PARALLEL ALGORITHMS FOR REAL-TIME MOTION PLANNING

THESIS PROPOSAL

MATTHEW MCNAUGHTON
mmcnaugh@ri.cmu.edu

THESIS COMMITTEE

Chris Urmson (chair)
Tony Stentz
Guy Blelloch, Computer Science Department
Dmitri Dolgov, Google Inc.

Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania, 15213

Latest revision September 23, 2009

Abstract

Real-time motion planning in complex dynamic environments requires new algorithms that exploit parallel computation. Robots that can move autonomously in complex and dynamic environments are desired for many purposes, such as efficient autonomous passenger vehicles, robots for industrial, agricultural, and military applications, etc. As an example, automating passenger vehicles would reduce traffic congestion and increase fuel efficiency while improving safety and revolutionizing the driving experience. Motion planning algorithms that can quickly plot a safe and efficient course through complex and dynamic environments require considerable computing resources. Future increases in computer performance will be realized by increasing parallelism, using processor architectures such as contemporary graphics processing units (GPUs), with multiple processor cores and SIMD instruction sets. This thesis will investigate parallel algorithms for motion planning, with a focus on algorithms relevant for autonomous ground vehicles. The work will involve both parallelizations of algorithms already used in motion planning, as well as development of new algorithms and motion planning approaches that are specifically designed to exploit the forms of parallelism offered by modern multi-core platforms.

In this proposal we present parallel algorithms to accelerate configuration free-space analysis and enable the creation of an improved heuristic function for heuristic search-based planning, we give a new single-source shortest path algorithm used to compute the heuristic, and we describe a novel state lattice structure and algorithm that will allow driving on-road at high speeds. Our parallel algorithms show significant performance improvements for motion planning tasks run on GPUs compared to sequential algorithms on traditional CPUs. In addition, the increased instruction throughput of GPUs enables problems to be solved in real-time that are too complex to be solved in real-time with traditional CPUs. We conclude that parallel algorithms are necessary to maximize the performance of motion planning systems, and we propose several extensions to our work, including a culminating demonstration on a robotic vehicle.

Contents

1 Introduction

The rate of performance improvement in sequential computing has slowed drastically in the last few years. The bulk of future computing performance improvements will be realized through processors that can run many tasks in parallel, and by developing algorithms that break problem instances into portions that can be solved in parallel. In order to effectively modify a sequential algorithm to run in parallel, we are often required to think about the problem in new ways.

We are interested in motion planning for robots. A motion planner for a robot uses a physical description of the robot and how it is capable of moving, together with a map of its surroundings, to produce a sequence of motion commands which, if executed by the robot, will take the robot from a given starting position to a desired ending position, while avoiding obstacles along the way and possibly optimizing some measure of motion quality such as smoothness of motion. Many interesting robot motion planning problems are NP-hard or even PSPACE-hard[49]. Systems to solve such problems must make use of as much processing power as they can. In order to continue increasing the performance of robot motion planning systems, we must learn how to apply parallel programming techniques.

Whereas there are few differences among the architectures of sequential computers, parallel computers are built in a variety of distinct forms, each one best-suited to running different types of parallel algorithms, and hence, solving different types of problems. When developing a parallel algorithm, it is helpful to decide first upon an architecture on which it is to be run. Contemporary graphics processing units (GPUs) are well suited to solving motion planning problems, as we will show. We will argue that technological and market forces will ensure the longevity of this architecture.

This thesis will explore ways of applying parallelism to the problem of robot motion planning. Our efforts will focus on algorithms that complement the strengths and weaknesses of GPUs. We will focus our efforts on robots with few degrees of freedom but where plans through complex environments with dynamic obstacles must be found quickly. We will demonstrate the value of our algorithms by implementing them on an autonomous passenger vehicle.

1.1 Motivation

In recent years, graphics processing units (GPUs) have evolved from specialized chips dedicated to rendering three-dimensional graphics for video games and CAD models, into general-purpose parallel processors. Algorithms that can be parallelized enjoy considerable performance increases on the GPU compared to their sequential counterparts. CPUs normally found in servers and desktop computers are also being manufactured with increasing numbers of cores that can run independent programs in parallel. The trend of regular and substantial improvements in the performance of sequential computers that peaked in the 1990s has come to an end. Processor performance is expected to increase mainly as a function of the number of processor cores that are incorporated into the chips, while the performance of individual sequential processor elements will likely increase only slowly. Parallel algorithms will be key to achieving continued performance growth.

Motion planning algorithms that have been demonstrated on real systems are typically sequential in nature, having been selected without regard to potential opportunities for parallelism. In order to exploit improvements in computing hardware to increase the performance of motion planning systems, it is necessary to develop algorithms that can take advantage of parallel computers.

Real-time motion planning demands hardware that can deliver high enough instruction throughput to enable a robot to perform interesting tasks, while fitting into a small physical package for incorporation into the robot. Due to their particular architectural features, GPUs can perform a relatively large number of computations for their phyiscal size, but due to these same features, they only perform well for certain workloads. In this document, we will explore the architectural features of GPUs that enable their performance, and look at some of the ways in which the motion planning problem is amenable to acceleration by the GPU.

In the following sections we will examine in further detail the motivations for this thesis: why are we interested in motion planning? And what sort of parallel computers will be particularly useful?

1.2 Motion Planning

The field of motion planning encompasses applications as diverse as computing protein interactions, creating assembly plans for complex manufactured products, deriving humanoid walking gaits, plotting dynamic aircraft trajectories, and driving ground vehicles through rough terrain or on busy highways. Generally, the problem is to plan coordinated motions of objects that consider dynamics, constraints on time, the need to obtain new sensory information during motion, and any other factor that affects the outcome of the motion. Motion planning therefore encompasses a broad range of problem domains, requiring a range of solution methods.

An investigation of parallel approaches for all motion planning problems is too broad a scope for this thesis. We restrict our inquiry to robots with few degrees of freedom, where dynamics must be considered, and solutions must be found in real-time. This applies both to air and ground vehicles, including indoor and outdoor rovers, domestic helper robots, and road vehicles. In particular, we will demonstrate our work on an autonomous passenger vehicle.

Autonomous passenger vehicles sophisticated enough to drive safely and reliably in any traffic condition would be of great benefit to society. They would decrease traffic accidents, lower costs in the transportation industry, decrease congestion and commute times, and grant independence to millions of people who are currently unable to drive because of age or infirmity.

According to the National Highway Traffic Safety Administration, approximately two million vehicle crashes occurred in the US in 2007[2]. The vast majority of accidents are estimated to have driver error as a critical reason for the crash. For example, 20% of crashes involved inadequate surveillance of the roadway by the driver, and 10% involved a distraction within the vehicle. The widespread deployment of autonomous passenger vehicles could significantly reduce the number of crashes, and a competent motion planner is necessary for the success of autonomous passenger vehicles.

1.3 Parallel Computers

Performance increases for sequential computers are expected to remain slight for the foreseeable future. Processor clock speeds have not increased at a substantial pace in over five years. Figure 1 shows the clock speeds of CPUs sold by Intel[16, 15, 66] since 1993. In 1993, the Pentium® processor was introduced, which marked an era of regular and substantial clock speed increases. By 2005, the fastest Pentium 4® processor reached 3.8 GHz, equivalent to a doubling in processor clock speed every two years for twelve years. The last five years, by contrast, have not seen a significant increase in the maximum clock speed of any processor sold.

Further increases in clock speed became difficult to achieve due to the increasing power demands and the heat generated at higher clock speeds. Performance increases have mainly been achieved since then by other innovations in processor architecture, chief among them being multi-core processing. All new desktop processors available from Intel contain at least two CPU cores, with the exception of processors in the very lowest budget segment.

Intel has been able to increase the number of cores on chips due to improvements in manufacturing technology that have allowed increasing numbers of transistors to be fit on a chip. Gordon Moore, a co-founder of Intel, predicted in 1975 that transistor density on computer chips would double approximately every 2 years[14]. This prediction has held true and become known as Moore's Law. According to Intel researchers[39], Moore's Law is expected to hold true for at least the next decade, resulting in computer chips with many billions of transistors and tens or hundreds of cores. Figure 2 shows the number of transistors actually put onto each processor sold by Intel. The wide variation is explained by the fact that the number of transistors on a chip is a significant factor in its cost. The observation that processor clock speeds have stopped increasing while transistor counts have continued to increase is important to establishing the motivation for our investigation of parallel algorithms. In the introduction to "Past, Present, Parallel", a review of parallel computers available in 1991[90], the editors opined that

> Since their invention in the 1940s, computers have been built around one basic plan: a single processor, connected to a single store of memory, executing one instruction at a time. Now, at the beginning of the 1990s, it is clear that this model will not be dominant for much longer. Parallelism – the use of many processors together in a single machine to solve a single problem – is the future of computing.

Not long after, sequential computer performance increased much more quickly than the authors likely anticipated. Research into parallel processing for robotics in the 1980s and early 1990s required considerable implementation effort to run on exotic parallel computers such as the MasPar and Connection Machine, both of which are now obsolete. As long as the power of sequential computers was increasing, it did not make sense to invest research effort in robotics into parallel algorithms on these large, expensive, and power-hungry machines. However, the predictions made in "Past, Present, Parallel" have finally been realized: normal CPUs are sold in multi-core configurations, and inexpensive parallel computers that have hundreds of cores, in the form of GPUs, are now widespread and fit easily into laptop computers.

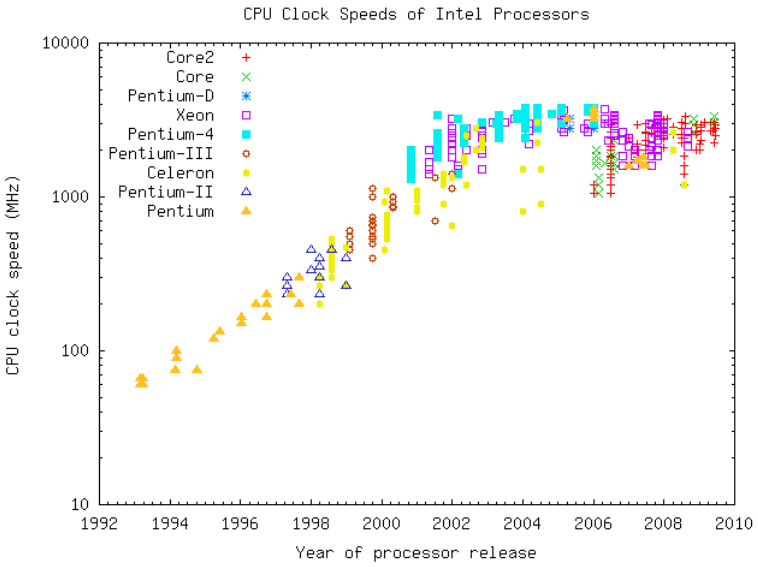


Figure 1: Log plot of CPU clock speeds of desktop and server processors sold by Intel from 1993–2008[16, 15, 66].

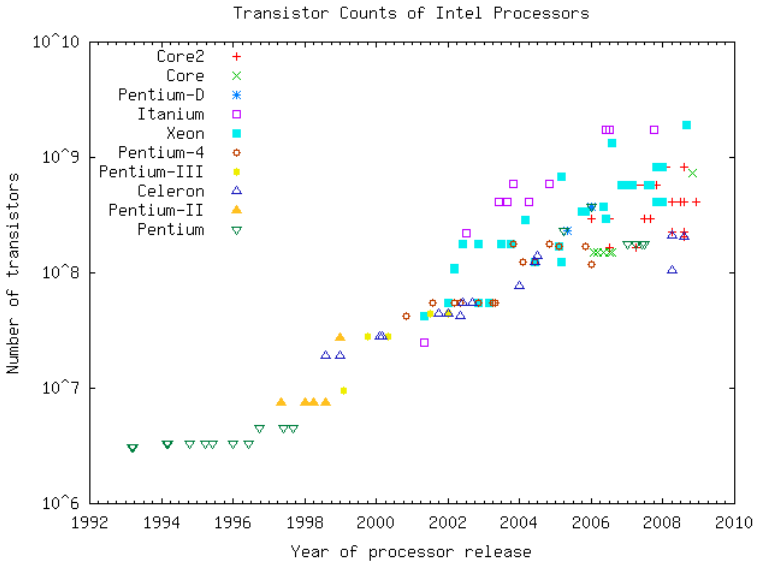


Figure 2: Log plot of numbers of transistors in desktop and server processors sold by Intel from 1993–2008[66, 15, 66].

1.4 Thesis Statement

One of our primary aims is to create a motion planner that can plan high-speed maneuvers for robots in complex environments. The thesis we propose to investigate is:

> *Parallel algorithms can improve the speed and quality of motion planning, enabling otherwise unattainable levels of performance for real-time applications.*

By the *speed* of motion planning we mean:

- The latency of the motion planner from reception of input to completion of its output.

By *performance* we mean:

- The planner's ability to plan complex maneuvers through complex situations, and at high velocities. Hence increased planner *performance* is dependent on increased planner *speed*.

By *real-time* we mean:

- The planner must be guaranteed to produce a plan within a deadline. Planning may be divided into *hard* and *soft* real-time responsibilities. For hard real-time, the deadline must be met or the planner is considered to have failed, just as though its plan had led the robot to collide with an obstacle. A vehicle driving at high speeds among moving obstacles requires a planner that can satisfy hard real-time deadlines. With a soft real-time discipline, it is desired but not critical that plans be produced quickly. A vehicle driving at low speeds in a static environment requires only a soft real-time planner. We propose to investigate parallel algorithms for both soft and hard real-time requirements.

By *quality of plan* we mean:

- The fidelity of the plan to the actual abilities of the robot.
- The expressiveness of the plan; i.e. can we express a complex desire and expect the motion planner to return a solution optimized to satisfy those desires?

In the next section we look at the parallel computing abilities of contemporary graphics processing units (GPUs) and how we intend to use them to test our thesis statement.

2 Overview of the GPU Architecture

Multi-core graphics processors(GPUs), developed most notably by Nvidia and AMD, are very different from CPUs. While both GPUs and contemporary CPUs are multi-core computers, and both are equally capable of solving decidable problems, they are designed to solve different types of problems quickly.

In this section we describe the major architectural features of the GPU, and most importantly, why some of the peculiarities of the GPU architecture will continue to be common, so that developing algorithms that take advantage of them is likely to be a worthwhile effort. We first contrast the two architectures before delving into the details of the GPU.

2.1 Major Differences between CPUs and GPUs

Contemporary personal computers are typically available with Intel or AMD processors containing between two and six traditional processor cores. These cores each typically run one sequential thread of execution at a time, and devote large amounts of chip space to running that single sequential thread of execution as quickly as possible. Speculative execution, branch prediction, a provision of excess functional units, analysis of instruction-level parallelism, and code translation are a few of many optimizations developed to squeeze performance out of a sequential thread of execution. Each of these consumes chip space. The functional units are designed to compute complex operations quickly, with diminishing returns in performance for each additional transistor. As opportunities to increase performance in sequential execution by adding transistors have dwindled, chip designers have resorted to utilizing the space available for additional transistors on a die by adding copies of the entire CPU.

GPUs also use many cores, numbering in the dozens or even hundreds. In contrast to contemporary CPUs, GPUs make no effort to execute any single thread quickly. Rather, they attempt to maximize the aggregate throughput of many threads. Hundreds or thousands of threads running the same algorithm must be available to execute in parallel for this strategy to be effective.

Figure 3 illustrates the broad design features that CPUs and GPUs incorporate to reach their respective performance goals. Whereas the CPU allocates large numbers of transistors to on-chip cache and control functions, the GPU allocates a relatively small number of transistors to cache and control, leaving the majority of chip space available for arithmetic logic units(ALUs), which actually compute the results. To achieve the maximum theoretical benefits requires an understanding of the architectural features of the processor.

2.2 Chip Architecture

Figure 4 illustrates the architecture of the Nvidia GPU. The processor is composed of a variable number of *multiprocessors*, each of which is composed of exactly eight *scalar cores*. Each scalar core can execute one instruction per clock cycle, though complex instructions such as division and built-in transcendentals may take longer. The eight scalar cores within each multiprocessor share a small cache of programmer-addressable local memory and each scalar core has a large number of registers to support the private storage needs of hundreds of threads. The multiprocessor has just one instruction issue unit, so each of the eight scalar cores execute the same instruction at the same time, albeit on different data. Multiprocessors can only share data by storing to and reading from the global device memory. Higher performance GPUs are obtained mainly by increasing the number of multiprocessors on the chip.

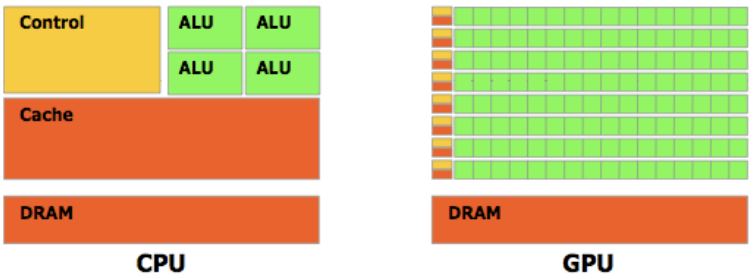


Figure 3: Comparison of transistor allocation to functional components in traditional CPU architecture vs. Nvidia GPU, reproduced from [68]

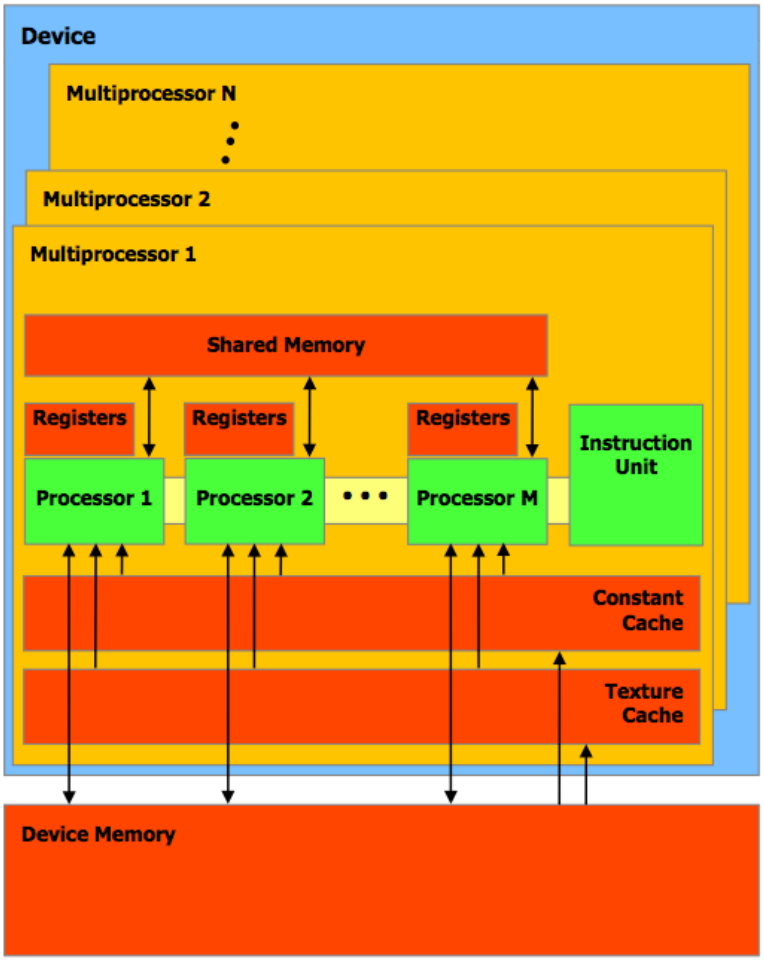


Figure 4: Architecture diagram of nvidia GPU, reproduced from [68]

2.3 Compute Unified Device Architecture

Early tools for employing graphics processors in general-purpose computing, an effort broadly termed General-Purpose GPU(GPGPU), were difficult to use and came with many restrictions, requiring the programmer to frame their algorithms as special programs for shading pixels, with debilitating restrictions on access to data and especially a lack of ways to synchronize between different operations. GPU manufacturers have been developing their graphics processors into more capable, general, and accessible parallel computing platform. One of these efforts is Nvidia's Compute Unified Device Architecture(CUDA).

