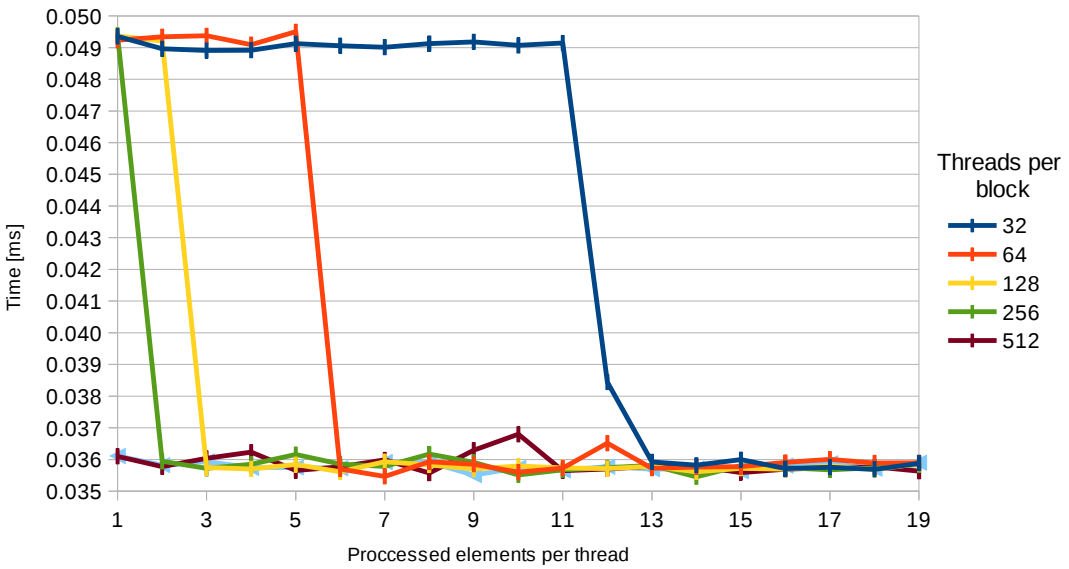


Figure 5.2: Execution time in milliseconds of *prepare arrays kernel* for different parameters. Each configuration was tested 500 times on a grid of size 5000×5000 and the final value is the average of those tests.

of individual elements are independent, a thread can proceed with the following element without waiting for the previous write to finish.

5.2 CUDA Parallel Accelerated A*

This section provides experimental evaluation of presented CUDA Parallel Accelerated A*. First, it is compared to the original Accelerated A* on selected flights in the *United States Airspace* (Figure C.2). Second, the impact of the planning-grid size on performance is inspected.

Each configuration was run twenty times and the stated results are the average of these repetitive tests. Unfortunately, the tested implementation of CPAA* failed to work when optimizations were allowed. Due to the algorithm complexity, it requires further analysis to discover the source of this behavior. Therefore, CPAA* was tested only without optimizations.

In addition, there was a problem with the dynamic memory allocation. Dynamic allocation inside the kernel function is supported from compute capability 2.x [20]. Before it can be used, the heap of fixed size for dynamic allocation must be specified before the kernel is launched. This operation takes some time, especially if the heap size is large. Normally, this would not matter, because it can be done only for the first time as long as all threads release allocated memory before the kernel terminates.

However, when CPAA* was run more than once, the execution time was longer with each iteration. This did not happen during tests of CHPA*. However, unlike CHPA*, CPAA* uses most of the available memory for dynamic allocation at the end of the kernel. Therefore, this suggests a problem with dynamic allocation when it is forced to look for a free segment in the heap.

To overcome this problem, the CUDA context was destroyed after each run of the algorithm. This, however, also meant that heap had to be allocated before the next run again. To provide an undistorted view on the CPAA* performance, the measured times do not include the time for heap allocation.

5.2.1 CPAA* Implementation Properties

Implementation of CPAA* includes all load-balancing improvements that were discussed in Section 3.2. A block consists of 32 threads (i.e. there is only one warp in a block) that cooperates on computationally intensive parts. During each iteration, the threads select an open list that has the state with the lowest $f(s)$ and expand only one state from this open list before moving to the next stage.

A task scheduler and buffers are implemented as a non-blocking LIFO queue that was covered in Section 4.1.3. The algorithm terminates if either all states are expanded or if all remaining states have the cost worse than the best found path. Furthermore, the algorithm can be instructed to find only a suboptimal solution that is guaranteed to be no worse than k % of the optimal solution. Lets call this property a limit of the solution (i.e. if the limit is 5 %, the found path will not be worse than 5 % of the best path).

To accelerate computation of a modulo operation, the number of blocks must be a power of two. Therefore, the kernel is launched with 32 blocks, which is the maximum number of blocks that satisfies it and that can reside at the same time on the tested GPU.

		CPAA*		
Setup	AA*	0 %	1 %	5 %
A	2 359.60	23 840.00	8 800.31	752.99
B	95.45	905.77	153.22	85.78
C	482.75	6 936.65	2 486.20	2 561.56
D	302.35	4 406.45	751.65	754.41
E	2 684.90	22 494.84	9 499.13	5 217.18
F	1 247.65	26 384.73	19 841.42	19 617.47
G	1 898.60	70 036.33	59 037.44	58 455.04

(a) Execution time in milliseconds

		CPAA*		
Setup	AA*	0 %	1 %	5 %
A	1 615 259	1 615 356	1 622 452	1 672 750
B	986 529	986 694	992 200	998 031
C	2 126 046	2 125 690	2 130 160	2 130 025
D	3 981 238	3 981 422	4 008 835	4 017 991
E	2 150 602	2 150 661	2 159 159	2 183 103
F	2 000 423	2 000 555	2 008 272	2 009 415
G	2 698 917	2 698 764	2 708 466	2 707 436

(b) Path length in meters

Table 5.5: The execution time and path length on selected flight plans in the United States Airspace for AA* and CPAA* with three different limits.

5.2.2 Comparison with Accelerated A*

The coordinates of the air-plane can be expressed by its *longitude*, *latitude* and *altitude*. In the experiments, a constant altitude was used. Therefore, the planning reduces to a two-dimensional setup. The longitude then corresponds to the x-dimension in Figure 3.2 and

the latitude to the y-dimension. The $x_{padding}$ and $y_{padding}$ were expressed as a ratio of x_{diff} and y_{diff}. Specifically in the executed tests, it was 0.25 for both dimensions. In addition, CPAA* was tested with three limits: (i) 0 % (ii) 1 % (iii) 5 %

The results are provided in Table 5.5. Figure C.1 contains a sample flight plan for setups (A) and (G).

When the limit is 0 %, CPAA* is always slower than AA*, but it returns a path with the same cost. Small differences are caused by imprecision of the floating point arithmetics [28] and the fact that CPAA* uses a single-precision floating point representation, but AA* uses a double-precision. In most setups, the slowdown is between ten to twenty percent. The exception is the last one (G), where CPAA* was almost 37 times slower.

Let's take a closer look at setup (A) (Figure C.1a). A straight path between the initial state and goal state is not possible because of the excluded areas. However, none of them forces the search to look completely elsewhere. Therefore, most expanded states should lie between the start and goal state. Hence they should also fit inside the planning-grid.

Figure 5.3 shows states expanded by the original AA* and CPAA*. AA* (a) expands states that are close to the ideal path. When it searches near an obstacle, the density of states is larger due to the small planning step, but it never expands a state that does not lead to the solution. Once it is clear of all zones, the search rushes directly towards the goal state.

CPAA (b) exhibits a little different behavior. It expands much more states than AA*. This was in part expected, because the only way how to parallelize a search algorithm is to also expand states that do not have the best estimated path cost. It is beneficial if the sequential algorithm will expand them later. Unfortunately, for the selected setup, AA* has no reason to do so (and therefore it will not). Nevertheless, that is not the biggest problem.

Remember that in order to find an optimal solution, CPAA* must enable re-opening of already closed states. Given the nature of the algorithm, this happens a lot. Furthermore, the impact of this behavior is actually increased by the fact that the replaced states can have successors that have successors and so on. In general it would be computationally expensive and impractical to remember all successors. Therefore, they are simply left without any modifications. As a consequence, successors of a new state will most likely replace them which will start this cycle all over again.

In addition, the work with the hash table (and therefore the similarity test as well) is not very effective, since the access happens at a random and therefore it cannot be coalesced.

The described problem is even more serious in the setup (G). States expanded by AA* and CPAA* with the limit 0 % are illustrated in Figure 5.4. CPAA* generates a massive amount of states in the area where the planning step is large. In other words, it completely breaks the main property of Accelerated A*, which is to accelerate the search when there are no excluded areas around.

The performance of CPAA* improves if the limit is bigger than 0 %. When the limit was set to 5 %, CPAA* was even faster on the first two setups, roughly 3 times on (A). Note however, these two were the only exceptions and on all the remaining configurations, it was still much slower than AA*.

Figure 5.3c and Figure 5.3d illustrate states expanded by CPAA* when the limit was 1 % and 5 %. The number of expanded states is smaller. In (d), the states that are closed to the excluded zones were completely omitted. The number of states was reduced by two modifications. The first one is rather obvious. Since a solution is only required to be within k % from the optimal one, the search can be terminated as soon as none of the opened states has a path cost better than k %. The second one pertains to the closed list. The already closed state has to be re-opened only if its path cost is better than k %.

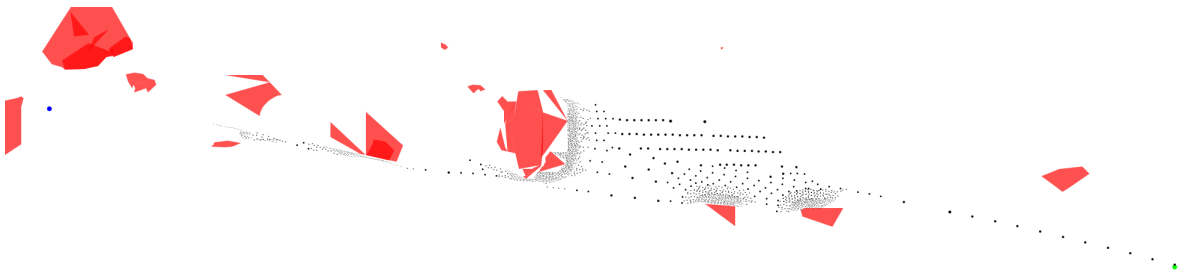
(a) AA*

(b) CPAA* 0 %

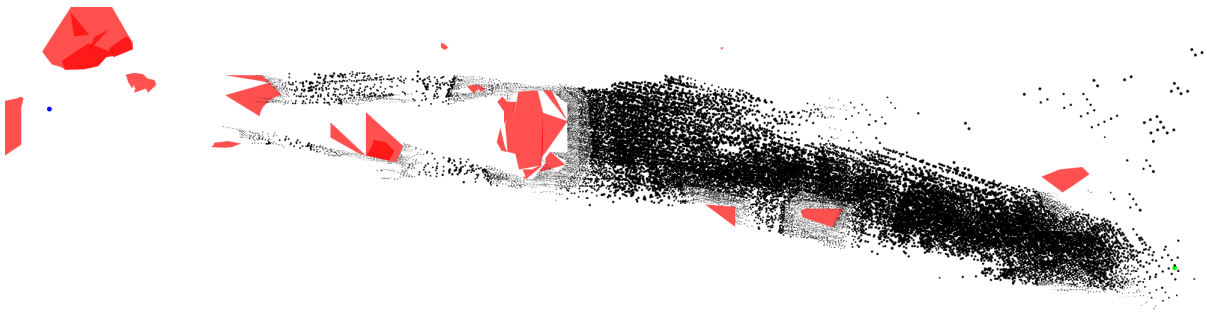
(c) CPAA* 1 %

(d) CPAA* 5 %

Figure 5.3: Expanded states in setup (A)



(a) AA*



(b) CPAA* 0 %

Figure 5.4: Expanded states in setup (G)

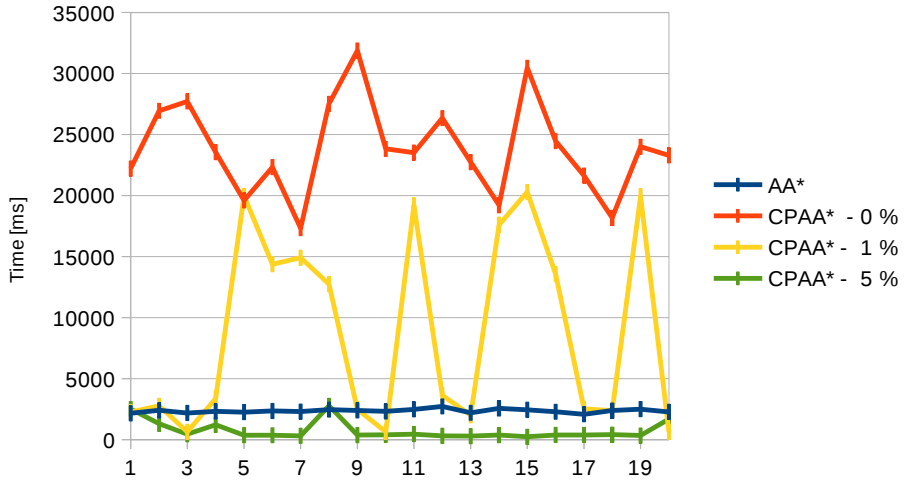
Therefore, the length of the final path depends on the way the partial paths are found. The limit k is merely the upper-boundary on a solution error. Figure 5.5 provides the progress of execution time and path cost for individual tests. For AA* and CPAA* with zero limit, the path length remains the same. In addition, the execution time of the original AA* remains more or less the same for the whole time. On the other hand, the execution time of CPAA* differs quite a lot. Also when $k > 0$, there is a dependency between the execution time and the path length. If time decreases the length increases and vice versa.

5.2.3 Impact of the Planning-Grid

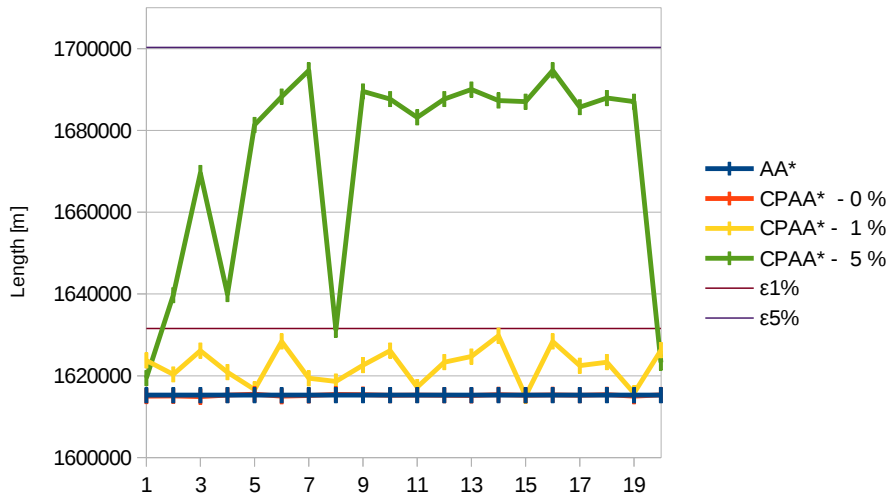
This section discusses the impact of the planning-grid on the execution time of CPAA*. The experiments were done on a sample scenario with a single obstacle that diagonally blocks the space between the start and goal state (Figure 5.6). It was intentionally chosen in such a way that it is also larger than the rectangle formed by the start and goal states. Thereupon, the search is forced to expand states that are outside the ideal path.

Figure 5.7 summarizes the results in a chart. Padding is expressed as a ratio of x_{diff} for the x-dimension and y_{diff} for the y-dimension. It has the same value for both of them. In the tested configuration, the ratio must be at least 0.25 for a grid to cover the whole excluded area. The figure also contains the execution time of AA*, which does not use the planning grid. Therefore its execution time remains the same. In general, this type of scenarios is the one where a parallel algorithm should be faster than a sequential one, because it is forced to expand less favored states anyway. And yet, all versions of CPAA* are slower. The difference between the execution times of AA* and CPAA* decreases as the size of the planning grid increases. In the global maximum, CPAA* with limit 5 % is even faster.

The selected configuration contains only one zone. Hence an intersect test and a planning step detection are inexpensive, which implies that the slowdown is caused by the open and closed lists and the number of generated states.



(a) Execution time in milliseconds



(b) Path length in meters

Figure 5.5: A progress of execution time and path length for setup (A) over individual runs.

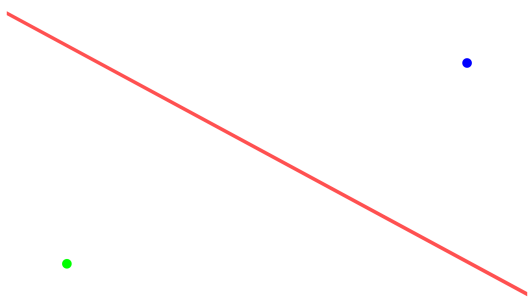


Figure 5.6: A configuration used for planning-grid evaluation.

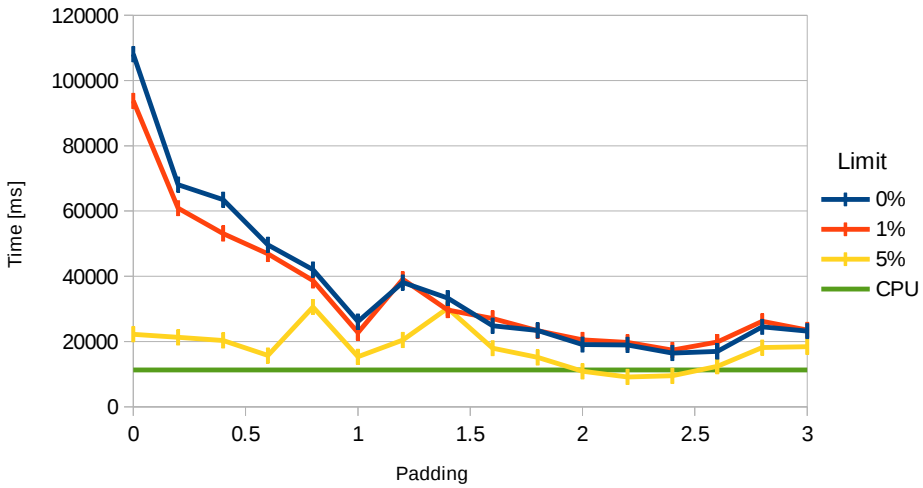


Figure 5.7: A time dependence of CPAA* on the planning-grid dimensions.

The time dependence of CPAA* has the same course when a limit is 0 % and 1 %. It starts with relatively high value and steadily decreases until the local minimum is reached when the padding ratio is one. The long execution time when the padding is small is the result of the situation that most of the expanded states fall outside the planning grid. Hence they are all assigned a default hash value. Each new state must be compared for similar states. Possibly similar states are selected according to their hash. The new state has to be at least compared with all states that have the same hash. Therefore, if there are a lot of states with the default hash, the large array must be processed and the similarity test becomes expensive and a possible bottleneck of the algorithm.