Several multi-core processors share important architectural features with the Nvidia GPU, so we shall describe the particulars of the Nvidia architecture as concretely as possible in order to make it easier to understand, and then show the similarities it shares with other multi-core platforms.

CUDA consists of syntax enhancements to C++, with an accompanying compiler and a runtime library. It works beneath the graphics layer and leaves the programmer to program in C with a few simple additions to the language to utilize the parallel features of the processor. AMD, another supplier of GPUs, is developing a similar effort using their own Stream SDK[1]. Additionally, a portable language for heterogeneous computing using CPUs and GPUs known as OpenCL[36] is newly available. At the moment, CUDA is the most accessible and the most advanced, so we will be demonstrating results using CUDA.

In CUDA, threads are organized into *blocks* which may contain up to 768 threads. All threads in the same block may synchronize with each other using a barrier operation, but threads from different blocks may not synchronize. Each of the threads within a block is addressed with a unique three-dimensional index (x, y, z). Threads can retrieve their indices using a hardware instruction. Blocks are organized into a grid, and like threads, are uniquely numbered by the scheduling hardware. Exactly one grid of thread blocks is run on the GPU per *kernel* invocation from the host CPU. Figures 5 and 6 show the core syntax of CUDA. Code running on the CPU invokes parallel functions called *kernels* to be run on the GPU. Each thread is designated to handle a small piece of the work. The one-dimensional convolution code shown in these figures illustrates the use of the programmer-controlled cache memory, also called *shared memory*. It also demonstrates the synchronization of threads within a block. The function shown in Figure 5 runs on the GPU. Threads with the same ***blockIdx*** can access the same shared memory locations and synchronize with the built-in __syncthreads() barrier primitive. Since each element of the kernel and image arrays will be accessed many times and by different threads, they are loaded into the shared memory. Then, each thread computes one element of the result and stores it. Figure 6 shows the code run on the host CPU. This is normal C++ code that runs in the usual way. The programmer must explicitly schedule data to be copied between buffers allocated on the host and GPU. The special $\lll\ggg$ syntax is used to invoke a GPU "kernel" function with the grid and block size layout within the brackets.

```
// 1D convolution with CUDA.
// This function runs on the GPU.
__global__ void convolve1d( float *image, int imgsize,
    float *kernel, int ksize, float *result ) {
    const int ksize2 = ksize / 2;

    // Index of this thread block's portion of the input image.
    const unsigned int img0 = blockIdx.x * (blockDim.x - ksize2*2);

    // Cache entire kernel and portion of image in block-shared memory.
    __shared__ float kcache[ MAXKSIZE ], imgcache[ MAXTHREADS ];

    // Load image cache, one thread per element, including padding.
    const int imgidx = img0 + threadIdx.x - ksize2;
    if( imgidx < 0 || imgidx >= imgsize ) imgcache[ threadIdx.x ] = 0;
    else imgcache[ threadIdx.x ] = image[ imgidx ];

    // Load kernel from global memory into shared cache.
    if( threadIdx.x < ksize ) kcache[threadIdx.x] = kernel[threadIdx.x];

    // Synchronize all threads in block; ensures data loaded into cache
    __syncthreads();

    // One thread responsible for each cell of result.
    // Some threads do nothing here; they exist only to load data above
    if( threadIdx.x >= ksize2 && threadIdx.x < blockDim.x - ksize2 ) {
        float r = 0;
        for( int i = -ksize2; i <= ksize2; ++i )
            r += kcache[i + ksize2] * imgcache[i+threadIdx.x];
        result[img0 + threadIdx.x - ksize2] = r;
    }
}
```

Figure 5: First part of the CUDA code sample showing a parallel one-dimensional image convolution operation. This "kernel" runs on the GPU.

```
int main( int argc, char *argv[] ) {
    // input: img, imgsize, kernel, ksize

    const int blocksize = MAXTHREADS, cells_per_block = blocksize - ksize - 1;
    const int gridsize = (imgsize + cells_per_block - 1) / cells_per_block;

    // Allocate room on the graphics card to store a copy of the image.
    void *img_device; cudaMalloc( &img_device, imgsize*sizeof(float) );
    // Copy img from host (CPU) to device (GPU).
    cudaMemcpy( img_device, img, imgsize * sizeof(float), cudaMemcpyHostToDevice );
    // Similarly: allocate space for result and kernel on GPU, copy from host...
    void *result_device; cudaMalloc( &result_device, imgsize*sizeof(float) );
    void *kernel_device; cudaMalloc( &kernel_device, ksize*sizeof(float) );
    cudaMemcpy( kernel_device, kernel, kernelsize * sizeof(float), cudaMemcpyHostToDevice );

    // Call CUDA convolution code.
    // gridsize blocks will be run, of blocksize threads each.
    convolve1d<<< gridsize, blocksize >>>
        ( img_device, imgsize, kernel_device, ksize, result_device );

    // Allocate space on the host for the result, and copy from GPU.
    float *result_host = malloc( imgsize*sizeof(float) );
    cudaMemcpy( result_host, result_device, imgsize * sizeof(float), cudaMemcpyDeviceToHost );
    // output: result_host
}
```

Figure 6: Second part of the CUDA code sample showing a parallel one-dimensional image convolution operation. This code runs on the "host", or CPU, and interacts with kernel code (Figure 5) through the CUDA API.

2.4 Features Common to GPUs

In the previous section we described the Nvidia GPU and the CUDA programming model. In this section we argue that the interesting features of the Nvidia GPU are also present in other platforms, such as AMD's Radeon graphics processors and Intel's upcoming Larrabee[82], and that their architectures are driven by the same design forces. The Nvidia GPU with its CUDA language extension is currently the most advanced and accessible platform for general-purpose parallel computing, but efforts made to develop algorithms and approaches will generalize to future platforms.

2.4.1 Data-Parallel Operations

The performance achieved by GPUs comes from their devotion of large numbers of transistors to computational rather than control elements, as illustrated in Figure 3. The control elements are responsible for instruction decoding, handling branches in the code, calculating addresses for operations on memory, analyzing code for instruction-level parallelism, and more. In order for the GPUs to keep these computational elements, or ALUs, supplied with work, the control elements must accomplish more with less. The simplest way to do this is by having each instruction cause the same operation to be performed simultaneously on multiple data elements. This is easy to do in many graphics applications, where rendering an image involves performing the same operations on many pixels.

An operation that performs the same computation on several pieces of data at once is termed *data-parallel*. Data-parallelism stands in contrast to *task-parallelism* where multiple threads of control may simultaneously perform entirely different functions. In order to gain the benefits of the GPUs for motion planning, it is necessary to develop algorithms that can use data-parallel operations.

Data-parallel computers typically organize data elements into fixed-size vectors. In CUDA for example, data-parallel operations are performed on vectors of 32 elements. Figure 7 illustrates. Part (a) of the figure shows a normal program using scalar variables. Part (b) shows the same program running simultaneously on several copies of the variables organized into vectors, denoted `<v>`. Control flow is accomplished by creating a new vector variable that contains the results of a logical test performed on independent vector elements. Computations for elements corresponding to positions that tested false are still performed, but their results are not stored. With appropriate hardware support, the program could skip lines 3 or 4 entirely if `mask1` had all the same value.

In CUDA, the operations appear to be computed by independent threads. However, threads are grouped into *warps* of 32 elements. Threads within a warp execute the same instruction at the same time, but warps may run completely independently of each other. Figure 8 illustrates this idea with an abbreviated warp of 8 elements. Each vector element is represented as a local variable for one thread, but threads in the same warp run the same instruction. When the code branches, only the threads that take the "true" branch run, simultaneously. When those threads reach the end of their branch, the threads that take the "false" branch run until the branches merge, when all threads can run again. If all threads were to take the same branch, then no time would be wasted

```
1: a = C[d];
2: if( a == 0 )
3:   d++;
4: else e--;
5: f += b * e;
```

(a)

```
1: <a> = C[ <d> ];
2: <mask1> = (<a> == 0);
3: <d> = ( <mask1> & (<d>+<1>))
       | (~<mask1> & <d>);
4: <e> = (~<mask1> & (<e>-<1>))
       | ( <mask1> & <e>);
5: <f> += <b> * <e>;
```

(b)

Figure 7: Comparison of (a) a program using familiar scalar variables with (b) explicit use of vector-parallel operations. Each operation is performed element-wise on the vector variables, and constants are also vectors. Note that identifier `C` is an array, but not a vector variable.

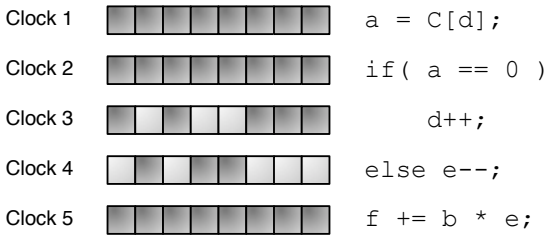


Figure 8: Serialized execution of threads in an eight-thread warp on the GPU

with the alternate branch, an improvement over the mask-vector concept of Figure 7.

Modern CPUs typically have vector instruction sets and vector registers in addition to their traditional scalar instructions and registers. For example, the Intel SSE and PowerPC Altivec instruction sets can operate on vectors of 4 elements. However, the CPU vector instructions cannot perform operations on vector elements conditionally, nor can they use a vector element as an index for a memory access. That is, traditional CPU SIMD instructions lack support for the instruction on line 1 of Figure 7(b), where the vector elements of `<d>` are used as indices into the array `C[]`. They also lack support for line 2, where the results of a conditional expression evaluated on each element of a vector are stored in another vector. Nor do they have an efficient built-in way of performing the mask operation expressed in lines 3 and 4.

Note that CUDA's warps are equivalent to the explicit use of operations conditioned on mask registers shown in Figure 7. AMD GPUs, IBM's Cell[73], and Intel's Larrabee all depend for their high performance compared to CPUs on the conditional vector-parallel operations just described. The motivation for vector-parallel operations from a performance perspective is that they allow more work to be done with the same number of clock cycles and transistors devoted to instruction decoding, out-of-order instruction issue, and address translation. The challenge for the software developer and algorithm

designer is to find ways to compose algorithms using vector-parallel operations. To achieve maximum performance using vector operations, we must be careful to organize our computation to ensure that all elements of each vector branch in the same direction as often as possible.

2.4.2 Memory Architecture

In this section we contrast the memory architecture of contemporary GPUs and CPUs and their effect on program performance. We then attempt to anticipate which aspects of the GPU architecture are likely to persist.

In traditional CPU architectures, cache misses are extremely detrimental to performance. All execution stops until the desired information is retrieved from memory further away from the CPU. The most common way to mitigate this problem is by using larger caches. Another way is to build hardware that can switch instantly to another runnable thread when one thread is stalled. For example, Intel's Hyper-Threading feature allows one secondary thread to be run simultaneously with the primary thread. When the primary thread is forced to wait, the secondary thread may continue running. This can provide a speed boost to some applications.

GPUs use a comparatively small amount of cache memory, and rely on the programmer to write algorithms that use hundreds of threads. The GPU schedules threads in hardware at a fine grain, so that when a thread stalls waiting for a memory access to complete, another thread can be run immediately. If enough threads are available, then the ALUs can be kept working even when threads must frequently wait for memory operations to complete. This is the major rationale for the GPU architecture - if an algorithm can muster enough threads to keep the ALUs busy, then the work done may be maximized, even though individual threads may have a higher latency than they would on a sequential processor.

Memory bandwidth and latency limitations require many operations to be performed on each piece of data loaded from memory in order to achieve peak performance. Algorithms that run best on the GPU therefore must intersperse memory accesses with a significant amount of work. Algorithms that frequently permute global memory are not suitable for the GPU.

CPUs use a cache memory structure that automatically stores the values in recently-accessed memory close to the processor in anticipation that they will be used again soon. In place of these, the *shared memory* of the Nvidia GPUs requires the programmer to explicitly determine when data will be transferred between the large, global memory from the small, fast memory near the processor.

Memory bandwidth is the aggregate amount of data that can be transported between main memory and the processor per unit time. GPUs have a considerably higher memory bandwidth than CPUs. The Intel Core 2 Duo, for example, has a theoretical bandwidth of 8.5 GB/s, whereas the Nvidia GTX 260 has a memory bandwidth of 112 GB/s. The higher memory bandwidth of the GPU comes as a result of a wider data path. In current designs, the peak bandwidth is only achieved when vector operations access consecutive memory locations. The higher bandwidth and lack of cache memory are complementary tradeoffs, in that multiple loads of the same data from main memory can be tolerated because of the high bandwidth available. Algorithms that

must operate on large amounts of memory are good candidates for the GPU.

Graphics applications demand high memory bandwidth because of the numerous large images used to texture 3D objects, the large numbers of triangles used to represent the shapes of the objects, the several stages in the rendering process and the dozens of frames that must be rendered per second in games, their major market. Since high memory bandwidth is vital to graphics, future generations of GPUs are likely to retain this advantage over CPUs. However, the reliance on a programmer-controlled shared memory in favor of an automatically managed cache may not persist. The designers of Intel's Larrabee, for example, chose the latter.

2.4.3 Summary

We have described some of the major architectural features of GPUs that determine which algorithms they run most efficiently. These are the SIMD operations with a related threading model, the high bandwidth and high latency of memory operations, limited communications abilities between multiprocessors, and small, manually allocated cache memory. We believe that with the possible exception of the latter, these architectural features are likely to persist through future generations of GPUs and that therefore they should be considered when designing and assessing algorithms that are likely to retain their value.

3 Related Work

In the following section we provide the reader with a survey of alternative parallel programming models, and of related work in motion planning, both sequential and parallel.

3.1 Parallel Computer Architectures

We briefly describe a few alternate computer architectures for parallel computation and the types of problems they are intended to address.

3.1.1 Distributed Systems

While GPUs accelerate data-parallel operations, a more general class of parallel systems are distributed systems. Distributed systems use large clusters of computers to solve large problems that can be broken into smaller chunks. The largest parallel computer systems are distributed systems composed of many smaller computing nodes. For example, the smallest supercomputer sold by Cray, the CX1, resembles a network of normal servers. It uses Xeon processors, with the largest configuration offering 6 computer "blades" connected with Infiniband network adapters, and each with dual quad-core Xeon processors.

Google's MapReduce[23] application is a well-known example of a software framework for programming distributed systems computing the results of massively parallel problems.

Some robotics applications use ad-hoc distributed approaches. Robotics applications typically have a profusion of software processes, handling a variety of perception and planning tasks, as well ancillary functions such as a user interface. In the DARPA Urban Challenge[22], many entrants[53, 71, 92, 78, 65, 9, 4, 61, 13] used distributed computing systems.

We will not address distributed approaches to parallel programming for motion planning in this thesis. While distributed systems can increase throughput for large computations, just as the SIMD parallelism employed by GPUs can, the latency of communications between computing nodes can make meeting real-time requirements of on-line motion planning more challenging.

3.1.2 Exotic Multi-core Platforms

In this thesis we are concerned with exploiting parallelism using SIMD operations such as those used by GPUs. We are interested in GPUs because they serve a large niche market that we expect will continue to be a rich source of fast and cheap parallel computers. However, other niche markets are also served by processors that use multiple cores on a single chip, but without the use of SIMD instructions. For example, the Tilera Corporation TILE64 processor[7] has 64 general-purpose cores on a single chip with a high-bandwidth interconnection between them. One application of this style of many-core computer is for network switches that perform extensive analysis on streams of packets. While such processors likely could be used to accelerate motion planning, their present and likely future inaccessibility renders them unsuitable for motion planning at this time.

Vector operations inherently require some memory access locality from the problem. There are important applications, however, that have very little memory locality. Parallel sorting, operations on linked lists, random access into hash tables, and operations on large sparse matrices are a few examples. The Cray XMT[20] architecture uses processors that run 128 threads simultaneously. The 128 threads share a single set of functional units to hide memory latency. The processors are distributed, but share a global memory space, and are connected by a very fast network fabric. The XMT architecture is optimized to maximize throughput when random access is required to very large data sets. Motion planning problems typically do not deal with such large data sets, so this form of parallel computer is not suitable for solving motion planning problems.

3.2 Complexity Analysis in Parallel Programming

A full introduction to complexity analysis for parallel programming, that is, the theoretical comparison of running times for parallel algorithms, is beyond the scope of this document. However, we introduce a few important terms. The reader can consult[91] for a deeper introduction.