Surprisingly, the execution time further decreased when the padding was increasing. Both versions have a global minimum of around 2.4. Since the planning grid is already large enough to cover most states, the additional speed up must be caused by something else. With the grid size, the size of individual sectors increases as well which means that newly expanded states are less likely to change a sector. As a consequence, only a few blocks will have states for expansion and the rest will only process states from the scheduler. The fact that it gives better performance only supports the observation made in the previous section that the distribution of states is not the right approach.

CPAA* with the limit 5 % does not seem to be as much dependent on the size of the planning-grid as the other two versions. The limit enables it to discard most states that are only slightly better therefore the similarity test does not have to check as many states.

Chapter 6

Conclusion

This thesis explored possibilities of parallel trajectory planning on the CUDA architecture. Two algorithms were suggested and tested. The first one is a parallel version of A* search [11] called CUDA Hash Distributed A* (CHDA*). The second one is built on Accelerated A* algorithm [16] and it is called CUDA Parallel Accelerated A* (CPAA*).

The first part of the thesis (Chapter 2) explained the planning problem as it was defined by Russel et al. [10]. It then provided the review of two classes of search algorithms - an uninformed search that only has information about states which are provided by the problem definition, and an informed search that uses additional knowledge about the problem to select a state for expansion. Then two algorithms that this work is based on were described. Namely A* search that uses additional knowledge to estimate the path cost and Accelerated A* that uses varying planning steps to speed up the search.

The second part addressed the parallel planning on the CUDA architecture. First CHDA* and CPAA* were introduced (Chapter 3). CHDA* is a modification of Hash Distributed A* [4] that was designed for distributed systems. It distributes states between available processors asynchronously by their hash value. CHDA* uses this concept to distribute states between threads that in general, can be in different blocks. Therefore, the algorithm relies on a block synchronization that is not officially supported by NVIDIA. CPAA* follows the same basic structure as CHDA* but it adds a few improvements. First, the search space is divided into sectors. Each thread is responsible for states in one sector. Second, threads in a warp cooperate on intensive parts such as an intersection or point-level test. At last, intensive parts can be computed by any thread (warp).

Then, it discussed some implementation details (Chapter 4). The most space was dedicated to communication between blocks on the CUDA architecture. Two solutions to this problem were suggested - a synchronized buffer that uses a locking mechanism to add or remove an item and a non-blocking buffer that is represented by linked-list and uses an atomic compare and swap operation to update the head of the list. Next, the interleaved array representation of a binary heap and the impact of representation of no-flight zones in the AgentFly on the CPAA* were described.

The last part of the thesis analyzed the presented algorithms (Chapter 5). CHDA* was compared against the original A* search in a two-dimensional grid setup with four-way and eight-way unit cost movement models. In all experiments, CHDA* was slower than A*. Poor performance of CHDA* follows from the fact that the GPU excels in floating point operations, but apart from the computation of heuristic, A* is mostly about access to memory, which, on the other hand, is something that CUDA is not good at. In general, CHDA* performed the worst when there were not any obstacles on the ideal path.

CPAA* was tested in three versions that differed in the cost of the returned path. The first version (a) always returned the best path. The other two were guaranteed to return a

path that was no worse than 1 % (b) and 5 % (c). Two types of experiments were conducted. First, the execution time and path cost of CPAA* were compared with the implementation of AA* in the AgentFly on selected flights in the United States Airspace. The version (a) was significantly slower than AA*. The version (b) performed a little bit better, but it was still noticeably slower. Only the version (c) was faster at least on some tasks. However, the found path was not as good as the one returned by AA*. The detailed analysis showed that CPAA* expanded a lot of states when a planning step was large. This behavior was caused by the distribution of states that resulted in a situation when a lot of already closed states had to be re-opened.

The second type of experiment was designed to investigate the dependence of a planning grid on execution time. CPAA* was tested with a different size of the planning grid on a simple configuration, where a straight path from the start to the goal state was diagonally blocked by a single zone. As expected, CPAA* was very slow when the planning grid was smaller than the zone, but the execution time dramatically improve as the size of the planning grid increased until it was large enough to encompass all expanded states. Nevertheless, compared to the original expectations, the execution time slowly decreased when the planning grid was further enlarged. The fact that this was happening only supported the argument that the distribution of states has negative side effects.

To summarize it, both CHDA* and CPAA* were slower than the original sequential algorithms. CHDA* paid for frequent accesses to global memory and lack of floating point operations that would hide the long latency of global memory. On the other hand, AA* contains enough computationally intensive parts, therefore, CPAA* should comply well with the CUDA architecture. However, the experimental evaluation showed that the problem lies in the basic concept of the algorithm, because the distribution of states together with varying planning steps causes expansion of a lot of similar states.

6.1 Future work

Accelerated A* has a potential to run faster on the CUDA architecture. Therefore, as an extension to this thesis, the parallelization of AA* can be further explored, especially following two concepts:

- **Master - slave hierarchy**: One block (master) would be responsible for operations on an open and close list and the other blocks (slaves) would perform all computation intensive parts such as an intersection test, point-level detection, or smoothing.
- **Shared open and closed list**: All threads in a kernel would be synchronized before each iteration. Each thread would receive one state from the open list and expand it. Threads would be synchronized again at the end of the iteration and they would collectively insert newly generated states into the open and closed list.

Bibliography

[1] D. Šišlák, M. Pěchouček, P. Volf, D. Pavlíček, J. Samek, V. Mařík, and P. Losiewicz, *AGENTFLY: Towards Multi-Agent Technology in Free Flight Air Traffic Control*, ch. 7, pp. 73–97. Birkhauser Verlag, 2008.

[2] M. Evett, J. Hendler, A. Mahanti, and D. Nau, "Pra*: Massively parallel heuristic search," tech. rep., College Park, MD, USA, 1995.

[3] W. D. Hillis, *The Connection Machine*. Cambridge, MA, USA: MIT Press, 1989.

[4] A. Kishimoto, A. Fukunaga, and A. Botea, "Scalable, Parallel Best-First Search for Optimal Sequential Planning," in *Proceedings of the International Conference on Automated Scheduling and Planning ICAPS-09*, (Thessaloniki, Greece), pp. 201–208, 2009.

[5] R. Zhou and E. A. Hansen, "Parallel structured duplicate detection," in *Proceedings of the 22nd National Conference on Artificial Intelligence - Volume 2*, AAAI'07, pp. 1217–1223, AAAI Press, 2007.

[6] E. Burns, S. Lemons, R. Zhou, and W. Ruml, "Best-first heuristic search for multi-core machines," in *Journal of Artificial Intelligence Research 39*, pp. 689–743, 2009.

[7] E. Burns, S. Lemons, W. Ruml, and R. Zhou, "Parallel best-first search: The role of abstraction," in *Abstraction, Reformulation, and Approximation*, 2010.

[8] A. Bleiweiss, "Gpu accelerated pathfinding," in *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '08, (Aire-la-Ville, Switzerland, Switzerland), pp. 65–74, Eurographics Association, 2008.

[9] S. Edelkamp, D. Sulewski, and C. Yücel, "Gpu exploration of two-player games with perfect hash functions," in *Proceedings of the Third Annual Symposium on Combinatorial Search, SOCS 2010, Stone Mountain, Atlanta, Georgia, USA, July 8-10, 2010* (A. Felner and N. R. Sturtevant, eds.), AAAI Press, 2010.

[10] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3 ed., Dec. 2009.

[11] P. Hart, N. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *Systems Science and Cybernetics, IEEE Transactions on*, vol. 4, pp. 100 –107, July 1968.

[12] D. E. Knuth, *The Art of Computer Programming, Volume 1 (3rd ed.): Fundamental Algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1997.

[13] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd ed., 2001.

[14] J. Williams, "Algorithm 232: Heapsort," *Commun. ACM*, vol. 7, pp. 347–349, June 1964.

[15] A. Patel, "Amit's A* pages." `http://theory.stanford.edu/~amitp/GameProgramming/`.

[16] D. Šišlák, P. Volf, and M. Pěchouček, "Accelerated a* path planning," in *Proceedings of 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009)* (S. C. Decker, Sichman, ed.), (Hungary), pp. 1133–1134, May 2009.

[17] D. Šišlák, *Autonomous Collision Avoidance in Air-Traffic Domain*. PhD thesis, Czech Technical University in Prague, Faculty of Electrical Engineering, Feb. 2010.

[18] Š. Kopřiva, D. Šišlák, D. Pavlíček, and M. Pěchouček, "Iterative accelerated a* path planning," in *Proceedings of 49th IEEE Conference on Decision and Control*, (Atlanta, GA, USA), Dec. 2010.

[19] J. Bentley, *Programming pearls (2nd ed.)*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000.

[20] NVIDIA Corporation, *CUDA C Programming Guide 4.1*. Nov. 2011.

[21] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st ed., 2010.

[22] N. Corporation, "Fermi compatibility guide for CUDA applications," May 2011.

[23] D. B. Mike Banahan and M. Doran, *The C Book*. Addison Wesley, second ed., 1991.

[24] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *Commun. ACM*, vol. 8, pp. 569–, Sept. 1965.

[25] R. K. Treiber, "Systems programming: Coping with parallelism.," Tech. Rep. RJ 5118, IBM Almaden Research Center, Apr. 1986.

[26] M. V. Wilkes, "Lists and why they are useful," in *Proceedings of the 1964 19th ACM National Conference*, ACM '64, (New York, NY, USA), pp. 61.1–61.5, ACM, 1964.

[27] "Binary heap. array-based internal representation." `http://www.algolist.net/Data_structures/Binary_heap/Array-based_int_repr`.

[28] "Ieee standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1 –58, 29 2008.

[29] "Intel Pentium 4 Processor 570J." `http://ark.intel.com/products/27475/Intel-Pentium-4-Processor-570J-supporting-HT-Technology-(1M-Cache-3_80-GHz-800-MHz-FSB)`.

[30] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2010.

[31] N. Corporation, "*NVIDIA*'s Next Generation CUDA Compute Architecture: Fermi," 2009.

[32] P. N. Glaskowsky, "NVIDIA's Fermi: The First Complete GPU Computing Architecture," Sept. 2009.

[33] N. C. Micikevicius Paulius, "Local Memory and Register Spilling." lecture, 2011.

Appendix A

CUDA

The twenty-first century is without a doubt the age of parallel computing. It is no longer the domain of expensive supercomputers. Almost every personal computer has a CPU with two, four or even more cores. Most personal computers are also shipped with a GPU. GPUs are relatively new, but as can be seen in Figure A.1, the modern GPU can outperform the CPU ten to a hundred times. The reason lies in a different design (Figure A.2) of the CPU and GPU, which is the result of varying original purposes. Note however, that the mentioned speed up is only theoretical. In order to fully utilize the GPU's resources, it is necessary to overcome certain limitations. If they are not met, performance can reduce dramatically. In the worst case, an application running on the GPU can even run much slower than on the CPU.

The first part of this chapter (Section A.1) briefly covers the history of the CPU and GPU development. The second part (Section A.2) introduces Fermi, the next generation of the CUDA architecture. The last part (Section A.3) provides the detailed describtion of the CUDA programming model.

A.1 Brief history

In order to understand limitations of the CUDA architecture and to be able to decide whether to use the CPU or GPU for computation, it is necessary to know a little bit about the development of these two processors.

A.1.1 Central Processing Units

The first personal computers contained CPUs that ran with internal clocks operating around 1 MHz [21]. As new technologies were invented and integrated circuits were getting smaller, the CPU clock speed increased. For thirty years, it was one of the most reliable sources for improved performance. In 2005, the *Intel Pentium 4* could run with a clock speed up to 3.8 GHz [29]. Because of the dramatic increase in the CPU clock speed, other factors such as memory latency and code branching[1] arose. Therefore, large *caches* and *control units* that can, for example, predict the result of the branch or alter the order of the instructions without changing the output, were added [30].

Nevertheless, due to the power and heat restrictions, as well as physical limitations to a transistor size, improving performance by increasing clock speed was no longer feasible [21]. Manufacturers were forced to look for alternative approaches. The inspiration came from

[1] Branching occurs when a code execution depends on some variable (i.e. during an *if-else* statement or a *loop*)

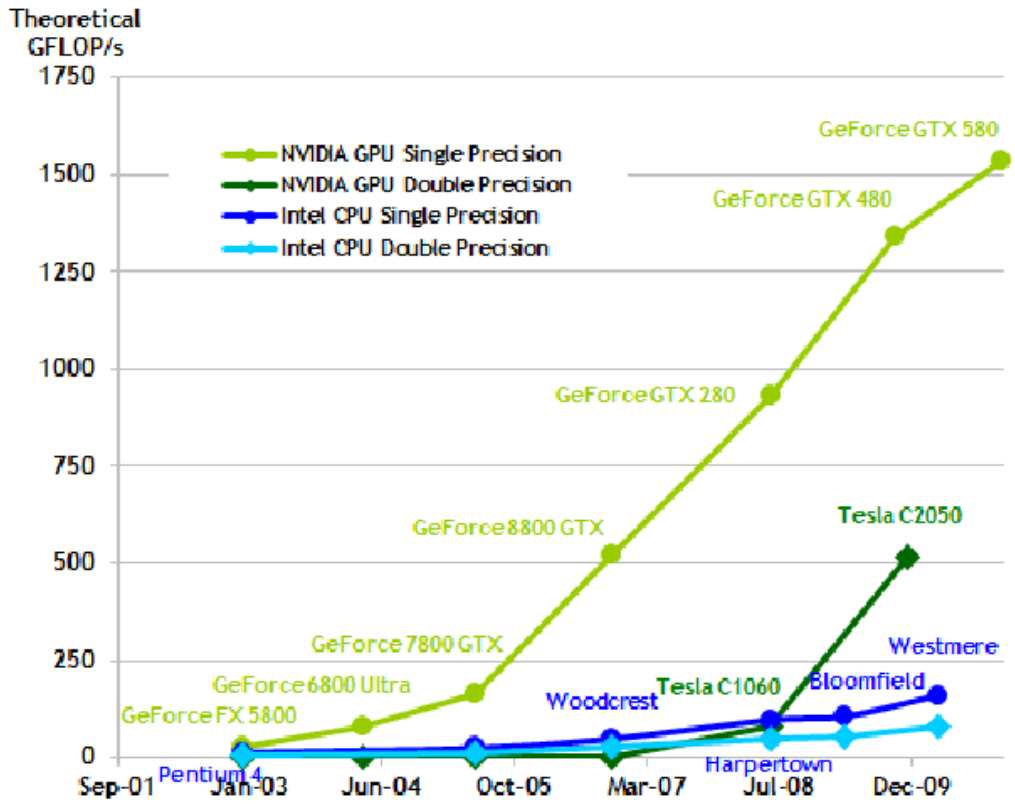


Figure A.1: Maximum (theoretical) number of floating point operations per second for the CPU and GPU. (source: [20])

the improvement of supercomputers which, besides improving a single processor, steadily increased the number of processors. Therefore, in 2005, leading vendors introduced processors with two computing cores instead of one and began the development of CPUs with even more cores. These processors have the same instruction set as the ordinary CPU, but multiple cores can execute more instructions at the same time.

A.1.2 Graphics Processing Unit

At the turn of the 1980s, the graphically driven operating systems such as Microsoft Windows helped to create a market for a new type of processor that offered hardware-assisted bitmap operations to assist in the display of the graphical context [21]. In the mid-1990s, the release of the first person games increased the demands for affordable graphics accelerators that would help to create more realistic 3D environments. This was achieved by NVIDIA's *GeForce 256*, which was the first graphics processor able to completely perform transform and lighting computations.

In the 2001, NVIDIA released *GeForce 3* series. It was the first one to implement Microsoft *DirectX 8.0* standard that required compliant hardware to contain both vertex and pixel shading stages that could be controlled by developers, and therefore, giving them some control over the computations performed by GPUs. In general, these GPUs were designed to produce a final color for each pixel in programmable arithmetic units known as *pixel shaders* by using its coordinates and some additional information such as input colors and textures. Researchers observed that since arithmetic performed on the input colors is control by programmers, they can trick the GPU into doing non-rendering tasks. The trick is to supplement real data as input colors, program pixel shaders to perform desired operations and then read the result as a final color.

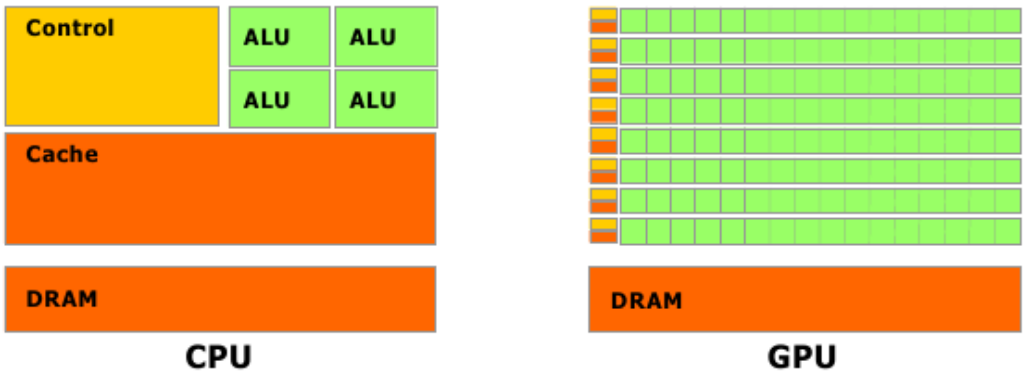


Figure A.2: The basic design of the CPU and GPU. The CPU devotes lots of resources on the big caches to hide memory latency. The GPU concentrate on the pure floating point computational power. (source: [20])

This had, however, several drawbacks. First of all, anyone who wanted to use the GPU for general-purpose computations would need to learn either *OpenGL* or *DirectX* so he or she could formulate his or her task as a graphical problem [30]. Secondly, there were tight memory constraints of how or where a programmer could write his or her results. Also it was undefined how a particular GPU would handle (if at all) floating-point data. Nevertheless, results from these earliest experiments showed new possibilities of what a graphics card can do.

After 5 years, in November 2006, NVIDIA unveiled CUDA[2] architecture and the first graphics card build-in with it, *GeForce 8800 GTX* [21]. This new family of graphics processors were specifically designed for general purpose programming. Vertex and pixel shaders were replaced by *unified shader pipeline*, allowing a programmer to use every ALU for general-purpose computations. In order to truly enable general-purpose computations, these ALUs supported floating point arithmetics specified by *IEEE* [28] and were designed to use an instruction set tailored for general computation. This required adding a large instruction memory, an instruction cache and an instruction sequencing logic. To avoid an extensive overhead with these modifications, several ALUs share these resources. This did not affect the original purpose of GPUs, because the same shared program needs to be applied to a massive number of vertices or pixels. NVIDIA also added memory load and store instructions that allowed a random byte addressing to fulfill the requirements of the *C* programs. Note however, that these GPUs did not support double floating point arithmetics, neither did they have any hardware-controlled cache.