Sequential algorithms perform their task one step at a time, while parallel algorithms may execute multiple steps at once. The time taken by a sequential algorithms is simply proportional to the number of steps it must take. For sequential computers, the theoretical time taken is usually a good predictor of the time actually taken by an

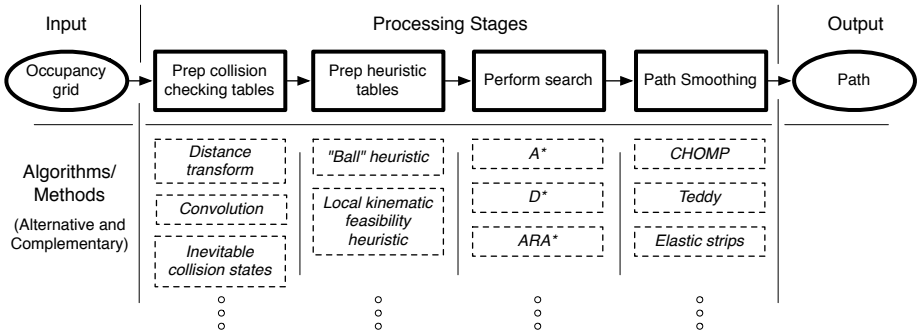


Figure 9: Flow of A*-based pathfinding in unstructured environments.

implementation of the algorithm. Since there are so many ways to organize a parallel computer, it can be difficult to choose an appropriate model to predict the performance of a parallel algorithm. The *work-depth* model focuses on the number of operations performed by an algorithm and the dependencies between these operations, without regard to a parallel machine model. The *work* W of a parallel algorithm is the total number of operations it executes. The *depth* D is the length of the longest sequence of dependencies among the operations. We define $\mathcal{P} = W/D$ as the *parallelism* of the algorithm. For a sequential algorithms $W = D$ so $\mathcal{P} = 1$. The greatest degree of parallelism is achieved when all operations are independent, that is $D = 1$. A parallel algorithm is called *work-efficient* if it performs at most a constant factor more operations than a sequential algorithm. For tasks where results must be computed within a short time frame, such as the real-time motion planning problem we are concerned with, a work-efficient algorithm is not necessarily preferred, if a work-inefficient algorithm with a smaller depth is available. Later in this proposal we will present a single-source shortest path algorithm that is work-inefficient but has low computation time in practice.

The concepts of work-efficiency and depth will be used later in this document.

3.3 Overview of Vehicle Motion Planning

A basic schematic of a motion planning system is shown in Figure 9. The world representation, in this case a simple occupancy grid, is processed into various forms. For example, tables used for collision checking, such as a distance transform are computed first. If a sequential search algorithm such as A*[38] or a variant such as D*[86] is used, then heuristic tables are constructed, such as a single-source shortest-path for a simplified vehicle or search space. Once the collision checking tables, heuristic tables, and any other preparations are done, a rough path is found using a planning algorithm such as A*. This is often called the *global planning* step. Due to the curse of dimensionality[8], the search algorithm is typically unable to plan a path through a high-dimensional state space representing the robot. This means either that the globally planned path found cannot actually be followed by the robot, or the path is suboptimal with respect to the cost function. A further path smoothing step is usually required to

find a feasible path in a neighbourhood of the initial rough path. A typical approach is to use the global path as input to a *local planner* that will find a kinematically, or better, dynamically feasible path in the neighborhood of the global path. Effective approaches are to use the global path as the initial guess in a trajectory optimizer[77, 41], or simulate the vehicle following a variety of perturbances of the global path using a geometric path tracker[63] such as pure pursuit[18] or others[85].

In this document we propose to investigate parallel algorithms to accelerate various stages of the motion planning pipeline.

3.4 Related Work in Parallel Algorithms

3.4.1 Parallel Algorithms For Motion Planning

Motion planning can be formulated as a search in a discrete graph embedded in a continuous space in which a robot can move. The two main phases are the construction of the graph and the search through the graph. The are many ways of constructing such a graph, many of which can be done in parallel. As an introduction, Henrich published a review[40] of research in parallelism for robotic motion planning. For example, parallel algorithms exist to compute Voronoi diagrams[79], which can then be used to construct a roadmap of navigable free space between obstacles[69]. Amato and Dale demonstrate near-linear speedup of parallel generation of nodes and edges in a probabilistic roadmap approach[3], while performing PRM queries sequentially. Many other methods exist to build discrete search graphs embedded in a continuous space. See LaValle[49] for an extensive review. Parallel single-source shortest path algorithms can be used to search through a graph[88, 89, 60, 19]. Parallel wavefront propagation algorithms[94] can be used when edges have uniform cost. Genetic search algorithms are easy to implement in parallel and can be used to find suboptimal paths through a graph, as for multiple robots in [83]. The A* algorithm used in graph search admits some parallelism in the expansion of child nodes. Mahanti and Daniels[56] describe Iterative-Deepening Parallel Search, a parallel search algorithm based on the simpler IDA* algorithm.

While embedding a discrete graph in the continuum can be an effective formulation, a gradient descent method directly in the continuum can also be used. However, motion planning solutions using gradient descent are vulnerable to being trapped in local minima. To address this problem, Barraquand and Latombe[6] proposed the Randomized Path Planning(RPP) method, which uses gradient descent alternating with random jumps in the search space. Challou et al.[11] report a parallelization of RPP where copies of the algorithm run on independent parallel processors, with the first processor to find the answer notifying the rest that they can stop.

The state lattice approach we describe in the next section combines graph embedding and gradient descent methods by using the a gradient descent method to drive short path segments that are composed into a regular graph structure embedded. In contrast to randomized methods, this graph structure allows an efficient and systematic search of the space. We shall see that the state lattice approach is amenable to parallelization on the GPU as well.

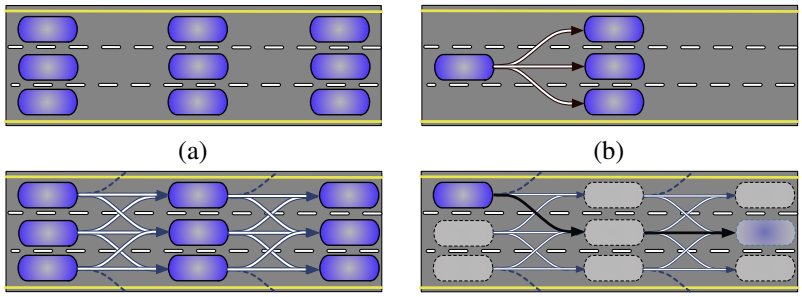


Figure 10: A state lattice on a straight road (a) uniformly samples states along and across the road to construct a search graph. Each state (b) has a finite number of outgoing action edges which (c) are repeated in a uniform pattern from each state to create a lattice. Actions leading off of the road are clipped. A search algorithm then (d) searches the lattice to find the best path from the current vehicle position to a future vehicle position further along the road. By construction, the path can be followed by the vehicle exactly as planned.

3.4.2 State Lattices

A common approach to planning for robotic vehicles, especially in earlier work, is to use a crude model of the vehicle kinematics and its environment to create a roughly optimal planned path. The path may then be followed in simulation by a local planning algorithm using a higher-fidelity model including more accurate kinematics and dynamics, and tested for safety. In [51] for example, a crude model of a vehicle was used to generate a rough plan using A* search. The path was followed using a pure-pursuit path tracker[18] with a higher-fidelity model, which was checked for safety before being run on the vehicle. Two problems with this approach became evident: first, if the vehicle approximation is an underestimate of the vehicle's true capabilities, then the global path can be much more expensive than the true optimal path. Second, if the planner overestimates rather than underestimates the vehicle's abilities, then the local planner will be unable to find a way to follow the global path at all, leading to a stalled vehicle.

As an attempt to bridge this gap, Kelly and Pivtoraiko[74] introduced the state lattice, a means of embedding a discrete graph into the continuous robot configuration space, where vertices in the graph represent robot states, and edges joining vertices represent trajectories that satisfy applicable constraints on the robot's motion. The benefit is that plans generated via a search through a state lattice are closer to the robot's actual abilities than prior approaches. The disadvantage is that it typically requires more computational resources to generate the lattice. Figure 10 illustrates a state lattice for an robotic vehicle travelling on a road. The state lattice approach has been shown to be effective for planetary rover navigation applications[74], a passenger vehicle[30, 29], and a wheeled indoor assistant robot[81]. However, lattice planners have not been applied successfully to vehicles driving in traffic, avoiding dynamic obstacles and subject

to complex performance constraints. We intend to explore the use of lattice planners in on-road driving scenarios. For on-road driving scenarios, use of the regular lattice structure assumed by prior works would cause an intractably large increase in the density of states and the size of the action set required to plan fine motions among moving vehicles on curving roads. We will investigate how a smaller state lattice may be warped online to better fit the local structure of the space. We shall also investigate how to integrate time and velocity dimensions into the lattice so that dynamic obstacles may be accounted for by the planner. As we shall see, the additional computational demands of on-road driving will require parallel processing to meet real-time deadlines. We propose that the SIMD operations provided by GPUs are well-suited to fulfilling this need.

We will describe state lattices in more detail in Section 5.4, where we also describe our initial efforts in devising a parallel state planner for on-road vehicles. We now review related work in parallel algorithms of use in the motion planning stack, including work aimed specifically at manycore platforms such as GPUs. These include parallel distance transforms, single-source shortest-path algorithms, collision checking algorithms, and a kinematic analysis useful for constructing search heuristics.

3.4.3 Planning In Dynamic Environments

Planning motions for robots in dynamic environments is a difficult problem. The additional computational cycles available with parallel computers may enable new techniques in planning among dynamic obstacles. A compelling application of a planner that can consider dynamic obstacles would be for a robotic vehicle driving on roads in dense traffic.

Erdmann and Lozano-Perez[27] introduced the idea of a configuration spacetime in planning motions for multiple bodies considering constraints on velocity. They formed a discretization of the configuration spacetime and computed all valid poses for a robot over a time range. They then searched a graph connected by regions where valid configurations overlapped in consecutive spacetime slices. Their solution did not consider dynamic or kinematic constraints and could only search for feasible, but not optimal paths.

Fraichard and Asama defined an Inevitable Collision State(ICS)[32] as a state s of a robotic system which is guaranteed to result in a collision with an obstacle no matter what action is taken. The ICS notion is an extension to static collision checking, which simply tells whether a robot is *currently* in collision with an obstacle. Martinez-Gomez and Fraichard[57] applied GPUs to the problem of checking inevitable collision states for a robot moving among dynamic obstacles in a planar domain. Their technique depends on the identification of a set of evasive and imitative maneuvers, such that it is reasonable to assume that if no maneuver in the set can avoid all obstacles, it is not possible to do so. The benefit of the ICS is that it can help to eliminate states early in the search, thus preventing their descendant states from being considered.

Kushleyev and Likhachev[47] developed a lattice planner that includes time and velocity dimensions but they failed to constrain actions to end at intelligently sampled lattice points, creating an explosion of states.

Tompkins[87] describes the Incremental Search Engine (ISE) algorithm, which is

designed to search resolution-optimally in state spaces with independent state parameters, while simultaneously supporting an approximate examination of dependent state parameters. While independent state parameters are constrained to discrete states, dependent parameters may take on any value. Nodes with the same values for the independent substate but different dependent states are culled if they are within the same "resolution-equivalent" bucket. As an example, robot position (x, y) can be taken as the independent state dimensions, with time as the dependent state dimensions. Tompkins does not consider application domains where kinodynamic constraints must be obeyed. Hybrid A*[25] buckets independent states in a similar way, without constraining actions to end directly on lattice grid points. A number of similar attempts to manage continuous states when using a discrete search algorithm can be found in the literature.

For the special case of on-road driving applications, researchers in the automotive community have identified bounds on motions and spacing for successful lane changing, merging[42, 44, 70], and emergency manuevers[35]. These works are special-case geometric analyses that are not formulated within a general motion-planning framework.

Early work in planning amongst dynamic obstacles for real-time applications tended towards reactive techniques, due mainly to the high computational demands compared to the available computing power. Nearness Diagrams[62] and the Dynamic Window approach[31] are two examples. These approaches were useful for robots with limited computational power but cannot scale up to take advantage of the capacity of modern parallel machines.

Trajectory deformation approaches attempt to continuously deform a path or trajectory in order to improve its utility. Brock and Khatib describe the Elastic Strips method[10] which deforms a path according to virtual forces, representing the robot dynamics as internal forces, and repulsive fields formed by obstacles as external forces. Their approach does not take moving obstacles into account, and must begin with a valid path. Fraichard and Delsart[33] describe `Teddy`, a trajectory deformation algorithm that extends the potential fields formed by obstacles along the time axis. Their method samples fixed points on the robot and uses the Jacobian of the point location with respect to the configuration variables to allow an articulated robot to avoid obstacles. `Teddy` requires an independent global motion planner to provide a valid path as input. Ratliff et al. present CHOMP[77], similar in spirit to `Teddy`, but able to take an invalid path as input, and better able to induce control changes at points earlier in the path in order to satisfy constraints on points later in the path, resulting in smoother paths.

Trajectory deformation algorithms appear to be easy to parallelize with SIMD operations, and in an obvious way. They perform the same operations on many points defining the path, and on points sampled over the boundary of the robot to interact with the obstacle force fields. To construct the obstacle force fields requires a distance transform to be computed. The distance transform is then used to form a gradient to push path points away from the obstacles. Depending on the size and complexity of the environment, the distance transform may be expensive to compute. It is difficult, however, to exploit data-parallel operations to compute an exact solution to the distance transform, an issue we will visit in the next section.

Planning methods typically used for on-road driving are based on schemes for sam-

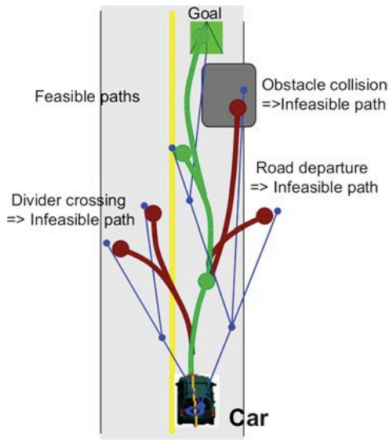


Figure 11: RRT-based motion planner for MIT's Talos, from[53]

pling simple trajectories that forward-simulate the vehicle's current pose to an endpoint somewhere within a short distance in front of the vehicle. Each trajectory is evaluated with a forward simulation of the vehicle to estimate how close it would come to obstacles and evaluate how well it meets other objectives for the vehicle behavior. Published methods from the Urban Challenge, for example, follow a range of methods within this general scheme.

MIT's Talos vehicle[53] (see Figure 11) used an RRT-based motion planner that sampled in the space of target points for the steering controller. The set of sampled plans was represented as a tree of potential vehicle trajectories. A leaf of the tree was extended by forward-simulating the entire vehicle system with the steering controller's reference point set to the sampled point. Planner efficiency was increased by heuristically biasing the sample distribution. Some sample points also controlled the velocity down to zero. The planner would run until it found an acceptable path that ended with the vehicle at a stop. The planner selected the best path, which the vehicle would then follow to a stop, unless the planner was able to generate a new plan in time. This planner cannot react quickly to sudden changes in the environment, and at high driving speeds, would run a risk of being unable to generate a new plan before the vehicle reaches the end of the old plan.

Stanford's Junior vehicle[64] sampled paths as swerves offset from a reference path defined by the center of the lane, as shown in Figure 12. This approach was sufficient for simple traffic interactions involving, for example, changing lanes to pass a single slow-moving vehicle.

With current methods and platforms, plans typically do not take into account all available information – ignoring, for example, interactions with obstacles that are beyond the vehicle's stopping distance but within its sensing range. They are also unable to create complex plans quickly, being based typically on sampling strategies with a simple structure. Where they are capable of making more complex plans, such as the Talos planner, they are not systematic and cannot guarantee that they will generate

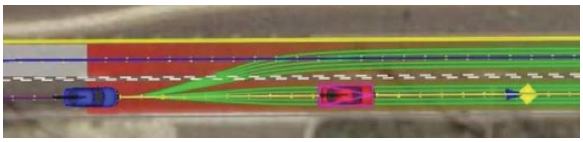


Figure 12: Lateral offsets sampled from a reference trajectory by Stanford's Junior[64]

plans within a real-time deadline, which is necessary for tasks with tight performance margins, for example high-speed driving.

Motion planning in dynamic environments is an important application that has been difficult to address due to its large computational demands. We propose that a systematic approach to searching a larger and more expressive space, such as one based on the state lattice described in the previous section, can be effected using a GPU. Data artifacts necessary to the search but independent of the search algorithm, such as the distance transform, can also be computed more quickly in parallel.

3.4.4 Distance Transform

For motion planning, the distance transform is used to test for collisions, and compute cost functions that can balance the path chosen between saving effort and risking collision by cutting too close to obstacles. It can also be used in incremental trajectory deformation algorithms to form potential functions for obstacles. The DT can additionally be used directly in cost shaping functions, and to compute generalized voronoi diagrams that are also useful in cost shaping and navigation functions[25].

Given a binary image occupancy grid $\mathcal{I}$ of $\{(x, y, v)\}$, where $v \in \{0, 1\}$, the Euclidean distance transform (EDT), first described by Rosenfeld and Pfaltz[80] $\mathcal{D}(\mathcal{I})$ is a function $d : \mathbb{R}^2 \rightarrow \mathbb{R}$, defined by $d(x, y) = \min_{(x',y',1)\in\mathcal{I}} ||(x, y) - (x', y')||$. The distance transform can be extended to arbitrary dimensions in the obvious way. Robotics applications typically use two or three dimensions, but could conceivably use a distance transform in up to four dimensions if computing potential fields in spacetime, as in[33], though a non-uniform metric would likely be required.

There are many algorithms both seqeuntial and parallel for computing the distance transform in the literature. Many of the parallel algorithms that work on images of dimension greater than one are approximations, for example the "Jump Flooding" technique of Rong and Tan[79]. The geometrical insight made by Felzenszwalb and Huttenlocher[28] has yielded the most efficient sequential algorithm. In theory it can be parallelized with depth dN for an N^d image, but it is a poor fit with vector-parallel operations. For a parallel distance transform algorithm that gives exact results and that is tuned for the GPU, the main challenge is to use its SIMD operations effectively.

An approximate algorithms using parallel operations was presented by Embrechts and Roose[26], but it used distributed rather than SIMD operations and their parallel speedup was poor.

Viitanen and Takala[93] present an approximate algorithm using vector operations, but their method only works in two dimensions on small images and where it can be assumed that all points are within a bounded distance from an obstacle.

Lee et al.[50] give an exact algorithm using SIMD operations on a hypercube parallel computer. Their method calculates results for an $N \times N$ image in $O(log^2 N)$ time but requires N^2 processors, and only works in two dimensions. It works on a hypercube computer but does not use vector instructions.

Weber et al.[94] present an approximate distance transform algorithm for the GPU with $O(N)$ depth. It is notable for operating on a parametrically deformed grid, but is limited to work in 2 dimensions.

We know of no exact distance transform algorithm that works in arbitrary dimensions and using SIMD vector operations. The geometric insight of[28] leads to an efficient algorithm for the sequential case but additional research effort is required to devise an exact distance transform using SIMD computations. We propose to investigate this problem. An efficient SIMD algorithm for the exact distance transform would accelerate numerous motion planning algorithms.

3.4.5 Single-Source Shortest Path

The single-source shortest path (SSSP) problem is to take a graph $G = (V, E)$ with edge weights $c : E \to \mathbb{R}^+$, and a designated sink vertex $t \in V$, and produce a table $s : V \to \mathbb{R}^+$ giving the cost for each vertex v of the lowest-weight path through the graph from v to t. The classic Dijkstra's algorithm [24] is inherently sequential. No general-purpose parallel algorithm known is guaranteed to accelerate SSSP on all graphs. However, special cases of graph can be addressed with a parallel algorithm, though since each parallel computer architecture is different, unique algorithms are often required for each one.

Träff and Zariolagis[89] described a parallel algorithm for general planar graphs that decomposes the graph into regions, performs SSSP on boundary vertices within the regions in parallel, and stitches the regions back together. Meyer's Δ-stepping algorithm for general graphs slides parametrically between Dijkstra's algorithm and value iteration over the entire graph. Prior attempts have been made to implement this algorithm directly on the GPU[5], but without success, due to the difficulties in adapting the algorithm to the GPU parallel architecture.

The parallel approximate distance transform described by Weber et al. [94] works on parametrically deformed grids, so it can also be used to approximate the SSSP algorithm when used on a grid. The performance of their algorithm in this case depends on the maximum number of turns that would have to be made when navigating through the grid to reach any node from the source, and the size of the disparity between maximum and minimum edge costs. It is not clear how well their algorithm would perform on grids with both very low and very high edge costs(approximating non-traversable areas).

Harish and Narayanan [37] propose an SSSP algorithm that amounts to Delta-stepping with $\Delta = \inf$, or equivalently, a Bellman value iteration over the whole grid. While this would seem to be extremely inefficient, the high memory bandwidth of the GPU and the highly structured nature of the memory accesses with this algorithm can make it competitive in some situations. They present results on random graphs. Graph diameter is a vital quantity to control for when evaluating a wavefront algorithm such as theirs, but they did not report graph diameter in their results. Since they used ran-

dom graphs, we assume the diameter was low. The diameter of a graph based on a grid is relatively high compared to the number of vertices, so their algorithm may perform poorly on graphs of interest in motion planning.

In Section 5.3 we present an adaptation of the Δ-stepping algorithm for the GPU that shows improved performance over a highly optimized sequential implementation of Dijkstra's algorithm.

3.4.6 Collision Detection

The configuration space $\mathcal{C}$ of a vehicle is the set of all possible poses of the vehicle. For a typical ground vehicle navigating on a flat surface this is the set of poses (x, y, θ). When obstacles are present, $\mathcal{C}_{obs}$ is the set of poses which may not be occupied since they collide with an obstacle, and the configuration free space $\mathcal{C}_{free} = \mathcal{C} - \mathcal{C}_{obs}$ is the set of poses which do not collide with an obstacle.

Exact geometrical descriptions of the configuration space can be composed when obstacles are polygonal. See [49] for an overview. It is practical, however, to represent obstacles as binary values in an occupancy grid.

For two-dimensional obstacles represented as a grid, a free space analysis can be done for a given orientation θ by rendering the robot shape into a matrix and performing a convolution of the obstacle grid with the robot grid. This is done once for each rigid body in the robot, and for each θ to a desired discretization. The convolution may be done directly in image space or by using a Fast Fourier Transform (FFT) in frequency space. Note that the time complexity is invariant to the complexity of the obstacles. It does vary depending on the sizes of the grids representing the free space and the robot. Kavraki[45] compares the performance of the two approaches on a CPU-based workstation. Lengyel et al.[52] used graphics hardware in a similar way to render C-space expanded polygonal obstacles and a polygonal robot over discrete ranges of object angles into a bitmap, which they then searched using BFS on a simple kinematic motion model. They did not consider kinematic constraints on robot motion.

One of the most costly computations in motion planning is collision detection[48], that is, telling for a given pose of the robot whether it intersects any obstacles. Typical approaches for complex robot or obstacle shapes use hierarchies of simpler shapes. GPUs have been used for collision detection of complex and deformable shapes[34]. Kavraki[45] explored the use of the Fast Fourier Transform(FFT) to compute configuraton-space obstacles. A single FFT can be computed in using parallel operations, and several FFT computations are done in parallel to compute the c-space obstacles. For robots that consist of a single rigid body, it is simple to construct a lookup table containing the admissibility of all possible poses within some discretization. More complex structures can also use one lookup table per component rigid body. Lozano-Perez and O'Donnell[55] describe a parallel method for computing the configuration space of a 6-DOF robotic arm, though they use a sequential search algorithm. Perez-Francisco et al.[58, 72] describe a parallel algorithm for collision detection based on a representation of obstacles as hierarchies of spheres. Their approach is suitable for complex interacting robots, but it is not the best choice for robots with simple rigid shapes and when obstacles are represented as an occupancy grid. Ferguson et al. use a distance transform to compose a hierarchical representation of a vehicle with circles[29], but

this still requires several table lookups to check a pose for collisions.

In Section 5.1 we revisit the comparison[45] between FFT and direct image convolution, using contemporary parallel computers.

3.4.7 Heuristics for Motion Planning in Unstructured Environments

For A*-based search, an admissible heuristic function is required to ensure an optimal path.

An optimal search proceeds to find the lowest-cost path from a given start node q_I to a given goal node q_G on a graph $G = (V, E)$ with edge costs $c : V \times V \to \mathbb{R}^+$. We define $c(u, v) = \infty$ for edges $(u, v) \notin E$, indicating that the edge cannot be part of a valid path.

To achieve a provably optimal result in A* and derived algorithms such as D*, the heuristic function $h(n)$ estimating the cost from node n to the goal node q_G must be *admissible*, meaning that $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost of the minimum path from the node n to the goal. A* examines nodes it encounters in order of increasing $f(n) = g(n) + h(n)$, where $g(n)$ is the proven minimum cost to reach n from q_I. This makes $f(n)$ an under-estimate of the total cost of the shortest path from q_I to q_G that passes through n.

For some problems, the graph is a discrete subset of a continuous metric space. For example, in robot path planning, the search graph is a discretization of $\mathbb{R}^2$. In these cases the heuristic can be a function derived directly from the metric space, rather than indirectly through the search graph. For example, the straight-line Euclidean distance $d(n, q_G)$ from n to the goal q_G is independent of the graph chosen to discretize the space for the search.

One can also use a direct relaxation of the search graph G as a heuristic. For example, one can construct a smaller heuristic graph $G_h = (V_h, E_h)$, where $p : V \to V_h$ for a chosen V_h with $|(V_h)| \leq |(V)|$, and

$$E_h = \{(p(u), p(v)) : (u, v) \in E\}.$$

The cost $c_h(u, v)$ must satisfy

$$c_h(u_h, v_h) \leq \min_{u,v \in V} \{c(u, v) : u_h = p(u), v_h = p(v)\}.$$

Figure 13 illustrates with a search graph. The full search graph consists of the sets of nodes $\{u_i\}$, $\{v_j\}$, and the edges between them. The projected heuristic graph contracts the $\{u_i\}$ and $\{v_j\}$ into the single nodes u_h, v_h with a single edge between them having a lower cost than any of the edge costs $c(u_i, v_j)$.

Pattern databases[21] are an effective way of composing heuristics for many search problems. For example, Urban Challenge entrants Junior[25] and Boss[29] combined two pattern database heuristics. The first heuristic function $h_1(\cdot)$ is formed as a database giving the length of the shortest path from all positions within a radius of the goal, respecting the kinematic constraints of the vehicle, and without regard for the presence of obstacles. This pattern database is independent of the obstacle grid. It can be precomputed and used unchanged for all planning scenarios. The second heuristic function

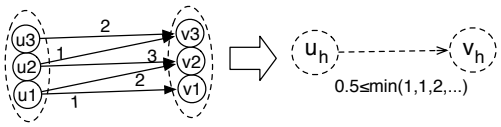


Figure 13: Projection of the search graph to the heuristic graph

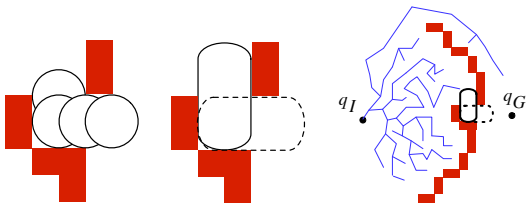


Figure 14: A gap (a) that the ball heuristic suggests may be passable, but (b) cannot be navigated by the real vehicle, and (c) the "bowl" of search space explored as a result

$h_2(\cdot)$ is formed as pattern database recomputed at each planning cycle. First, a distance transform is applied to the occupancy grid. Second, a single-source shortest-path is applied over the cells of the grid, where adjacent cells are connected if the distance to the nearest obstacle exceeds the radius of the largest ball that fits inside the vehicle. This heuristic is admissible since a ball that fits inside the vehicle and can move in any direction can follow any path the vehicle can. The final heuristic is combined as $h(n) = \max(h_1(n), h_2(n))$. The maximum of two admissible heuristics is also admissible. The heuristic $h_2(\cdot)$, or "ball" heuristic, can mislead the A* search by drawing its attention towards gaps that could be navigated by a ball containing the vehicle, but which the vehicle cannot navigate, as illustrated in Figure 14. In this case, the A* search must expand all nodes along paths with an estimated total cost less than the estimated total cost of the actual lowest-cost path. This includes the many nodes within the bowl shown in part (c) of the figure.

In Figure 13, if $c(u_h, v_h) \lll \min_i c(u_i, v_i)$, then the heuristic is misleading, and is potentially creating a bowl.

Increasing the value of $h(q)$ for some nodes q while retaining admissibility may reduce the number of nodes expanded by A* and can never increase it[67].

Junghanns and Schaeffer[43] address this problem using *relevance cuts* in the context of a combinatorial problem with multiple subgoals. In this case, the search algorithm will try all transpositions of moves that advance independent subgoals. Relevance cuts attempt to characterize which moves are independent from each other and enforce that only moves that are affected by the previous one should be considered. They demonstrated results using the Sokoban puzzle, but optimality is guaranteed only with IDA*.

Raphael[76] proposes Coarse-To-Fine Dynamic Programming (CFDP), a search algorithm that uses the observation illustrated in Figure 13 to search in success refinements of an approximated search space. However, CFDP relies on a clear prior

ordering of states in the space.

In Section 5.2 we present results for an improved heuristic that uses parallel computation to find and correct situations where $c(u_h, v_h) \lll \min_i c(u_i, v_i)$ in the "ball" heuristic, resulting in an improved heuristic.

3.5 Summary of Related Work

We have seen that effective motion planning solutions make use of a variety of algorithms and search space formulations. There are opportunities for additional research into parallel algorithms to compute data artifacts related to collision checking, heuristics used for A* and related search algorithms, and cost functions to shape behavior. The state lattice approach also offers an interesting opportunity to exploit parallel computation to improve plan quality and reduce planner latency. In the next section we describe how we propose to use GPUs to address these challenges.

4 Technical Approach

The aim of this thesis is to show how modern GPUs can be applied to enhance the performance of motion planning. We are interested in motion planning problems that require the robot to consider complex kinematic and dynamic constraints in plotting possible courses of action, and to consider its own velocity as well as the velocities of moving obstacles in their environment, when calculating how to avoid collisions. These problems require significant computation to resolve. We will develop novel parallelizations of existing algorithms and show how they can significantly improve the real-time performance of a range of solutions in motion planning. We will also investigate new approaches to problems in motion planning that are enabled by the differing performance tradeoffs made by these processors as compared to traditional CPUs. Approaches will include using GPUs to plan faster in existing spaces; plan in higher dimensional spaces; and use more computationally expensive cost functions to improve the expressiveness and quality of plans.

The architectural features of the GPU that we will exploit(or finesse) are the data-parallel conditional vector operations, high memory bandwidth and high latency, and high throughput for data-parallel arithmentic operations. Conditional vector operations are the core form of parallelism in GPUs from both Nvidia and AMD, the two market leaders, as well as hybrid sequential/parallel processors such as IBM's Cell[73] and Intel's proposed Larrabee[82],

Naturally, an initial approach to parallel motion planning should be to examine existing algorithsm for independent operations that can be executed in parallel. For example, loops where the data elements processed in each iteration are independent of others can be executed in parallel. Algorithms involving additions and multiplications of matrices are good candidates. In later sections we shall see an approach to configuration space analysis that is also easy to perform in parallel.

A second approach would be to reformulate an algorithm to expose new opportunities for parallelism, or to look for alternative existing algorithms that expose more opportunities for parallelism. For example, the single-source shortest path algorithm is

commonly solved using Dijkstra's algorithm[24]. Dijkstra's algorithm is sequential in nature. It examines one vertex in its input graph at a time and processes all of its edges before moving on to the next vertex. A direct parallelization of Dijkstra's algorithm would at best process all edges of each vertex in parallel. While no general-purpose parallel algorithm exists for SSSP with a guaranteed parallel speedup, more successful parallelizations approach the problem differently, for example by managing a wave-front of vertices whose edges can all be processed in parallel, or by assuming certain properties of the graph.

Processor architectures such as GPUs that rely on vector operations may have more trouble exploiting opportunities for parallelism if execution paths for independent data items tend to be highly divergent. The balance between the desired latency for a computation, the volume of data to process, and the computation resources available may affect the choice of algorithm. For example, an algorithm that does n^2 work with n depth may be desirable if few processors are available compared to the size of the data set, and overall latency is not a concern. If there are many processors available and latency is a concern, then an algorithm that does $n^2 \log n$ work overall but with $\log n$ depth may be preferable.

When it is not possible to effectively parallelize existing algorithms, or if additional resources are still available after all possible parallelization is achieved, then one can consider improving the qualitative performance of the problem. That is, increase the size of problem to be solved without changing the algorithm, or perform additional computations to enhance the result. In motion planning, for example, we may add dimensions to the state space, use a finer discretization along some dimensions, or increase the size of the action set in a dynamic programming formulation. All of these measures may increase the quality of the solution without increasing the planning latency, if parallel computational resources are available. For motion planning again, an example of computing additional quantities in order to enhance the results is to increase the complexity of the cost function. For example, Dolgov et al.[25] computed a complex potential function using the distance transform and generalized voronoi diagram to guide a robotic vehicle towards open spaces, without discouraging it from navigating through narrow passages. This function could be precomputed in a table, with each element computed in parallel.

We will demonstrate the efficacy of our work by implementing a high-performance real-time motion planner on a real vehicle using a GPU-powered computing platform.

5 Work Completed To Date

We have done preliminary research in a number of areas reviewed in the section on related work. We have found that GPUs have the ability to significantly accelerate collision checking tasks. The ability of the GPU to rapidly generate a lookup table for collision tests can be leveraged into an improved heuristic for sequential search-based motion planning using a local kinematic analysis to repair misleading values in the heuristic table. We have developed a novel parallel algorithm for the single-source shortest path problem that can be used to accelerate computation of the heuristic with a GPU. Finally, we have derived a novel construction of the state lattice that can be

Processor	Time (ms)	Speedup
Intel 2.33 GHz Core 2 Quad *(one thread, no SSE operations)*	56	1
Intel 2.33 GHz Core 2 Quad *(four threads, SSE operations)*	12	4.7
Nvidia 8600 GTS, 32 cores	11	5
Nvidia GTX 260, 216 cores	2.2	25

Table 1: Execution times for single FFT on grid of size 1024×1024.

applied to on-road driving tasks.

We review each of these in turn in the following sections.

5.1 Parallel Configuration Space Analysis

As noted in Section 3.4.6, collision detection is one of the most costly computations in motion planning.

A Fast Fourier Transform (FFT) can be used to compute the convolution of a robot body shape with the obstacle grid. The well-known Convolution Theorem states that

Theorem 5.1 (Convolution Theorem) *If functions f and g defined on $\mathbb{R}$ are integrable then $\widehat{f \otimes g}(x) = \widehat{f}(x) \cdot \widehat{g}(x)$, where $\widehat{h}$ denotes the Fourier Transform of function h.*

We compared direct convolution in image space with use of an FFT to create a table for a single robot in a static obstacle grid.

Since the overlap of a robot with an obstacle is a binary condition, 32 orientations can be tested in parallel on a 32-bit computer with no extra effort by using one bit for each orientation in the rasterization of the vehicle, and using an occupancy grid with bits set to all ones or all zeros. Bitwise "and" and "or" operations can then be used to test for overlap between the vehicle and the obstacles. When a GPU can be used it is profitable to perform the full analysis of configuration free space over the occupancy grid. The algorithm is simple to implement, unlike approaches based on analytical geometry.

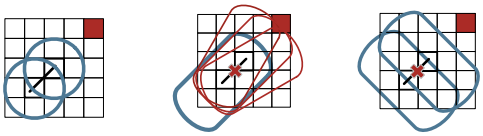
The GPU is able to rapidly analyze the configuration free space and compute a simple lookup table for collision checking purposes. Use of Intel's SSE (SIMD) instructions on a multi-core processor can also accelerate these calculations, though to a lesser degree. Use of the GPU reduces to a minimum the cost of collision checking for a single-body robot, and simplifies the implementation significantly compared to prior approaches based on, for example, using a distance transform to represent the robot as an agglomeration of circles. The lookup table can be used as a basis for a deeper analysis of viable robot motions that can significantly improve the performance of sequential search, as we show in the next section.