A.2 Fermi Overview

The computational power of the Fermi GPU is divided into streaming-multiprocessors (SMs) that share a *global* memory and a *read-only constant* memory[3]. The number of SMs varies per device model, but the structure of each SM is the same for all GPUs with the Fermi architecture.

Figure A.3 provides the basic scheme of Fermi's SM. Each SM includes:

- 32 cores, that execute both a floating point and integer arithmetics instructions such as an addition or a multiplication

[2]CUDA stands for *Compute Unified Device Architecture*

[3]They also share a *texture* memory, but it is only the special case of global memory. See p. 58 for more information about memory hierarchy.

- 16 load/store units, which calculate a source and a destination address for sixteen threads per clock and load and store the data at each address to cache or *DRAM*.
- Four special-function units, that execute transcendental instructions such as sin, cos or reciprocal per clock.
- A register file with 32 768 words (registers). Registers are split evenly between all threads running on the multiprocessor.
- 64 KB RAM, which can be configured in ration 2:1 to be used as a software-controlled shared memory and a hardware-controlled *L1* cache.
- Thread control logic that consists of a common instruction cache and two warp schedulers.

Fermi was also designed for a double precision arithmetic. When a double precision operation is needed, two cores are used instead of one. Fermi can therefore perform up to 16 double precision operations per clock. Fermi is also the first GPU that includes a hardware-controlled cache. Apart from the configurable *L1* cache, a 768 KB *L2* cache shared by all SMs is available.

A.3 Programming Model

Even with the CUDA architecture, it was still necessary to learn *OpenGL* or *DirectX* to utilize the computational power of the GPU. Therefore, NVIDIA extended industry-standard `C` with a relatively small number of keywords and released a public compiler for this new language, which is known as CUDA C [20]. Together with the CUDA programing model, it provides a comfortable way to write applications that will benefit from new GPUs series without any modifications. Each CUDA device is characterized by *compute capability*, which determines what features it implements[4]. From now on, discussion will be strictly limited to the NVIDIA's Fermi series, which has a compute capability of 2.0 or higher. Note however, that most information applies to GPUs with a compute capability of 1.x as well[5].

A.3.1 Program Structure

A typical program usually consists of several phases that are executed either on the CPU (also known as *host*) or on the GPU (also known as *device*) [30]. A CUDA program is a unified source code that contains both a *host code*, written in `C/C++`, and a *device code*, written in CUDA C. A CUDA compiler, *nvcc*, separates them during the compilation process. The host code is then compiled using the standard `C/C++` compiler and run as an ordinary CPU process.

The code is executed on the device from the host by calling *kernel* functions. They are often referred to simply as *kernels*. The kernel typically generates a large number of *threads* that all execute the same program. Each thread is uniquely identified by its *block* and *thread index*[6]. Therefore, it can be used to supplement different data to individual threads.

When a device is used for the first time by the host thread, a CUDA context is created [20]. The CUDA context is analogous to a CPU process. All resources and actions are bound to this context. This includes memory allocated on the device and texture references. Normally,

[4]For example, a device of compute capability 1.3 and higher supports double floating point arithmetics
[5]See *Appendix F* of [20] for more information about compute capabilities.
[6]A *thread hierarchy* is explained in Section A.3.2

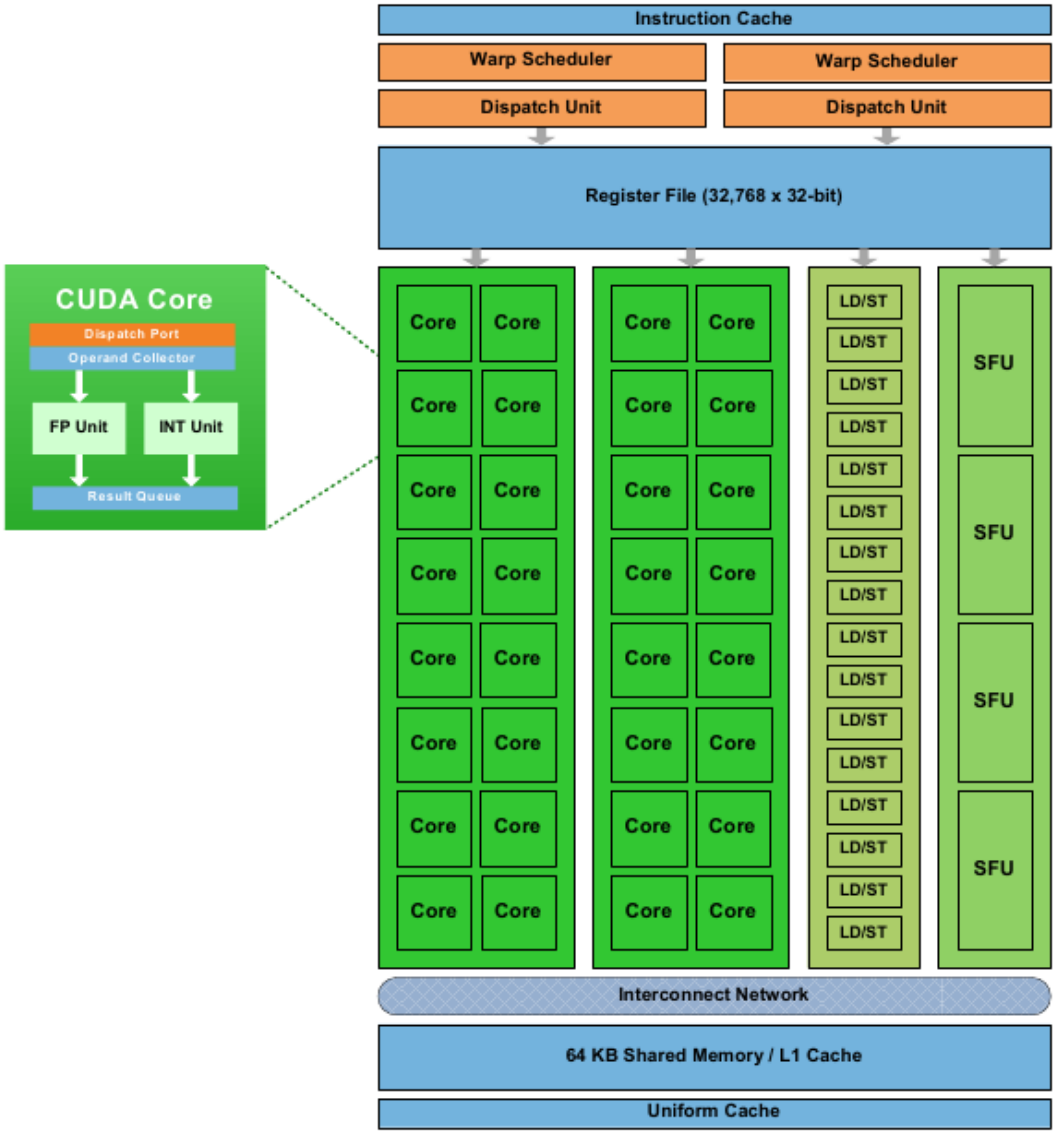


Figure A.3: Fermi multiprocessor (source: [31])

the CUDA context stays alive through the entire run of the host thread[7]. When the context is destroyed, all recourses are automatically cleaned-up by the system.

A.3.2 Thread Hierarchy

CUDA was designed to support scalable programming, where thousands of threads can be generated regardless of the actual device limitations [30]. As a result of that, the CUDA program can benefit from a new version of the GPU without any modifications. NVIDIA introduced the *three-layer thread model* to support this concept [20], consisting of: (i) *grid* (ii) *block* (iii) *thread*. Figure A.4 illustrates this model in a two-dimensional example. The purpose of a grid and blocks is mainly an abstraction designed to make a programmer's life easier.

Whenever a new kernel is launched, a large number of threads is created [30]. All threads execute the same program, the *kernel function*, but with different data that are selected based on the *unique id* within the thread hierarchy [20]. Threads are grouped into one-, two-, or three-dimensional blocks. The maximum number of threads per block is 1024 with

[7]It can be destroyed early by calling CUDA API functions. See Appendix G.1 of [20] for more information.

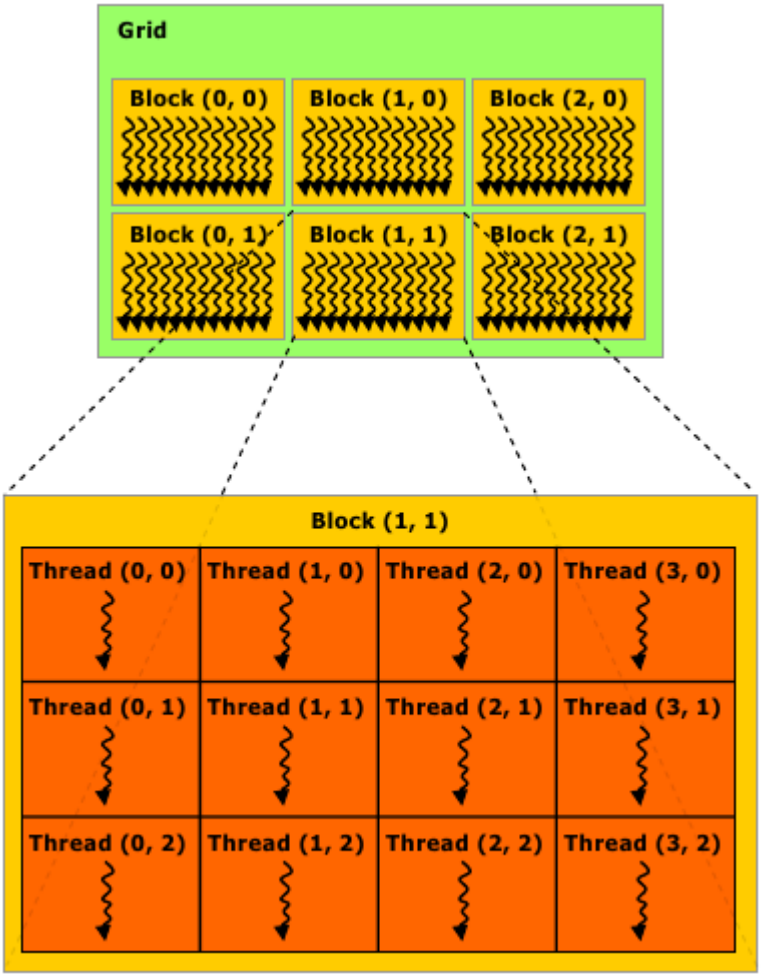


Figure A.4: CUDA thread hierarchy - a two-dimensional grid with two-dimensional blocks. (source: [20])

the maximum number of 64 threads in the z-dimension on Fermi. In order to fully utilize the GPU, the number of threads per block should be a power of two and no less than 32. Each thread is identified by its position within a block that can be obtained from a three-dimensional vector `threadIdx`. It is also possible to access all the block's dimensions through a `blockDim`.