5.2 Local Kinematic Analysis

In kinodynamic planning, the Inevitable Collision State(ICS)[32] is a heuristic used to anticipate which states must lead to a collision. A search algorithm can use this heuris-

Method	Processor	Time (ms)	Speedup
FFT	Intel 2.33 GHz Core 2 Quad *(one thread, no SSE operations)*	1850	1
	Intel 2.33 GHz Core 2 Quad *(four threads, SSE operations)*	400	4.6
	Nvidia 8600 GTS, 32 cores	365	5
	Nvidia GTX 260, 216 cores	75	25
Direct	Intel 2.33 GHz Core 2 Quad *(four threads, SSE operations)*	192	1
	Nvidia 8600 GTS, 32 cores	113	1.7
	Nvidia GTX 260, 216 cores	19	10

Table 2: Execution times to create collision table for 32 body orientations on grid size 1024×1024, with kernel size 33×33.



(a) (b) (c)

Figure 15: Motions which a vehicle with typical kinematic constraints is not capable of making, though (a) the ball heuristic would indicate that it is, while in (b), the vehicle can perform the motion but would be blocked by an obstacle. In (c) the vehicle cannot perform the motion even though the obstacle does not interfere.

tic to avoid performing an exhaustive search for a valid path from such a state[12]. The ICS does not help in motion planning problems that require a non-circular robot to navigate tight spaces at low speeds, that is, where dynamics are not a consideration.

In Section 3.4.7 we described how large discrepancies between the estimated heuristic $h(n)$ used in A* search and the true lowest cost $h^*(n)$ to reach the goal from a given node n can induce bowls in the search space that can cause A* to spend excessive time exploring. We observed that these bowls are formed when

$$c(u_h, v_h) \lll \min_i c(u_i, v_i). \tag{1}$$

For vehicle search using the ball heuristic, Equation 1 obtains in and near narrow gaps between obstacles. If the detailed configuration-space analysis enabled by parallel computation described in Section 5.1 is available, the heuristic can be improved. Each edge (u_h, v_h) in the ball heuristic represents the projection of a set of edges $\{(u_i, v_i)\}_{i=0\ldots n}$ in the full state space. We can analyze all of those edges, compute $\hat{c}(u_h, v_h) = \min_i c(u_i, v_i)$, and replace the value given by the ball heuristic. This computation can be done in parallel for all edges in the heuristic graph.

Applications where this heuristic enhancement for low-speed navigation would be helpful include cars driving in parking lots[29], or rovers in rocky terrain[84].

There is the question of whether it is more efficient simply to compute the shortest path as a single-source shortest path problem directly in the full (x, y, θ) state space using the convolution as the source graph. Preliminary experiments suggest that this is not so. When the direction convolution method is used, one can restrict the full configuration space analysis to areas near obstacles. If the terrain contains large open areas and narrow passages, the savings could be significant. We intend to do additional experiments in this domain to determine the tradeoffs.

5.3 Parallel Single-Source Shortest Path

When A* or a derived search algorithm is used to find a path, a heuristic function $h(s)$ is required to give an estimate of the cost-to-go from the state s to the goal. To guarantee that the path is optimal, $h(s)$ must be *admissible*, meaning that $h(s) \leq h^*(s)$, where $h^*(s)$ is the true optimal cost-to-go. A rich source of heuristics for many problems is to compute a table giving the cost-to-go from all states for a relaxation of the problem. For motion planning, a pattern database table[21] can be used to create an effective heuristic[54, 25] for a problem instance by approximating the vehicle as a two-dimensional point and computing the cost-to-go from each grid cell to the goal. Since a point moving with no constraints can travel any path that the vehicle can travel, this heuristic is guaranteed to be admissible.

The cost-to-go heuristic table can be computed given the cost graph corresponding to the grid by using a single-source-shortest path (SSSP) algorithm. The SSSP problem is to find the shortest path from all nodes in a graph G to a single sink node $s \in V(G)$. SSSP is an important graph problem that is recognized as being difficult to parallelize. Meyer described a parallel SSSP algorithm called Δ-stepping[60]. We have developed the GPU-Δ-stepping and GPU-Δ-sliding algorithms, which are more appropriate to the GPU architecture. The original Δ-stepping algorithm is reproduced in Figure 16 for convenience. Various steps in the algorithm can be parallelized. Line 10, 13–14, 15, 16-17 can be done in parallel. Calls to relax() require some form of inter-processor communication or synchronization in order to avoid race conditions in simultaneous updating of values of $tent(v)$, which is the lowest-known cost from the source node s to v. When the algorithm is finished, $tent(v)$ is the cost of the shortest path from v to s.

The design of our GPU-Δ-stepping algorithm differs from the stock Δ-stepping algorithm. We are more explicit about what is parallel, and how operations are synchronized. Certain data structures related to recording which bucket $B[]$ each node v is in are not explicitly addressed by Δ-stepping, and these turn out to be difficult to parallelize efficiently. Therefore we are explicit in our algorithm about the form of the data structures called for by the GPU architecture. In GPU-Δ-stepping, we do not explicitly store the bucket each node v is in. All nodes that are reached but not settled are stored together. Line 5 tests for each $v \in V_{in}$ whether v is in the bucket currently being relaxed. This approach takes advantage of the fact that a large number of threads can be simultaneously executed by the GPU, avoiding any time wasted recording which bucket a node is in, as is necessary at Line 9–10 in stock Δ-stepping in Figure 16. It is faster to load and re-store nodes that are not expanded than it is to keep track of which nodes should be expanded so that only they are loaded.

Δ-stepping algorithm:

Input: $G(V, E)$, source vertex s, length function $\ell : E \to \mathbb{R}$
Output: $\delta(v), v \in V$, the weight of the shortest path from s to v

1 **foreach** $v \in V$ **do** *Initialize data structures*
2 $heavy(v) \leftarrow \{(v, w) \in E : \ell(v, w) > \Delta\}$
3 $light(v) \leftarrow \{(v, w) \in E : \ell(v, w) \leq \Delta\}$
4 $tent(v) \leftarrow \infty$
5 relax$(s, 0)$ *Source node at distance 0*
6 $i \leftarrow 0$
7 **while** B is not empty **do** *Still have queued nodes*
8 $S \leftarrow \emptyset$
9 **while** $B[i] \neq \emptyset$ **do** *Relax nodes in current bucket*
10 $Req \leftarrow \{(w, tent(v) + \ell(v, w)) : v \in B[i] \wedge (v, w) \in light(v)\}$ *Nodes to relax later*
11 $S \leftarrow S \cup B[i]$ *Track all nodes that were in $B[i]$*
12 $B[i] \leftarrow \emptyset$
13 **foreach** $(v, x) \in Req$ **do**
14 relax(v, x) *May re-insert nodes into $B[i]$*
15 $Req \leftarrow \{(w, tent(v) + \ell(v, w)) : v \in S \wedge (v, w) \in heavy(v)\}$ *Relax heavy edges*
16 **foreach** $(v, x) \in Req$ **do**
17 relax(v, x)
18 $i \leftarrow i + 1$ *Current bucket empty, look at next one*
19 **foreach** $v \in V$ **do** *Finalize tentative weights*
20 $\delta(v) \leftarrow tent(v)$

relax() routine for Δ-stepping algorithm:

Input: v, weight request x

1 **if** $x < tent(v)$ **then** *New weight for v is lower, move into new bucket*
2 $B[\lfloor tent/\Delta \rfloor] \leftarrow B[\lfloor tent/\Delta \rfloor] \setminus \{v\}$
3 $B[\lfloor x/\Delta \rfloor] \leftarrow B[\lfloor x/\Delta \rfloor] \cup \{v\}$
4 $tent(v) \leftarrow x$

Figure 16: Meyer's Δ-stepping algorithm[60]

GPU-Δ-stepping:

Top-level SSSP algorithm

Input: $G(V, E)$, start node s, length function $\ell : E \to \mathbb{R}$,

Output: $\delta(v)$, shortest path to s for all nodes

1 $V_{in} \leftarrow \{s\}$; $i \leftarrow 0$
2 **while** V_{in} not empty **do**
3 $(V_{out}, i) \leftarrow$GPU-Δ-stepping-step(V_{in}, i)
4 Swap V_{in} and V_{out}

GPU-Δ-stepping-step:

Single parallel update step for SSSP algorithm.

Input: input list V_{in} of reached nodes, current bucket number i

Output: New list V_{out} of reached nodes, updated $\delta(v), v \in V$, next bucket number i'

1 $i' \leftarrow i + 1$
2 **for** $\{v | v \in V_{in}\}$ **in parallel do**
3 $vcost \leftarrow tent(v)$
4 $vbucket \leftarrow \lfloor vcost/\Delta \rfloor$
5 **if** $vbucket = i$ **then**
6 **for** $\{(v, w) | (v, w) \in E(G)\}$ **in parallel do**
7 $newwcost \leftarrow vcost + \ell(v, w)$
8 $oldwcost \leftarrow \text{atomicMin}(tent(v), newwcost)$ atomicMin(a, b) *sets* a *atomically to* min(a,b) *and returns* a
9 **if** $newwcost < oldwcost$ **then** *Cost of* w *did change*
10 $oldwbucket \leftarrow \lfloor oldwcost/\Delta \rfloor$
11 **if** $oldwcost = \infty$ or $oldwbucket = i$ **then** w *is newly reached or already in bucket* i
12 $V_{out} \leftarrow V_{out} +\!\!+ \{w\}$ *This could duplicate* w *in* V_{out}
13 **if** $newwbucket = i$ **then**
14 $i' \leftarrow i$ *assume concurrent write*
5 **else**
15 $V_{out} \leftarrow V_{out} +\!\!+ \{v\}$ v *is not in bucket* i *so defer*

Figure 17: Our parallel GPU-Δ-stepping algorithm

An undesirable side effect of not recording which bucket each node v is in is noted at Line 12 of Figure 17. Two threads t_1 and t_2 could simultaneously relax from two nodes v, v' to the same w with a $newwcost$ respectively of a and b, where $b < a < tent(w)$, and w is already in bucket i, (i.e. $i = \lfloor tent(w)/\Delta \rfloor$). If t_1 executes atomicMin() first, and t_2 second, then the test at lines 9 and 11 will succeed for both threads, and w will be appended to V_{out} twice. We have not yet investigated exactly how often this happens, but we expect it to be uncommon.

Note that in GPU-Δ-stepping we have dispensed with the separating of edges into $heavy$ and $light$ sets, opting to update all edges (Line 6) at every iteration. This is not work-efficient, but constraints on the GPU made it impossible to divide up edge relaxations into separate $heavy$ and $light$ phases while achieving good performance.

We have an implementation of GPU-Δ-stepping that is tuned for the GPU, of course, and for an adjacency list representation tuned for the graph of 8-connected grid cells. We store edge costs in a matrix $g[n][8]$ where $n = |V(G)|$, and each entry $g[v][x]$ is the cost $\ell(v, neighb(v, x))$ of the edge from the v to the xth neighbor of v in the grid. It is convenient that 32, the number of threads scheduled together in a warp, is divisible by 8, the number of neighbors of each vertex in the graph. Our implementation assigns each warp of 32 threads to process in parallel the neighbor of groups of 4 vertices in V_{in}, at line 2 and line 6 of the graph.

A limitation of GPU-Δ-stepping is that an entire iteration of GPU-Δ-stepping-step may be executed to relax a bucket $B[i]$ even when there is only one node left in that bucket, and many nodes in the next bucket $B[i+1]$. Since the bucket data structures have been dispensed with, it is possible to adapt the bucket boundaries from $[i\Delta, (i+1)\Delta]$. We devised a new GPU-Δ-*sliding* algorithm with a sliding bucket.

Referring to Figure 18, at the end of each iteration GPU-Δ-sliding-step, each block b records to a global memory structure the lowest tentative cost $low(b) = \min_{v \in V_{out}} tent(v)$ of any node v written to V_{out} by any thread in the block. At the start of the next iteration, each block loads the $low(b)$ values of *all* blocks from the previous iteration, computes the minimum $m = \min_b low(b)$, and sets the "bucket" for that iteration to $[m, m + \Delta]$. Note that having each block write out its own $low(b)$, then load all values and compute the same minimum value independently is faster than having each block use an atomic memory operation to maintain a single minimum value in global memory. Figure 18 shows the algorithm. The vertices that are settled at each iteration, that is, those vertices whose optimal cost-to-goal value is known, are identified as those vertices v in V_{in} whose cost-to-goal is less than $\min_b low(b)$ at the next iteration.

Tests were run on a representative occupancy grid shown in Figure 19.

Figure 20 shows the performance of GPU-Δ-sliding compared to GPU-Δ-stepping and the best CPU performance obtained. GPU-Δ-sliding is faster than GPU-Δ-stepping for the majority of values of Δ, and tends to achieve its peak performance at lower values of Δ. Table 3 summarizes the peak performance obtained on each platform by each algorithm and tuning parameter value. The 8600 GTS is able nearly to meet the performance of the sequential algorithm run on the Core 2 Quad, and the GTX 260 achieves a 30% faster time, at 14 ms compared to 22 ms. Further developments in the SSSP algorithm for the GPU should improve that number further.

We propose additional enhancements to GPU-Δ-stepping. The current version performs poorly when the graph has a large variance in edge costs. When Δ is large

GPU-Δ-sliding:

Top-level SSSP algorithm

Input: $G(V, E)$, start node s, length function $\ell : E \to \mathbb{R}$,

Output: $\delta(v)$, shortest path to s for all nodes

1 $V_{in} \leftarrow \{s\}$; $low[1 \ldots nblock] \leftarrow 0$

2 **while** V_{in} not empty **do**

3 $(V_{out}, low) \leftarrow$GPU-$\Delta$-sliding-step($V_{in}$,$low$)

4 Swap V_{in} and V_{out}

GPU-Δ-sliding-step:

Single parallel update step for SSSP algorithm.

Input: input list V_{in} of reached nodes, current bucket number i

Output: New list V_{out} of reached nodes, updated $\delta(v), v \in V$, new array low

1 $minlow \leftarrow \min_{b \in [1 \ldots nblocks]} low(\text{b})$; $lowtmp(\text{thread}) \leftarrow \infty$

2 **for** $\{v | v \in V_{in}\}$ **in parallel do**

3 $vcost \leftarrow tent(v)$

4 **if** $vcost < minlow + \Delta$

5 **for** $\{(v, w) | (v, w) \in E(G)\}$ **in parallel do**

6 $newwcost \leftarrow vcost + \ell(v, w)$

7 $oldwcost \leftarrow \text{atomicMin}(tent(v), newwcost)$ atomicMin(a, b) *sets* a *atomically to* min(a,b) *and returns* a.

8 **if** $newwcost < oldwcost$ **then** *Cost of* w *did change*

9 **if** $oldwcost = \infty$ **then** w *is newly reached or already in bucket* i

10 $V_{out} \leftarrow V_{out} +\!\!+ \{w\}$ *This could duplicate* w *in* V_{out}

11 $lowtmp(\text{thread}) \leftarrow \min(lowtmp(\text{thread}), oldwcost)$

else

12 $V_{out} \leftarrow V_{out} +\!\!+ \{v\}$ v *is not in bucket* i *so defer*

13 $low(\text{thread.block}) \leftarrow \min_{\text{thread} \in block}(lowtmp(\text{thread}))$ Record the lowest-cost node seen by any thread in current block

Figure 18: The parallel GPU-Δ-sliding algorithm, with sliding bucket boundaries

Platform	Time	Notes
P4 3.2GHz	30 ms	$O(n)$ Dijkstra's
Core 2 Quad 2.33 GHz	22 ms	$O(n)$ Dijkstra's
8600 GTS	23 ms	Δ-sliding, $\Delta = 8$
GTX 260	14 ms	Δ-sliding, $\Delta = 16$
8600 GTS	26 ms	Δ-stepping, $\Delta = 64$
GTX 260	16 ms	Δ-stepping, $\Delta = 256$

Table 3: Comparison of time taken by CPU and by the fastest implementation of GPU-Δ-stepping and GPU-Δ-sliding on each GPU, with best values of tuning parameters.

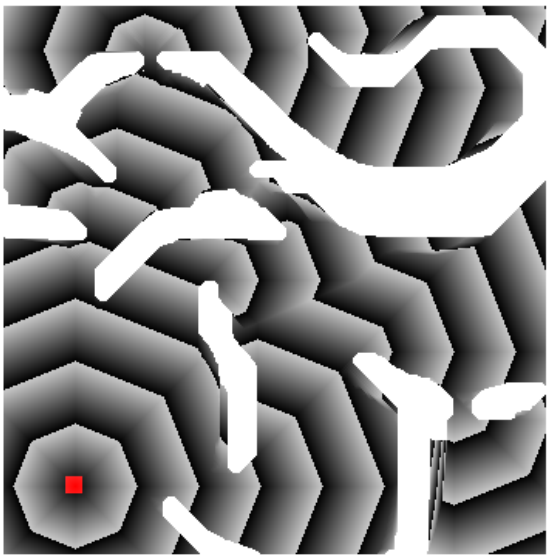


Figure 19: 384×384 occupancy grid test case for performance comparison of SSSP algorithms on GPU and CPU. The red dot in the lower-left represents the chosen source, white regions indicate obstacles, and grey ramps indicate the computed distance from the source.

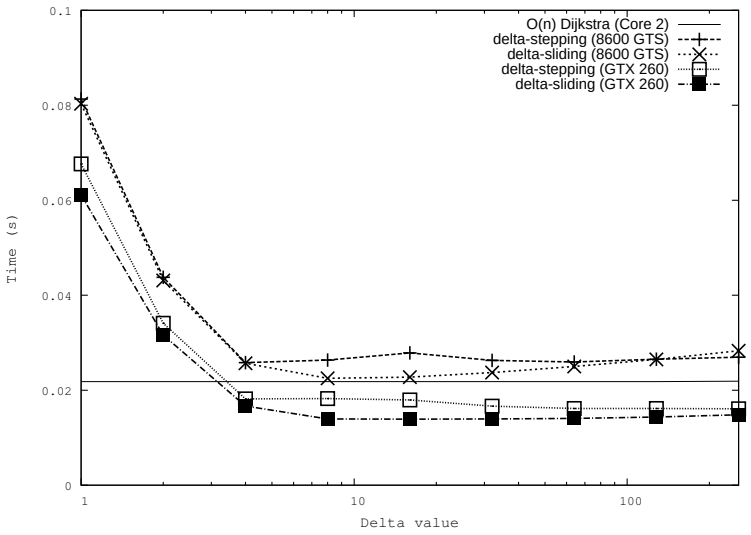


Figure 20: Performance comparison of Δ-sliding and -stepping on 2 GPUs vs $O(n)$ Dijkstra on 1 CPUs for various values of Δ. Each data point is the average of 100 runs to compute the SSSP of Figure 19. Error bars are not drawn since variation was typically less than 1 ms.

compared to the costs of the edges from nodes in the wavefront, many nodes may be added to the wavefront before their predecessor costs are settled. This results in a large wavefront with many nodes that are relaxed repeatedly before finally settling. When Δ is small compared to the costs of the edges in the wavefront, many relaxations may be performed before the heavy edges are finally settled. In order to solve both of these problems, the value of Δ must adapt. One possible solution would be to relax the k lowest-cost edges out of the tentative nodes in the wavefront. k would be tuned according to the number of multiprocessors available. The corresponding value of Δ could be computed approximately with a parallel median-finding algorithm. We will develop the algorithm further to mitigate this problem.

The current implementation assumes that fast atomic operations on global memory are available, but not on shared memory. Newer GPUs have atomic operations on shared memory, and future enhancements to the algorithm will take advantage of this to create more efficient maintenance of the wavefront data structures.

5.4 Kinodynamic Lattice Planner

Driving at high speeds on roads and in traffic presents a different set of challenges for the motion planning problem compared to driving at low speeds off-road, or in parking lots. In the latter cases, vehicle dynamics may be neglected, and the sudden appearance of an obstacle can be dealt with by stopping the vehicle and taking time to compose a new plan. In addition, time and velocity typically need not be considered in the plan, insofar as it is relevant to the vehicle's interaction with other vehicles or obstacles. By contrast, driving on-road requires the planner to consider the dynamics of the vehicle when evaluating candidate actions, and also to consider the velocities of other vehicles on the road, and their predicted future locations. A sudden change in conditions may require a new plan to be generated very quickly, and coming quickly and safely to a stop in order to compute a plan may itself require a complex plan.

When a planner uses a model significantly dissimilar from the real robot, the resulting plans often cannot actually be followed closely by the robot. This can lead the robot into a situation it cannot plan a way out of. In Section 3.4.2 we described the *state lattice*, a recent development in search-based motion planning methods for vehicles. The purpose of the state lattice approach is to efficiently find kinematically feasible plans using a discrete graph search. To date, state lattices have been used for planning in unstructured environments. In this section we will describe how we applied it to an on-road driving scenario. We have run experiments to set a lower bound on the fidelity it should be possible to achieve between a plan and the actual path followed by the robotic car. We have also established that the planner is amenable to acceleration by a GPU and that the use of the GPU allows the planner to compute results in real-time for a relatively simple but realistic case.

5.4.1 Approach

A state lattice is a graph formed by sampling states from a continuum. Each vertex in the graph represents one such sampled state. A directed edge (a, b) in the graph corresponds to a trajectory from state a to state b consistent with the robot kinematics

or dynamics. Since such trajectories are expensive to compute, states are sampled in a regular pattern as illustrated in Figure 10 of Section 3.4.2, so that the same trajectories can be used out of each node.

More precisely, if the robot state is represented as a vector $\mathbf{x}$, and the dynamics by $\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u}(\mathbf{x}, t))$, then an edge connecting two states $\mathbf{x_1}, \mathbf{x_2}$ is the trajectory described by

$$\mathbf{x}(\mathbf{x}_1, \hat{t}) = \mathbf{x}_1 + \int_0^{\hat{t}} f(\mathbf{x}(t), \mathbf{u}(\mathbf{x}(t), t))dt,$$

and $\mathbf{x}(t_f) = \mathbf{x_2}$ for some t_f. The main challenge in constructing the lattice is in finding a function $\mathbf{u}(\mathbf{x}, t)$ that satisfy the boundary constraint $\mathbf{x}(\mathbf{x}_1, t_f) = \mathbf{x}_2$ for each $\mathbf{x}_1, \mathbf{x}_2$. By evaluating a cost functional over the trajectory, a cost can be assigned to each edge in the search graph. An example of such a cost functional could be the length of the path, with ∞ on edges where the vehicle would strike an obstacle along the path.

The trajectory can be generated by any available means to solve a boundary-value problem from $\mathbf{x_1}$ to $\mathbf{x_2}$ using the differential equation describing the vehicle dynamics $\dot{\mathbf{x}} = f(\mathbf{x}, u)$. With Boss[41] for example, a shooting method was used to find parameters $\mathbf{p}$ in a parameterized control scheme $\mathbf{u}(\mathbf{p}, \mathbf{x}, t)$.

Once the graph is constructed, a search algorithm can be applied to the graph. We shall discuss the issue of selecting the search algorithm later in this section.

Next we describe the lattice construction procedure in more detail.

5.4.2 Lattice Construction

For lattice generation and planning, we approximate the vehicle with a simple kinematic bicycle model, the same as that used for Boss[29] in the Urban Challenge.

$$\begin{aligned}
\mathbf{x} &= [x \ y \ \theta \ \kappa_w v] && (2)\\
\dot{x} &= v\cos(\theta) && (3)\\
\dot{y} &= v\sin(\theta) && (4)\\
\dot{\theta} &= v\tan(\kappa_w)/W && (5)\\
\dot{v} &= \textit{control input} && (6)\\
\dot{\kappa}_w &= \textit{control input}, && (7)
\end{aligned}$$

where x, y are the coordinates of the center of the rear axle, θ is the vehicle heading, W is the wheelbase of the vehicle, v is velocity, and κ_w is the front-wheel angle. The local curvature of the path followed by the vehicle is $\kappa = \tan(\kappa_w)/W$. In our implementation the model is independent of W, so that $\mathbf{x} = [x \ y \ \theta \ \kappa]$ with $\dot{\theta} = v\tan(\kappa)$ with appropriate modifications made to the steering control input during path execution. We parameterized the steering function $u(\mathbf{x}, t)$ used a polynomial spiral as in[46]:

$$\kappa(s) = \kappa(0) + p_1 s + p_2 s^2 + p_3 s^3,$$

where $\kappa(s)$ is a function s, the length of the path driven.

Previous work in state lattices[75, 29] for unstructured environments have used regular distribution of states in (x, y) space, allowing identical action trajectories to be

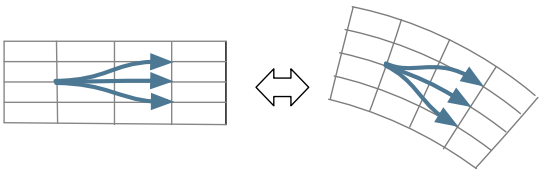


Figure 21: Actions in (s, l) space are warped to match (x, y, θ, κ) values at the warped locations of endpoints in (x, y) space.

used from all states. Roads may have high variations in curvature over short distances, so that the same action set cannot be used at all states without radically increasing the number of actions in the set. A better approach may be to warp the lattice to fit the roadway, and optimize each action to match. We sampled vertices for the lattice in the space $(s, \ell, \theta, \kappa_s, v)$, where s and ℓ are the station and latitude respectively of the center of the vehicle's rear axle, θ is the heading of the sprung mass of the vehicle (i.e. the rigid body of the car supported by the suspension system) with respect to the local heading of the road, κ the instantaneous curvature of the vehicle's motion with respect to the local curvature of the road, and v the longitudinal velocity, i.e. the component of the velocity collinear with the vehicle heading. Referring to the center line of the road, the station of a point is its distance along the road from an origin, and latitude its perpendicular distance from the center. Since our regular lattice in (station,latitude) space is warped onto a curved road in (x,y), each trajectory must be individually tuned to join connecting states. An action $a_i = (s_i, l_i)$ joins state pairs of the form $[(s, l) \rightarrow (s + s_i, l + l_i)]$.

For our experiments we selected states of the form $(s \text{ m}, 0.1\ell \text{ m}, 0, 0, v(s))$ for $s, \ell \in \mathbb{Z}$ and $v(s)$ the velocity given as a function of station. Edges connecting states were defined by pairs (a, b) with $a, b \in Z$, connecting $(s, 0.1\ell, 0, 0, v(s))$ to $(s + a, 0.1(\ell + b), 0, 0, v(s + a))$. These states have the vehicle oriented locally parallel to the road, with the front wheels set to follow the local road curvature. The actions connecting the states can be seen as swerves. The two boundary points of each action are mapped into global (x, y, θ, κ) space using the nominal "zero" θ for states in the lattice as the local heading of the road at (x, y), and the nominal "zero" curvature κ as the local curvature of the road. Figure 23 illustrates. Each action must then be tuned to correctly join the endpoints warped into the (x, y) space, as in Figure 21.

Given an initial state $\mathbf{x_I}$ and desired terminal states $\mathbf{x_T}$, we need to find path parameters $\mathbf{p} = [p_1 \ p_2 \ p_3 \ s_T]$ where $p_{1\ldots3}$ are the cubic polynomial coefficients and s_T is the length of the path. A simple shooting method suffices. Again, denoting the dynamics of the system

$$\dot{\mathbf{x}}(t) = f(\mathbf{x}(t), \mathbf{u}(\mathbf{p}, \mathbf{x}, t)),$$

then the final state $\mathbf{x_T}$ given parameters $\mathbf{p}$ is

$$\mathbf{x_T}(\mathbf{p}) = \mathbf{x_I} + \int_{t_I}^{t_f} f(\mathbf{x}(t), \mathbf{u}(\mathbf{p}, \mathbf{x}, t))dt.$$

$\text{FindActionParameters}(\mathbf{x_I}, \mathbf{x_T}) \rightarrow \mathbf{p}$
    $\mathbf{p} \leftarrow [\ 0\ 0\ 0\ ||x_F - x_I, y_F - y_I||\ ]$
    repeat
        $\Delta\mathbf{x_T} = \mathbf{x_T} - \mathbf{x_T}(\mathbf{p})$
        $\Delta\mathbf{p} = -{\mathbf{J_P}}^{-1}(\mathbf{x_T}(\mathbf{p}))\Delta\mathbf{x_T}$
        $\mathbf{p} \leftarrow \mathbf{p} + \Delta\mathbf{p}$
    until convergence or iteration limit reached

Figure 22: The shooting-method algorithm used to solve for action parameters

Given the error

$$\Delta\mathbf{x_T} = \mathbf{x_T} - \mathbf{x_T}(\mathbf{p})$$

between the desired terminal state $\mathbf{x_T}$ and the actual terminal state $\mathbf{x_T}(\mathbf{p})$, we can compute an update to the parameter vector using the Jacobian of the terminal state with respect to the parameters

$$\Delta\mathbf{p} = -{\mathbf{J_P}}^{-1}(\mathbf{x_T}(\mathbf{p}))\Delta\mathbf{x_T},$$

which can then be applied to the parameters

$$\mathbf{p} \leftarrow \mathbf{p} + \Delta\mathbf{p}.$$

This process is repeated until $\Delta\mathbf{x_T}$ is acceptably small, or an iteration limit is reached. The Jacobian $\mathbf{J_P}(\mathbf{x_T}(\mathbf{p}))$ is of the form

$$\mathbf{J_p}(\mathbf{x_T}(\mathbf{p}))_{ij} = \left[\frac{\delta\mathbf{x_T}(\mathbf{p})_j}{\delta\mathbf{p}_i}\right],$$

and the entries are computed numerically via central differencing with a step parameter h, that is,

$$\frac{\delta\mathbf{x_T}(\mathbf{p})_j}{\delta\mathbf{p}_i} \approx \frac{\mathbf{x_T}(\mathbf{p} + \mathbf{h_i})_j - \mathbf{x_T}(\mathbf{p} - \mathbf{h_i})_j}{2h}$$

where $\mathbf{h_i}$ is the indicator vector with h in the ith position and 0 in all others.

$$\mathbf{h_i} = [0\ \ \cdots 0\ \ h\ \ 0 \cdots 0]$$

Typically an initial guess for $\mathbf{p}$ close to the desired answer is necessary. A large table of initial guesses was used for Boss[29], but the on-road driving trajectories used in this experiment have simpler shapes, so that a simple guess equivalent to maintaining the current steering direction is sufficient:

$$\mathbf{p} \leftarrow [\ 0\ 0\ 0\ ||(x_F, y_F) - (x_I, y_I)||\ ].$$

The algorithm is summarized as FindActionParameters() in Figure 22.

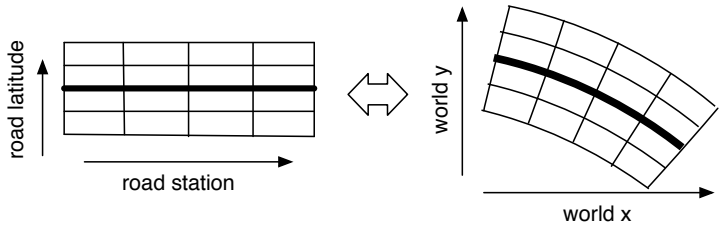


Figure 23: Mapping from the (station,latitude) lattice points defined on the road into global Euclidean (x, y) space.

5.4.3 Searching in the Lattice

A best-first search algorithm such as A* can be used to search a state lattice or indeed any graph, but for on-road driving, real-time constraints must be respected. In that case, we must seek a guarantee that the time taken by the algorithm in the worst case is less than the real-time deadline. For A*, in the worst case all nodes in the state space may be examined before a solution is found. It follows then that the state space discretization should be sized to the length of time taken in that worst case. In addition, if all nodes must be examined, then a more efficient algorithm than A* should be chosen. A straightforward and efficient dynamic-programming algorithm for a table can be used if the states in the graph have a clear ordering. For driving in a single direction down a road, states can be ordered by their longitudinal coordinate, and the values of states at the same longitude can be computed independently. An additional difficulty with A* is the effort required to compose informative and admissible heuristics. We are aware of very little work in finding such heuristics for on-road driving in traffic. Further, we anticipate that on-road driving will require extensive experimentation with cost functions in order to achieve a refined driving behavior. Each new cost function would require an accompanying A* heuristic. The need to compose a new heuristic function with each cost function tried would significantly increase the amount of effort required and reduce the amount of experimentation that could be done. All of these considerations suggest that an A* or similar search algorithm would not be a good choice for experiments with on-road driving under real-time constraints. We propose a dynamic programming algorithm using a suitably organized table.

Figure 24 shows StateLatticeDP(), our proposed algorithm. Variables s and l are the station and latitude respectively. Figure 23 illustrates the mapping between world coordinates (x, y) and road coordinates (s, l). Figure 25 illustrates an update step for the algorithm. All actions a incoming to each state $\mathbf{x}$ are evaluated. The cost of the action is added to the cost of reaching the action's predecessor state $a^{-1}(\mathbf{x})$, and the cost for $\mathbf{x}$ is taken as the minimum over all of those. The function $\mathbf{x} : \mathbb{Z}^2 \rightarrow \mathbb{R}^2$ maps from station and latitude into continuum coordinates, and $\mathrm{sl}(\cdot)$ is its inverse. The function $\mathrm{cost}(\mathbf{x}_1, \mathbf{x}_2)$ computes the cost of the edge. The output of the algorithm is the $\mathrm{costs}[]$ array which gives the cost to reach each cell from the start position. As Figure 25 shows, the regular format of the state lattice is well suited to a rapid dynamic

```
StateLatticeDP(x_V) → costs
    ∀s, l costs[s, l] ← ∞
    costs[sl(x_V)] ← 0
    s_o ← s(x_V), s_h ← planning horizon
    for s ← (s_0 + 1) to s_h
        for l ← l_min to l_max
            x_F ← x(s, l)                    // x(s, l): map (s, l) to world coords
            c ← ∞
            for (s_a, l_a) in actions
                x_I = x(s − s_a, l − l_a)
                // sl(x): map x to (s, l) space
                // cost(x_I, x_F): cost of traversing between states
                // (time, proximity to obstacles, etc.)
                c ← min(c, cost(x_I, x_F) + costs[sl(x_I)])
            costs[sl(x_F)] ← c
```

Figure 24: A dynamic-programming search algorithm to find the lowest-cost path through a state lattice

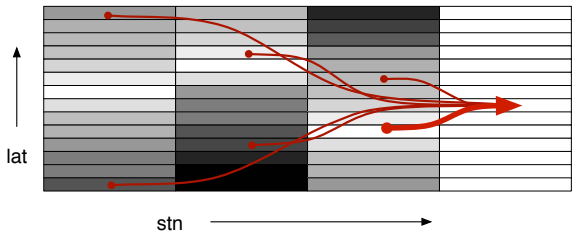


Figure 25: Dynamic programming update step in the state lattice. The cost of each incoming action to a state is added to the lowest known cost to reach its origin state, and the minimum over all incoming costs is taken as the lowest cost to reach the state.

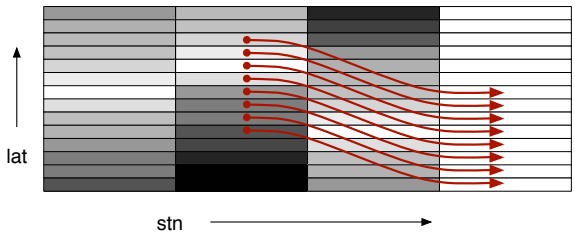


Figure 26: Dynamic programming update step in the state lattice using vector operations. The costs of states at the same station can be computed in parallel by considering the same-shaped action incoming to each state at once.

programming computation by a GPU. Each thread in a warp of the GPU can compute the cost of the same action leading out from states with consecutive latitudes.

Since it is infeasible to plan within the lattice space to the ultimate goal, which may be hundreds of kilometers away or more, we choose the lowest-cost path to the most distant station in the table. That is, we first find the minimum cost table entry for any latitude at s_h, the furthest station within the planning horizon, and then recursively check which action provided the minimum cost to reach the entry.

5.4.4 Cost Functions

The nature of the path to the planning horizon chosen by the planner, that is, the path with the minimum cost, depends on the cost function. The state lattice approach allows us to derive approximately globally-optimal paths that can be followed exactly by the robot. An additional benefit for systems where dynamics are important is that cost functionals that would be difficult to to minimize using a continuum trajectory optimization method can be approximated discretely, limited by the sampling and connectivity of states in the lattice. The cost function need only satisfy Bellman's Priniple of Optimality[8]. Typically a cost function takes the form of a weighted sum of terms

$$\sum_i w_i c_i(\mathbf{x_I}, \mathbf{u}(t)),$$

where each term assigns a numerical value to the path taken by the vehicle, as represented by the control trajectory $\mathbf{u}$ applied starting from $\mathbf{x}$ to the planning horizon. We experimented with terms expressing

- total length of the path $(pathlen)$
- integrated square of lateral acceleration over the path $(latmag^2)$
- integrated lateral deviation of position from the center of the road $(laterr)$

Paths that touch an obstacle are given a cost of ∞.