Threads in a block can cooperate with each other [20]. They can exchange data through *shared* memory[8] and their execution can be synchronized by a *barrier synchronization*. The barrier synchronization , represented by a `__syncthreads()` instruction in CUDA C, stalls a thread until all threads in a block reach it. This is a powerful mechanism since it does not add much overhead. Yet, it can be very tricky and dangerous. If a *thread divergence*[9] occurs and `__syncthreads()` is placed inside a divergent path, it will not be reached by all threads and some (or all, if some other barrier follows) threads will halt and never finish [21].

Communication between individual blocks is not supported, neither is their synchronization. This results from the fact that blocks are required to be independent[10] (i.e. order

[8]See Section A.3.5 for more details.

[9]Thread divergence is a situation, when some threads in a block take a different path in the program than others (i.e. execute the instruction(s) that others do not) [21]. This path is then known as a *divergent path* [30]. This can often happen if some part of the code depends on the condition (i.e an *if-then-else* or a *loop*)

[10]This is more like a recommendation than a rule. As a matter of fact, this work heavily depends on blocks cooperation. Since compute capability 1.1, CUDA has supported *atomic functions* that performs

in which they are executed should not affect the output) [20]. Figure A.5 shows the scalability achieved by dividing a program into the set of independent blocks. Hence, writing a program that will benefit from future GPUs is nothing more than writing a program with enough blocks to occupy additional SMs.

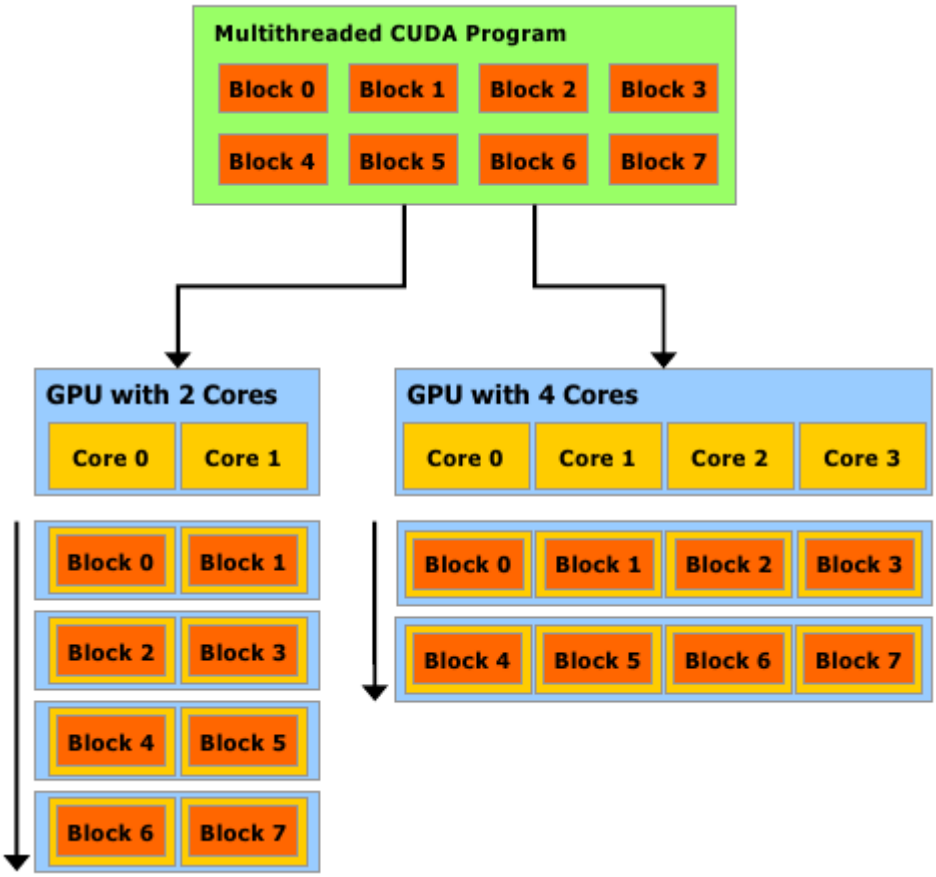


Figure A.5: CUDA scalability - A multi-threaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more SMs will automatically execute the program in less time than a GPU with fewer SMs. (source: [20])

The last layer in the CUDA thread hierarchy is a *grid*. Blocks are organized into a grid that can be either one-, two- or three-dimensional on Fermi [20]. The maximum number of blocks is limited to 65 535 per dimension. Each block is identified by its position on the grid that is accessible through a three-dimensional vector `blockIdx`. The grid dimension can be obtained from a `gridDim` variable.

A.3.3 Thread Execution

When a kernel is launched, the thread hierarchy is mapped to the *hardware hierarchy* of the GPU [31]. The grid is mapped to the available GPU. Modern GPUs, including Fermi, can execute more kernels, which belong to the same application (thread to be precise), at the same time. Since blocks are expected to be independent, they are distributed among SMs with the available execution capacity [20]. Multiple blocks can be assigned to one SM.

a *read-modify-write* operation without interruption from other threads [20]. Therefore, it is possible to implement synchronization and exchange data between blocks by using atomic operations on variables in global memory). Kernel will execute even if blocks are not independent, but it is up to the software developer to ensure that the program will run properly on a specific device. Note however, that NVIDIA discourages any attempt at block synchronization.

Threads from one or multiple blocks that reside on one SM are executed concurrently. Once a block is assigned to an SM, it stays there until all the threads in that block finish the kernel execution.

As was mentioned earlier, a typical CUDA program consists of hundreds to thousands of threads. To keep the cost of managing, scheduling and executing threads low, CUDA uses a unique architecture called Single-Instruction, Multiple-Thread, abbreviated as SIMT [30]. As the name indicates, a single instruction is executed by multiple threads, where each thread uses its own instruction arguments. Therefore, overheads, caused by fetching and processing instructions, can be amortized over multiple threads.

Fermi and all previous GPUs so far, partition threads in a block into groups of size 32, called *warps* [20], which are managed and scheduled together. Threads are divided consequently according to their *unique thread id*. Each thread in a warp has its own *instruction address counter* and *register state*. Therefore, they can execute independently and take different paths in a conditional code.

However, a warp executes one instruction at a time. Because of the SIMT architecture, this instruction can be executed by all warp's threads. If threads in a warp diverge (i.e. some of them take a different path in a conditional code), the warp execution will execute these paths one by one [30]. In the worst case, if all 32 threads take a different path, the execution will get completely serialized. Hence, to fully utilize GPU, threads in a warp should have the same control flow paths.

A.3.4 Warp Scheduler

As was shown in Figure A.3, each SM on Fermi contains two *warp schedulers*. Each one of them can issue one instruction every two cycles [31]. They are often referred together as a *dual warp scheduler*. Every two clock cycles, a dual warp scheduler select two warps and issues one instruction for each warp. This process is illustrated in Figure A.6. Since the execution context (i.e. program counters, registers, etc) for each warp are located directly on the SM for the entire lifetime, switching from one warp to another has no cost.

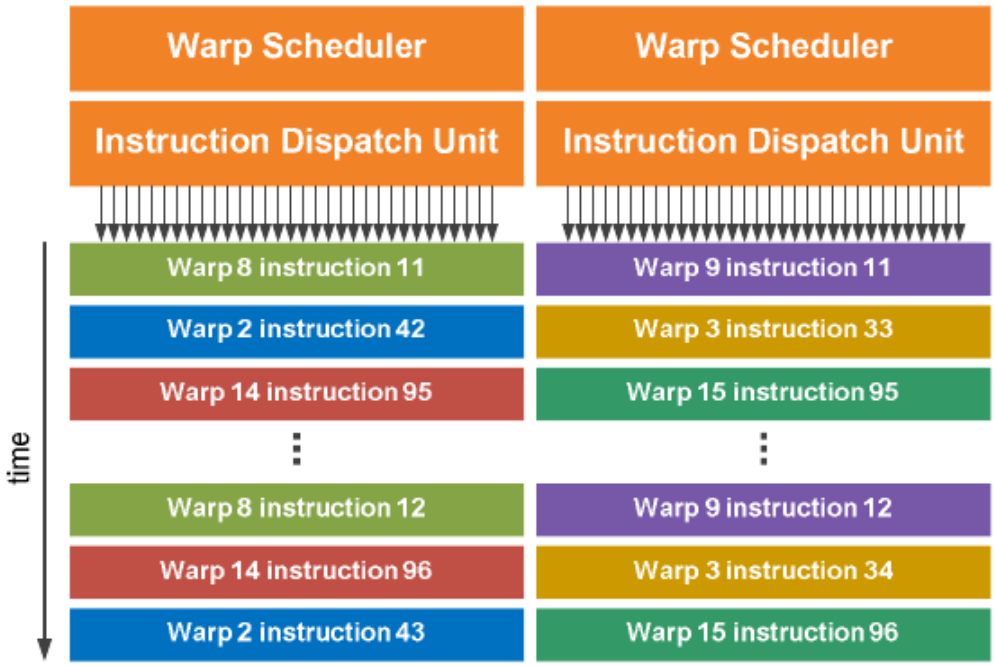


Figure A.6: Fermi Dual Warp Scheduler

32-bit instructions are issued to a group of 16 execution cores, 16 load/store units or four SFUs. Therefore, it takes two clock cycles to execute a warp arithmetic instruction on execution cores or a load/store instruction [32]. 32-bit special-functions instructions take eight clock cycles to execute, but they are issued in a single clock cycle. 64-bit instructions (i.e. double precision and long instructions) need all 32 cores to execute arithmetic instructions [31]. Therefore, only one warp can be selected every two cycles. Also special-functions instructions take twice as much time (i.e 16 clock cycles). This reduces the instruction throughput by a factor of two.