A desirable trait for semi-autonomous as opposed to completely autonomous driving is that the vehicle should normally be under the driver's direct control. However,

Planning Stage	Nvidia GTX 260	Intel CPU (1 core)	Intel CPU (4 cores, est.)
Lattice generation	30 ms / meter	3 s / meter	750 ms /meter
Planning	80 ms / 100 meters	2 secs / 100 meters	500 ms / 100 meters

Table 4: Comparison of times between CPU and GPU to compute lattice parameters and perform path planning search through the lattice.

when an obstacle suddenly presents itself to the driver, the car should automatically apply the brakes. If the vehicle predicts that it would nevertheless strike the obstacle unless it swerves, it should do so at the last possible moment. In future work we will experiment with cost functions that express this behavioral preference.

5.4.5 Experiments

We implemented the StateLatticeDP() algorithm shown in Figure 24 for the CPU and in CUDA for the Nvidia GPU.

We used an action set with approximately 190 actions, representing hard swerves within the same curvature and curvature rate limits used for Boss. Figure 27 illustrates the displacements of the available actions. For our lateral discretization of 0.1 m, and our 10-meter wide roadway, There are $19\ 000$ individual sets of action parameters to compute via the algorithm $\mathrm{FindActionParameters}$ of Figure 22, *per meter of roadway*. These actions need only be computed once as each 1 meter stretch of roadway comes into the planning horizon, and they can be reused through multiple planning cycles until the vehicle leaves that part of the roadway behind.

Experiments were conducted on a curved road 10 m wide and 400 m long. The state lattice had 401 by 101 vertices, with a total of approximately $400 \cdot 100 \cdot 190 \approx 8\mathrm{M}$ edges. Collision checking was done by approximating the vehicle shape with a binary occupancy grid with the same dimensions as the state lattice. A diagram of the road course on which tests were run is shown in Figure 28. It begins with a few quick curves and transitions to a long curve right followed by a long curve left. Cones are placed at various points along the course. For these tests a single plan is composed before the vehicle begins moving, and it follows the plan to completion. The velocity profile $v(s)$ begins with a rolling start of 1 km /h and accelerates to 50 km/h.

We implemented the lattice generation using FindActionParameters() (Figure 22) and the search algorithm StateLatticeDP(). Tests were run on 2 different GPUs, an 8600 GTS with 32 scalar cores, and a GTX 260 with 216 scalar cores. The CPU used was a single core of an Intel Core 2 Quad 2.3 GHz. Table 4 shows the timing results. Since both the lattice generation and planning algorithms are highly parallel, we estimate that performance on the CPU should increase linearly with the number of cores used. The GTX 260 is able to construct the lattice at the rate of 30 ms per meter. A vehicle travelling at 100 km/h covers one meter in approximately 35 ms, meaning that a current-generation GPU would be able to construct the lattice in real-time. The planning latency of 80 ms per 100 meters allows a global plan extending well beyond a reasonable stopping distance of 100 m for a vehicle travelling at 100 km/h with fair tires on a dry road.

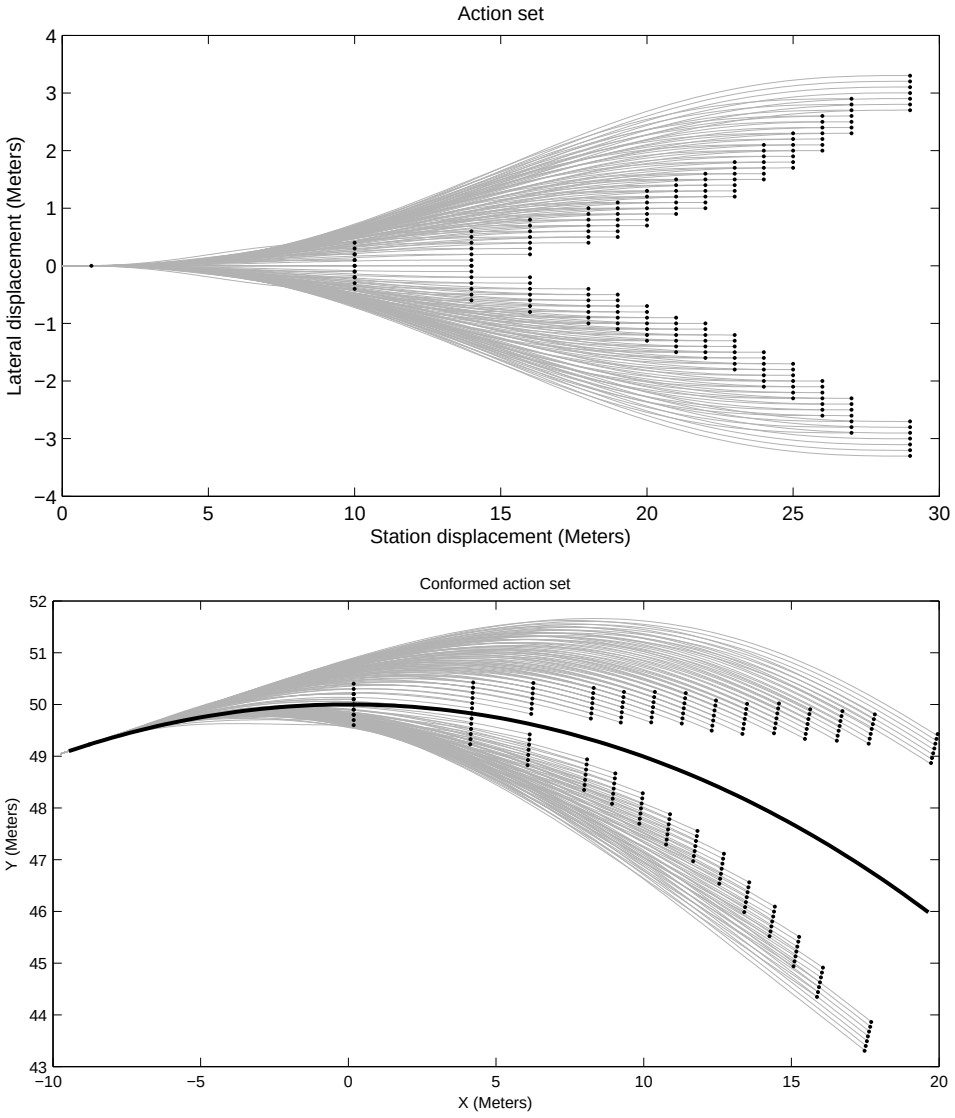


Figure 27: Top figure: displacements in (s, l)-space of the actions used in the second lattice planner experiment. Curves illustrate nominal trajectories rendered onto a straight road. Bottom figure: actions with the same (s, l) displacements warped onto a curved roadway. The thick line marks a lateral displacement of zero.

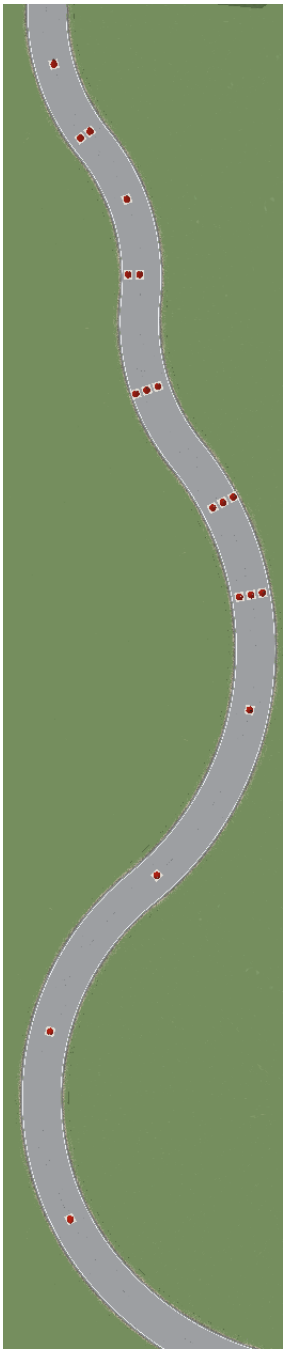


Figure 28: The road course used for state lattice experiments, driven from top to bottom. The road is 10m wide. Circles on the roadway are cones, expanded for visibility.

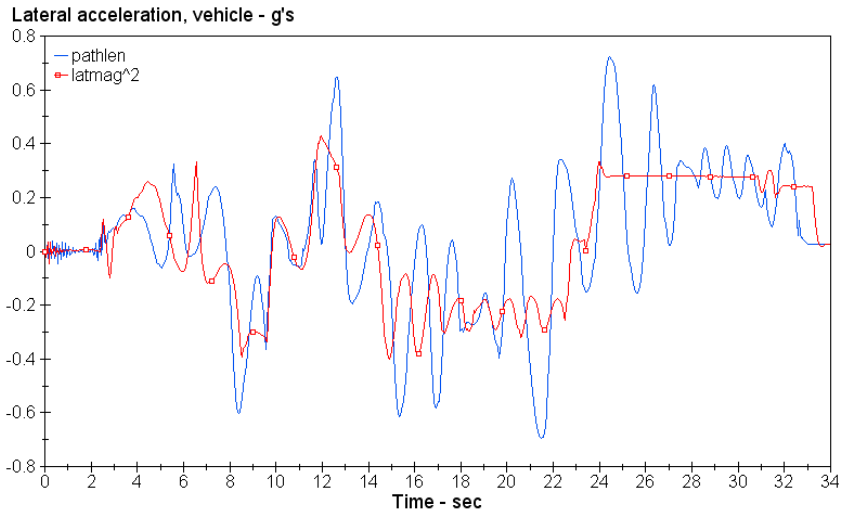


Figure 29: Lateral acceleration undergone by vehicle following path selected to minimize length vs. path minimizing sum of squared lateral acceleration.

To test and visualize the quality of the plans, we programmed a simulated car to follow the plans. We used CarSim[17], a mechanically realistic car simulator. The feed-forward curvature in the plan is applied to the steering wheel using the nominal steering ratio listed for the CarSim model, increasing as a function of vehicle speed. A feedback path-tracking controller based on the "Stanley method"[85] is used to correct for discrepancies between the simple kinematic bicycle model used to construct the path and the realistic CarSim vehicle model used to follow it.

Figure 29 shows the lateral acceleration undergone by the vehicle when following two different paths. The first path is selected to minimize overall length, and the second is selected to minimize the squared lateral acceleration integrated over the path. The path chosen to minimize overall length selects actions with high lateral accelerations. The planner is also successful in finding a path that minimizes lateral acceleration, as the figure shows.

Figure 30 shows the lateral deviation of the executed path from the planned path. Tracking performance depends on the feasibility of the path in terms of vehicle capabilities, and the quality of the tracking controller. Performance is worst at lower speeds(0m-80m) and at constant-curvature sections (190m-250m, 290m-390m), suggesting that the quality of the path tracking controller is responsible, and not the feasibility of the planned path itself.

Figure 31 shows several paths found by the lattice planner using several different cost functions. The green path shows the path with lowest overall length. Note that it deviates from a perfect racing line because the lattice does not contain edges encoding long smooth traverses across a range of latitudes. We can thus see that our lattice was implicitly constructed to be a better structure for lane-keeping. The blue line shows the path resulting from a minimization of the integrated squared lateral acceleration. It

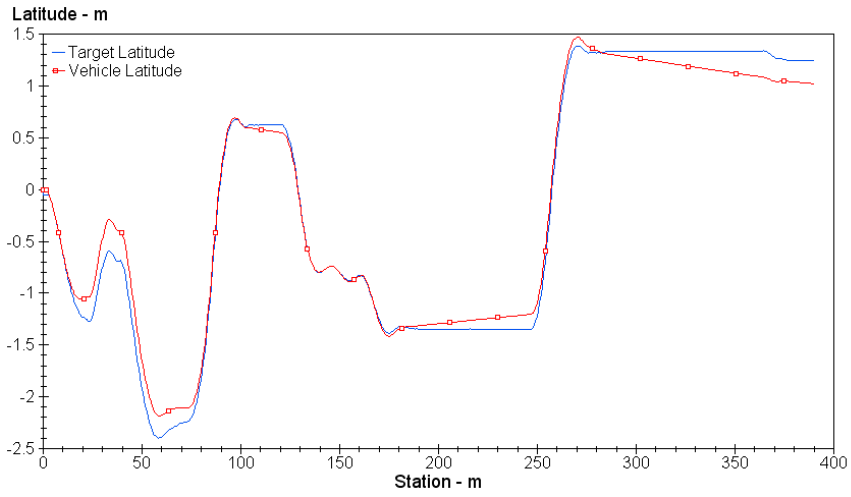


Figure 30: Path-tracking performance of the vehicle following a planned path.

cuts some corners, but compared to the green path, it does not attempt to find seek the inside lane. The red line results from a minimization of the integrated lateral deviation of the vehicle from the center of the road. When forced out of the center by a cone, it quickly chooses to swerve back, whereas the blue path maintains a line that requires less steering. The black dashed path was derived using a weighted sum of the previous three cost functions, balancing the desire to simultaneously minimize path length, lateral acceleration, and position deviation from the center of the road.

We can see from these experiments that once a cost function is chosen, it may be beneficial to modify the sampling strategy for the state lattice vertices to reduce the cost of the best path through the lattice.

5.4.6 Summary

We have shown that the state lattice is a feasible approach to creating complex paths for driving on-road at high speeds. We have also shown that the GPU is able to accelerate construction and search on a lattice conformed to a curving road, sufficiently so that the entire procedure may be carried out in real-time on a fast-moving vehicle. Finally, we have demonstrated that a variety of cost functions can be reasonably satisfied by the optimal paths found within the state lattice.

In the next section we propose a variety of extensions to our work with the state lattice as well as the algorithms described in previous sections.

6 Proposed Work

The theme of the proposed work is to develop algorithms for contemporary vector-parallel processing architectures that accelerate motion planning tasks. We are inter-

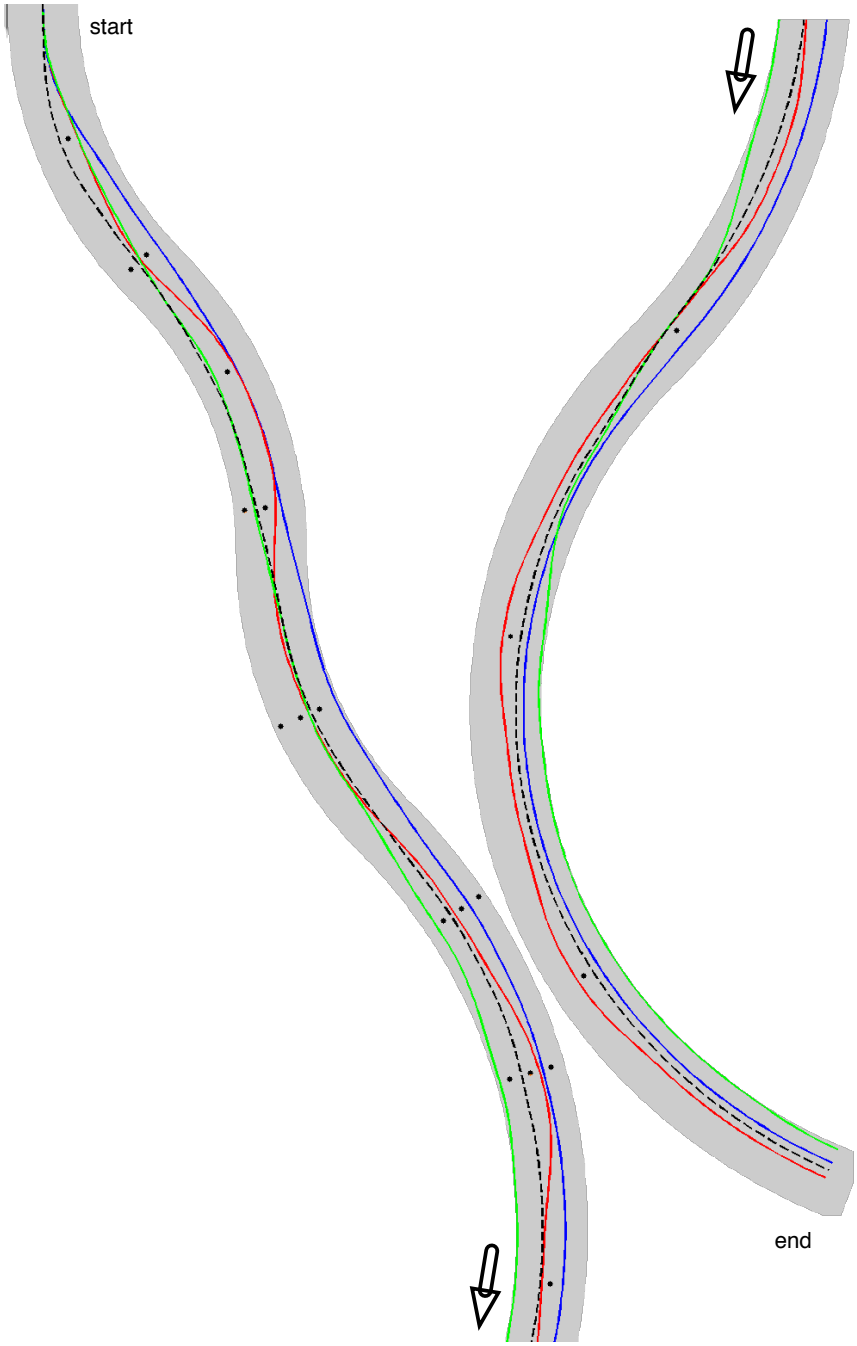


Figure 31: The paths resulting from four cost functions. Red: minimize integrated lateral deviation from road center line. Blue: minimize integrated square of lateral acceleration. Green: minimize overall path length. Black, dashed - linear combination of previous three.

ested both in parallel versions of existing sequential algorithms and in developing new motion planning algorithms that exploit the additional processing capacity of contemporary GPUs. We aim to use GPUs to attain higher planning performance for robotic vehicles than the existing state of the art, in terms of improved plan quality, and the time required to generate an acceptable plan. We intend to demonstrate our developments on an autonomous passenger vehicle.