A.3.5 Memory Hierarchy

Unlike the CPU, the GPU does not rely on a hierarchy of large caches to hide memory latency [20]. Instead of that, it tries to hide it with computations. Hence, transistors devoted to big data caches on the CPU can be used for data processing. However, this puts some requirements on a programmer, because certain memory access patterns must be used as well as a ratio between arithmetics and load/store instructions must be met in order to achieve high performance. Hence, it is necessary to understand CUDA memory hierarchy. This is true even for Fermi which introduced *L1* and *L2* caches [31]. Physically, the GPU contains three types of memories [20]: (i) *local* registers (ii) *shared* memory (iii) *device* memory.

Local registers present the fastest and the smallest memory available on the CUDA device [20]. They are located directly on a SM. On Fermi, each SM includes 32 768 *registers*. These registers are assign per thread. Therefore, it is not possible to share results through a register. Registers are split evenly between all available threads on a SM. The maximum number of registers that Fermi can allocate to one thread is 63 [33].

Shared memory is another kind of *on-chip* memory (i.e. it is located on a SM) [20]. As the name implies, it can be shared between threads within a block. The content of shared memory ceases to exist with the end of the kernel.

Shared memory is divided into equally-sized memory modules, called banks, which can be accessed simultaneously [20]. Because it is on-chip memory, access to it is fast as long as a read or write request is made of n addresses that fall to n distinct memory banks. If two or more addresses fall into the same bank, bank-conflict occurs. In such a case, the original request must be split into as many bank-conflict free requests as necessary. The original request is said to cause *n-way* bank conflicts, where n is the number of separated bank-conflict free requests.

On Fermi, shared memory has 32 banks with a bandwidth of 32 bits per two clock cycles. Memory is organized in such a way that successive 32-bit words are assigned to the distinct banks[11]. If more threads request a read from the same address, the word is broadcasted within a single transaction without causing a bank-conflict. If more threads try to write to the same address, each byte is written by one of the threads. Which thread writes which byte is undefined.

Because of low latency, shared memory can be used as a temporally, software controlled cache [30]. A common example is matrix multiplication. Threads within a block can cooperate on loading part of the matrix into shared memory and then using it to perform the arithmetic operations[12]. Correct use of shared memory can have a significant effect on the final performance.

[11] For example no bank-conflict will occur if all threads within a warp read or write a corresponding *float* number from the array of size 32.

[12] See *Chapter 5* of [30] for more details.

Fermi is the first GPU that allows part of shared memory to be used as hardware controlled *L2* cache [31]. Each SM contains 64 kB memory that can be used in one of the following two configurations:

- 48 kB shared memory, 16 kB L2 cache.
- 16 kB shared memory, 48 kB L2 cache.

Device memory is the largest and the slowest memory on the GPU [20]. The latest GPUs support up to 6 GB *DRAM*. Device memory is shared by all kernels and its content exists through the entire CUDA context. Access to device memory takes between five to eight hundreds clock cycles.

The CUDA programming model splits device memory into four types, which differ in their usage and addressing: (i) *global* (ii) *constant* (iii) *texture* (iv) *local* memory.

Global memory is the most general memory available for the programmer that resides in device memory [20]. It supports read and write access and it is shared by all threads. Global memory is accessed via 32-, 64- or 128-byte naturally aligned[13] transactions. When a warp executes an instruction that accesses global memory, it coalesces accesses of individual threads into as few transactions as possible. The number of transactions depends on the size of the word accessed and the distribution of the memory accesses. The first GPUs had rather strict limitations - consecutive threads had to access consecutive addresses. These limitations became more relaxed with each new version of the CUDA architecture.

Another aspect that a programmer must be aware of is size and alignment requirements [20]. Global memory instructions can only access words of one, two, three, eight or 16 bytes that are naturally aligned. Otherwise, access has to be divided into multiple instructions. As result of that, access cannot be fully coalesced, which has a negative effect on the memory throughput.

Special care must be taken when using two-dimensional arrays. Common representation stores rows successively. If the size of a row does not meet alignment requirements, the following rows after the first one will not be naturally aligned. Hence, access to these rows will not be correctly coalesced. Fortunately, CUDA C provides several functions that allocate memory for two-dimensional array with correct padding at the end of the row[14].

Constant memory is the 64 kB read-only memory that resides in device memory [20]. Because data in constant memory cannot change, a cache can be used effectively. Each SM contains a special 8 kB cache that is designated only for constant memory. Right usage can significantly speed up the application.

Unlike global memory, constant memory does not use coalescing. It can only handle one address at a time. Therefore, the original request is split into as many requests as there are different memory addresses. This decreases throughput by a factor equal to the number of separated requests. Because each SM contains an *L2* cache on Fermi, it might be faster to use global memory instead of constant memory if each thread within a warp accesses a different address.

Texture memory is another read-only memory that resides in device memory [20]. Unlike constant memory, it does not have any size limitations. It shares its space with global memory. The difference between global and texture memory lies in access. Texture memory

[13] The address is a multiple of the accessed segment size.

[14] See [20] for more information.

is cached in a special cache that is optimized for a 2D spatial locality (Figure A.7). Note however, that it can still be used for 1-dimensional arrays. The size of the cache is device dependent, but it should be between 6 to 8 kB per SM.

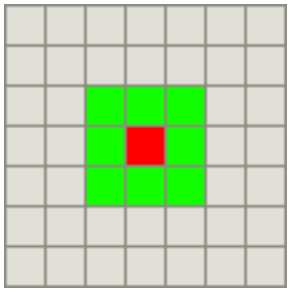


Figure A.7: A spatial locality of the texture cache. If a value at the *red* position is requested, adjacent values (*green*) are loaded into the cache as well [21]. This is different from the cache normally used on the CPU or by global memory on Fermi, where only adjacent values in a single row are loaded.

Texture memory has several other advantages [20]. First, data in texture memory can be either addressed by absolute coordinates (i.e. $0, 1, 2, \cdots, n-1$) or by linear coordinates (i.e. any number in $\langle 0, n-1 \rangle$), where the closest coordinates will be selected. For linear addressing, texture coordinates can be also normalized (i.e. any number in $\langle 0, 1 \rangle$). Secondly, it can remap *out-of-range* coordinates to the valid range based on the *addressing mode*. Lastly, filtering can be used if data are fetched by linear coordinates[15].

All of these computations (texture addresses, filtering) are computed by special load/store units designated only for texture memory. Therefore, load/stores instructions located on each SM can be used for some other memory requests.

Local memory is per-thread memory that resides in device memory [20]. It is used for some automatic variables such as large arrays or structures or whenever a kernel uses more registers than available [16]. The same limitations and requirements as for global memory apply for local memory[17]. In order to improve throughput, consecutive 32-bit words are allocated to the consecutive threads with respect to their *id*. Therefore, as long as threads within a warp access the same relative address, the access is fully coalesced.

On Fermi, each thread can use up to 512 kB of local memory.

A.3.6 Streaming Multiprocessor Occupancy

As was mentioned in the previous section, CUDA tries to hide memory latency by computation. How successful it will be, depends on two factors [30]:

- The number of arithmetic instructions per load/store instruction.
- The number of warps with instructions ready for execution.

The first one is self-explanatory. The second one is often affected by the number of warps that reside at a given time on a SM. The maximum number of warps per SM is 48 on Fermi [20]. Besides that, SM has several other limitations that can reduce the number of resident warps:

[15] See Appendix E of [20] for more details on texture fetching.
[16] This is known as *register spilling* [33]
[17] This also means that local memory is cached on Fermi

- The number of registers per SM (32 768 on Fermi). If a kernel uses too many registers, CUDA may be forced to assign less blocks per SM and hence reduce the number of warps.
- Amount of shared memory available per SM (Fermi can be configured to use either 16 kB or 48 kB shared memory). If a block uses too much shared memory, CUDA may be forced to assign less blocks per SM and hence reduce the number of warps.
- Maximum number of blocks that can reside on a SM (eight on Fermi). For example if a block consists only of 32 threads, then there can be no more than 256 threads (eight warps)
- Maximum number of threads that can reside on a SM (1536 on Fermi). For example if a block consists of 1024 threads, it will not be possible to assign more blocks on the SM and the remaining 512 threads (16 warps) will stay "unused".

A programmer must be well aware of these limitations. Selecting the right kernel configuration is not a simple task. Sometimes it can be better to use more registers even at the cost of reducing the number of blocks per SM. This can especially apply if number of threads per block is small. In another situation, it might be better to load some data again in order to save a few registers. Usually, it is necessary to experiment with several configurations in order to find an optimal solution.

Appendix B

Algorithms

Algorithm B.1 uses the following notations:

- *tid* and *bid* refers to a thread id within a block and a block id.
- when *tid* is use as a subscript, it denotes the variable of a given thread (e.q. $OPEN_{tid}$).
- all other variables are shared by all threads in a block.

Algorithm B.1 CUDA Parallel Accelerated A* algorithm pseudocode

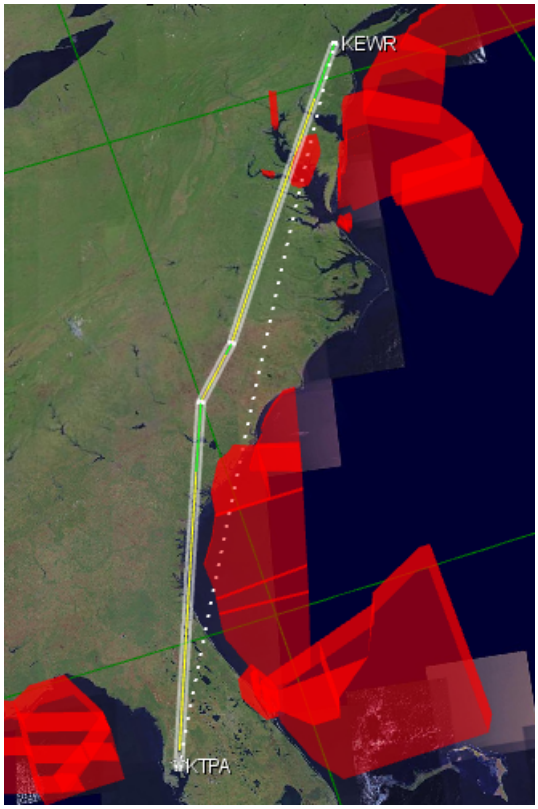
Require: pc_S and pc_G are valid, $finished = false$ **and** $emptyCounter = 0$
Ensure: *solution* contains a solution to the problem if it exists.

```
function CPAASearch(pc_S, pc_G, buffer, scheduler, finished, emptyCounter, solution)
    emptyFlag_tid ← false
    OPEN_tid ← ∅
    CLOSED_tid ← ∅
    (bid_pcS, tid_pcS) ← GetSectorId(pc_S)
    if bid_pcS = bid then
        fp^end ← Connect(pc_S, pc_G)
        s_S ← ⟨pc_S, DetectSamplingStep(pc_S), ⟨⟩, IsValid(fp^end), 0, Cost(fp^end), −⟩
        if tid_pcS = tid then
            InsertOrReplaceIfBetter(s_S, OPEN_tid)
    while ¬finished and (emptyCounter < n or ¬IsEmpty(scheduler, buffer)) do
        s_C_tid ← RemoveTheBest(OPEN_tid)
        Insert(s_C_tid, CLOSED_tid)
        foreach s_C_i do                                   ▷ Iterate over all s_C_tid
            if u^sC_i then
                s_G ← ⟨pc_G, −, Connect(pc^sC_i, pc_G), true, g^sC_i + h^sC_i, 0, s_C_i⟩
                s_G ← SmoothPath(s_G, pc_S)
                fp^result ← ReconstructPath(s_G)
                if tid = 0 then
                    UpdateIfBetter(solution, fp^result)
                    finished ← true
            else
                foreach fp_j ∈ Expand(s_C_i) do
                    if IsValid(fp_i) then
                        pc_N ← EndConfiguration(fp_j)
                        if tid = 0 then
                            Push(scheduler, (s_C_i, pc_N, g^sC_i + Cost(fp_i))
```

```
28:     while HasNext(scheduler) do
29:         (s_C, pc_N, g^N) ← Pull(scheduler)
30:         ξ_N ← DetectSamplingStep(pc_N)
31:         fp^end ← Connect(pc_N, pc_G)
32:         s_N ← ⟨pc_N, ξ_N, fp_i, IsValid(fp^end), g^N, Cost(fp^end), s_C⟩
33:         s_N ← SmoothPath(s_N, pc_S)
34:         if tid = 0 then
35:             (bid_{s_N}, tid_{s_N}) ← GetSectorId(s_N)
36:             Push(buffer_{(bid_{s_N}, tid_{s_N})}, s_N)
37:     while HasNext(buffer_{(bid,tid)}) do
38:         s_{C_tid} ← Pull(buffer_{(bid,tid)})
39:         if ContainsBetter(s_{C_tid}, CLOSED_tid) then
40:             continue
41:         else
42:             InsertOrReplaceIfBetter(s_{C_tid}, OPEN_tid)
43:     if IsEmpty(OPEN_tid) then
44:         if ¬emptyFlag_tid then
45:             AtomicInc(emptyCounter)
46:             emptyFlag_tid ← true
47:     else
48:         if emptyFlag_tid then
49:             AtomicDec(emptyCounter)
50:             emptyFlag_tid ← false
```

Appendix C

Figures



(a) Setup A

(b) Setup G (rotated by 90°)

Figure C.1: A path returned by Accelerated A* for selected setups. (source: AgentFly)

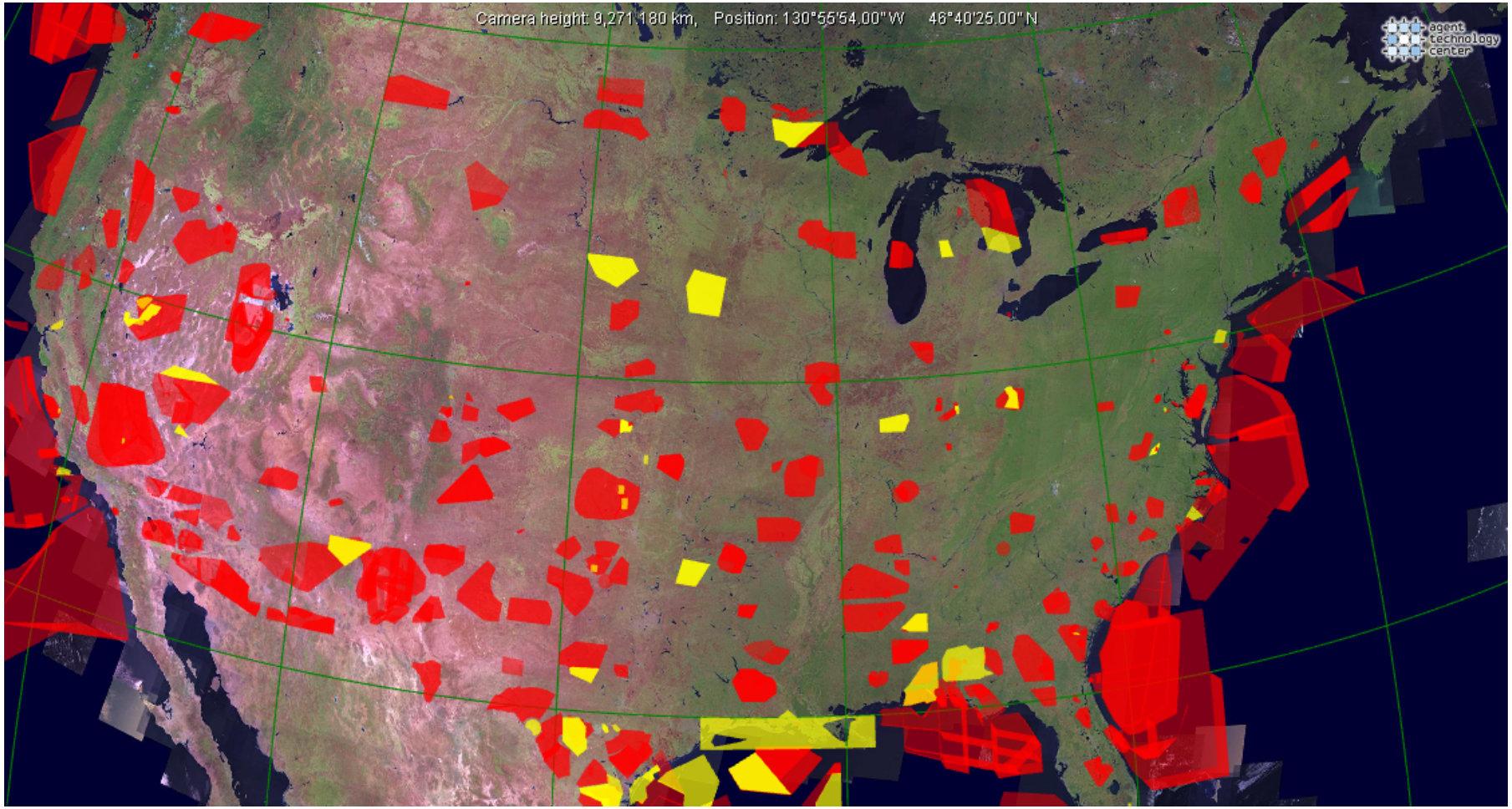


Figure C.2: United States Airspace with all no-flight zones. (source: AgentFly)

Appendix D

Contents of the CD

- CHDA* source codes
- CPAA* source codes
- Data for CHDA* experiments in text files
- Data for selected flights from the AgentFly in binary files for 32-bit architecture
- Figures of found paths by AA* for selected flights.
- Figures of expanded states by AA* and CPAA* for selected flights.
- Bachelor thesis in Portable Document Format (pdf)