6.1 Algorithms to be Developed

6.1.1 Parallel Exact Distance Transform

As discussed in Section 3.4.4, no parallel exact distance transform algorithm has been developed for the GPU. We have done preliminary work in developing an algorithm using the primary insight made by Felzenszwalb and Huttenlocher[28] augmented by the parallel prefix scan method used by Lee et al.[50] in their parallel exact method, and Rong and Tan with their "jump flooding"[79] approximation. Additional speedups using vector operations should be possible, while retaining the ability to generalize to higher dimensions. Huttenlocher and Felzenszwalb's method generalize easily to arbitrary dimensions, while Lee et al.'s does not. A two-step algorithm using the insight of their method applied to the result of a bounded approximate method may be a productive direction. We intend to continue our work on this problem.

6.1.2 Parallel Single-Source Shortest Path

In Section 5.3 we presented GPU-Δ-sliding, a parallel single-source shortest path algorithm based on Δ-stepping[60] suitable for the GPU architecture. This algorithm is inefficient when a wide range of edge costs are present in the graph. This can be remedied by tuning the optimal value of Δ at each iteration using parallel algorithms for computing the kth largest element of an array of numbers. Since we developed our algorithm, GPUs with more sophisticated intra-block communication and synchronization primitives to coordinate between threads have become available. We propose to investigate whether more capable parallel hardware is likely to allow a faster variant of the algorithm to be developed.

Finally, we will compare our GPU-Δ-sliding SSSP algorithm to the parallel SSSP algorithms for the GPU presented by Weber et al.[94] and Harish and Narayanan[37].

6.1.3 Focused Kinematic Analysis for A* Heuristic

For A*-based planners, the quality of the heuristic function can have a significant effect on the average-case performance of the planner, even when worst-case guarantees are not improved. While high speed driving on road may not be a suitable application for A* search, with its poor real-time guarantees, a full planner must be able to handle unstructured situations driving at lower speeds as well. A major cause of slow A* planning cycles is the presence of cluttered bottlenecks in the occupancy grid that appear to be navigable to the ball heuristic, but which are not actually navigable when the full vehicle kinematics or dynamics are considered. This can cause the A* search to expand a large number of nodes in an attempt to prove that there is no way to pass

through the gap from any node whose cost-to-go entry in the heuristic table leads towards the gap. FAHR[59] investigated a method of detecting and mitigating the effect of these gaps. The kinematic vehicle model used was simplistic. I propose to apply the focused heuristic enhancement used in FAHR to the more realistic kinematics used in a lattice planner for unstructured environments.

6.1.4 Kinodynamic Lattice Planner

In Section 5.4 we demonstrated the feasibility of a real-time lattice planner with dynamic motions, using a simulated car.

We have not yet experimented with dynamic obstacles in conjunction with a lattice planner. Whereas in our work so far we have assumed a velocity profile given as a unique function $v(s)$, in order for the robot to interact appropriately with dynamic obstacles the planner will need to consider independently varying the velocity at which the robot travels. Both time and velocity dimensions would have to be added to the state space in order to effect collision checking against dynamic obstacles. The size of the velocity dimension for an on-road vehicle would be at least the number of distinct velocity values taken on as it passes over each discrete station value used in the lattice. For a vehicle with a speed limit of 100 km/h and a reasonable acceleration time of 10 s to reach 100 km/h from a stop, driving on a lattice with 1 m longitudinal spacing, the number of discrete velocity values needed for the lattice would be equal to the number of meters travelled while accelerating from a standing start up to full speed. Assuming a constant acceleration so that $v = at$, $p = vt = \frac{1}{2}at^2$, and with final time $t_f = 10\text{s}$ to reach 100 km/h

$$\begin{aligned} a &= v(t_f)/t_f && (8)\\ &= \frac{100\text{km/h}}{10\text{s}} = 2.78\text{m/s}^2, && (9)\\ p(t_f) &= \left(\frac{1}{2}\right)\left(2.78\frac{m}{s^2}\right)(10s)^2 = 139m, && (10) \end{aligned}$$

meaning that a nearly 150 distinct values would be needed for a velocity dimension in the state lattice in the naïve case. The time dimension would multiply the size of the lattice by a similar factor. This space, while not intractable, is too large for real-time planning.

Drawing inspiration from Tompkins's[87] distinction between independent and dependent state dimensions, we propose instead to augment the state space with time and velocity as dependent dimensions. We continue to define the independent dimensions in the state lattice as $\mathbf{x} = (x, y, \theta, \kappa)$, constraining actions to start and end at discrete lattice points. Associated with each node we keep a set of vectors for the dependent states $d_i = (t_i, v_i, c_i)$, each giving a real-valued time and velocity at which the robot could reach $\mathbf{x}$, and the minimum cost c_i to do so. When actions leading out of two states $\mathbf{x}_1$, $\mathbf{x}_2$ reach the same terminal state $\mathbf{x}_3$, the d_i vectors from each of them are propagated forward according to the durations and accelerations of the actions applied. If multiple such vectors fall into the same resolution-equivalence bucket, the one with lowest cost is kept.

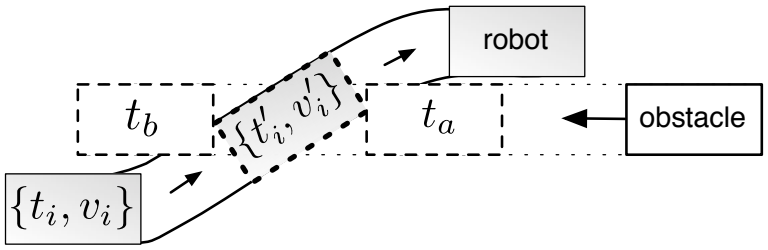


Figure 32: Illustration of proposed dynamic collision checking in lattice. If $t_a < t_i' < t_b$, then the robot would collide with the obstacle and the corresponding dependent state vector can be culled.

The dependent (v, t, c) vectors must be propagated forward with some sampling of accelerations. These could be selected to approximate moderate, hard, and panic braking, as well as zero acceleration and a small range of mild to hard throttle commands. The product $(\#d)(\#a)$ of the number of dependent vectors and the number of accelerations would seem to inflate the amount of work to be done to unmanageable levels, but there are two tricks available to reduce the effort. First, we constrain the actual path driven by the robot to be purely kinematic, that is, independent of its velocity. The robot may not be able to satisfy this constraint when executing the path. This execution error can be approximated by dilating the c-space obstacles appropriately. Then, given that the robot will occupy the same sequence of poses $\mathcal{A}_i$ regardless of the time at which it arrives at them and the speed at which it drives over them, for each dynamic obstacle o_j, we can compute the time interval $[t_a, t_b] = \mathcal{A}_i \cap o_j(t)$ during which it is expected to intersect with the pose. The dependent state vectors can then be quickly filtered according to the time at which they schedule the robot to occupy that pose. A collision with a static obstacle eliminates all of them. Figure 32 illustrates.

We propose to explore further how best to sample a lattice to achieve the greatest likelihood that a near-optimal path exists in the lattice, and how to select the pattern of connectivity in the lattice. Following the example of Pivtoraiko et al.[75], we will experiment with the selection of actions in the lattice to improve the quality of the resulting plans while keeping the number of actions low. A further issue, which has not been much explored in the literature, is how to sample the lattice using all dimensions in (x, y, θ, κ) while minimizing the state space explosion.

6.2 Robot Implementation

I propose to implement the GPU-based planning suite described above on a real robotic vehicle, and test it at highway speeds. The planner will allow the robot to straddle short or negative obstacles (e.g. potholes). I will show that it has shortened planning latency over previous solutions, indicating better safety. It will also plan through complex situations while maintaining reasonable speed.

I will implement this planner on a series of platforms, beginning with the mechanical simulator CarSim.

7 Timeline

Immediately following the proposal, I plan to work at Google Inc on an internship for the fall semester. I plan to complete the work of this thesis over the course of the year following. The schedule is as follows:

Fall 2009

- Internship at Google.

Winter 2010

- Global path planning using a state lattice, accounting for vehicle dynamics, with static obstacles.
- Implement a simple planner with static obstacles and a curved road on Boss.
- Global path planning, straddling short obstacles.

Spring 2010

- Local kinematic analysis using actions on state lattice.
- Complete parallel single source shortest path.
- Parallel distance transform using SIMD operations and derived parallel generalized voronoi diagram.

Summer 2010

- Experiment with integrating velocity into state space for lattice planner.
- Global path planning on lattice using dynamic obstacles with fixed trajectories.

Fall 2010

- Extensive experiments on real robot.
- Write up dissertation and defend.

8 Contributions

This thesis will contribute new parallel algorithms for motion planning on many-core computers using the data-parallel operations they offer. We will develop a parallel single-source shortest path and distance transform that will be used to accelerate the computation of heuristic and cost functions. We will also develop a real-time planner for global motions and demonstrate it on an autonomous vehicle driving at high speeds. We will show how the additional computations that can be performed by the GPU can be used to achieve higher plan fidelity and a more rich expression of the desired results.

References

[1] Technical overview: ATI stream computing. Technical report, AMD Corporation, 2009.

[2] National Highway Traffic Safety Administration. 2007 traffic safety annual assessment – highlights, August 2008. DOT HS 811 017.

[3] N.M. Amato and L.K. Dale. Probabilistic roadmap methods are embarrassingly parallel. In *Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on*, volume 1, pages 688–694 vol.1, 1999.

[4] Andrew Bacha, Cheryl Bauman, Ruel Faruque, Michael Fleming, Chris Terwelp, Charles F. Reinholtz, Dennis Hong, Al Wicks, Thomas Alberi, David Anderson, Stephen Cacciola, Patrick Currier, Aaron Dalton, Jesse Farmer, Jesse Hurdus, Shawn Kimmel, Peter King, Andrew Taylor, David Van Covern, and Mike Webster. Odin: Team victortango's entry in the darpa urban challenge. *J. Field Robotics*, 25(8):467–492, 2008.

[5] Daniel L. Baggio. GPGPU based image segmentation livewire algorithm implementation. Master's thesis, São José dos Campos, SP, Brazil, 2007.

[6] Jerome Barraquand and Jean-Claude Latombe. Robot Motion Planning: A Distributed Representation Approach. *The International Journal of Robotics Research*, 10(6):628–649, 1991.

[7] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, Liewei Bao, J. Brown, M. Mattina, Chyi-Chang Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. Tile64 - processor: A 64-core soc with mesh interconnect. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 88–598, Feb. 2008.

[8] R. E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[9] Jonathan Bohren, Tully Foote, Jim Keller, Alex Kushleyev, Daniel D. Lee, Alex Stewart, Paul Vernaza, Jason C. Derenick, John R. Spletzer, and Brian Satterfield. Little ben: The ben franklin racing team's entry in the 2007 darpa urban challenge. *J. Field Robotics*, 25(9):598–614, 2008.

[10] O. Brock and O. Khatib. Real-time re-planning in high-dimensional configuration spaces using sets of homotopic paths. In *Robotics and Automation, 2000. Proceedings. ICRA '00. IEEE International Conference on*, volume 1, pages 550–555 vol.1, 2000.

[11] D. Challou, D. Boley, M. Gini, and V. Kumar. A parallel formulation of informed randomized search for robot motion planning problems. In *Robotics and Automation, 1995. Proceedings., 1995 IEEE International Conference on*, volume 1, pages 709–714 vol.1, May 1995.

[12] Nicholas Chan, James Kuffner, and Matthew Zucker. Improved motion planning speed and safety using regions of inevitabl e collision. In *17th CISM-IFToMM Symposium on Robot Design, Dynamics, and Contro l (RoManSy'08)*, July 2008.

[13] Yi-Liang Chen, Venkataraman Sundareswaran, Craig Anderson, Alberto Broggi, Paolo Grisleri, Pier Paolo Porta, Paolo Zani, and John Beck. Terramax$^{\text{tm}}$: Team oshkosh urban robot. *J. Field Robotics*, 25(10):841–860, 2008.

[14] Intel Corporation. Excerpts from a Conversation with Gordon Moore: Moore's Law. `ftp://download.intel.com-/museum/Moores_Law/Video-Transcripts-/Excepts_A_Conversation_with_Gordon_Moore.pdf`.

[15] Intel Corporation. Microprocessor quick reference guide. `http://www.intel.com/pressroom/kits/quickreffam.htm`

[16] Intel Corporation. Sspec/qdf reference. http://ark.intel.com/SSPECQDF.aspx.

[17] Mechanical Simulation Corporation. Carsim. Software.

[18] R. Craig Coulter. Implementation of the pure pursuit path tracking algorithm. Technical Report CMU-RI-TR-92-01, Robotics Institute, Pittsburgh, PA, January 1992.

[19] A. Crauser, K. Mehlhorn, and U. Meyer. A parallelization of dijkstra's shortest path algorithm. In *In Proc. 23rd MFCS'98, Lecture Notes in Computer Science*, pages 722–731. Springer, 1998.

[20] Cray Inc. *Cray XMT Programming Environment User's Guide S247913*, 1.3 edition. http://docs.cray.com/books/S-2479-13/S-2479-13.pdf.

[21] Joseph Culberson and Jonathan Schaeffer. Searching with pattern databases. In *CSCSI '96 (Canadian AI Conference), Advances in Artificial Intelligence*, pages 402–416. Springer-Verlag, 1996.

[22] DARPA. Urban challenge. competition, Nov 2007. `http://www.darpa.mil/grandchallenge/`

[23] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Sixth Symposium on Operating System Design and Implementation (OSDI)*, San Francisco, CA, October 2004.

[24] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, December 1959.

[25] Dmitri Dolgov, Sebastian Thrun, Michael Montemerlo, and James Diebel. Practical search techniques in path planning for autonomous driving. In *Proceedings of the First International Symposium on Search Techniques in Artificial Intelligence and Robotics (STAIR-08)*, Chicago, USA, June 2008. AAAI.

[26] H. Embrechts and D. Roose. A parallel euclidean distance transformation algorithms. In *Parallel and Distributed Processing, 1993. Proceedings of the Fifth IEEE Symposium on*, pages 216–223, Dec 1993.

[27] Michael Erdmann and Tomas Lozano-Perez. On multiple moving objects. pages 1419–1424, 1986.

[28] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. Distance transforms of sampled functions. Technical Report TR2004-1963, Cornell Computing and Information Science, 2004.

[29] Dave Ferguson, Thomas M. Howard, and Maxim Likhachev. Motion planning in urban environments. *Journal of Field Robotics*, 25(11-12):939–960, 2008.

[30] David Ferguson, Thomas Howard, and Maxim Likhachev. Motion planning in urban environments: Part i. In *Proceedings of the IEEE/RSJ 2008 International Conference on Intelligent Robots and Systems*, September 2008.

[31] D. Fox, W. Burgard, and S. Thrun. The dynamic window approach to collision avoidance. *Robotics & Automation Magazine, IEEE*, 4(1):23–33, Mar 1997.

[32] Theirry Fraichard and Hajime Asama. Inevitable collision states: A step towards safer robots? *Advanced Robotics*, 18(10):1001–1024, 2004.

[33] Thierry Fraichard and Vivien Delsart. Navigating dynamic environments with trajectory deformation. *Journal of Computing and Information Technology*, 2009.

[34] Russell Gayle, Paul Segars, Ming C. Lin, and Dinesh Manocha. Path planning for deformable robots in complex environments. In *Proceedings of Robotics: Science and Systems*, Cambridge, USA, June 2005.

[35] D.N. Godbole, V. Hagenmeyer, R. Sengupta, and D. Swaroop. Design of emergency manoeuvres for automated highway system: obstacle avoidance problem. In *Decision and Control, 1997., Proceedings of the 36th IEEE Conference on*, volume 5, pages 4774–4779 vol.5, Dec 1997.

[36] Khronos Group. OpenCL. Computing Standard. `http://www.khronos.org/opencl/`

[37] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *HiPC*, pages 197–208, 2007.

[38] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, July 1968.

[39] Jim Held, Jerry Bautista, and Sean Koehl. From a few cores to many: A tera-scale computing research overview. Intel Corporation. White paper, 2006.

[40] Dominik Henrich. Fast motion planning by parallel processing – a review. *Journal of Intelligent and Robotic Systems*, 20(1):45–69, 1997.

[41] Thomas M. Howard, Colin J. Green, Alonzo Kelly, and Dave Ferguson. State space sampling of feasible motions for high-performance mobile robot navigation in complex environments. *Journal of Field Robotics*, 25(6-7):325–345, 2008.

[42] H. Jula, E.B. Kosmatopoulos, and P.A. Ioannou. Collision avoidance analysis for lane changing and merging. *Vehicular Technology, IEEE Transactions on*, 49(6):2295–2308, Nov 2000.

[43] A. Junghanns and J. Schaeffer. Sokoban: Enhancing general single-agent search methods using domain knowledge. *Artificial Intelligence*, 129(1-2):219 – 251, 2001.

[44] A. Kanaris, E.B. Kosmatopoulos, and P.A. Loannou. Strategies and spacing requirements for lane changing and merging in automated highway systems. *Vehicular Technology, IEEE Transactions on*, 50(6):1568–1581, Nov 2001.

[45] L.E. Kavraki. Computation of configuration-space obstacles using the fast fourier trans form. *Robotics and Automation, IEEE Transactions on*, 11(3):408–413, Jun 1995.

[46] Alonzo Kelly and Bryan Nagy. *Reactive Nonholonomic Trajectory Generation via Parametric Optimal Control*, 22(1):583 – 601, July 2003.

[47] A. Kushleyev and M. Likhachev. Time-bounded lattice for efficient planning in dynamic environments. In *Robotics and Automation, 2009. ICRA '09. IEEE International Conference on*, pages 1662–1668, May 2009.

[48] J. C. Latombe. Motion planning: A journey of robots, molecules, digital actors, and other artifacts. *Int. J. of Robotics Research*, 18(11):1119–1128, November 1999.

[49] S. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.

[50] Yu-Hua Lee, Shi-Jinn Horng, Tzong-Wann Kao, Ferng-Shi Jaung, Yuung-Jih Chen, and Horng-Ren Tsai. Parallel computation of exact euclidean distance transform. *Parallel Computing*, 22(2):311 – 325, 1996.

[51] Brett M. Leedy, Joseph S. Putney, Cheryl Bauman, Stephen Cacciola, J. Michael Webster, and Charles F. Reinholtz. Virginia tech's twin contenders: A comparative study of reactive and deliberative navigation. *J. Field Robotics*, 23(9):709–727, 2006.

[52] Jed Lengyel, Mark Reichert, Bruce R. Donald, and Donald P. Greenberg. Real-time robot motion planning using rasterizing computer graphics hardware. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 327–335, New York, NY, USA, 1990. ACM.

[53] John Leonard, Jonathan How, Seth Teller, Mitch Berger, Stefan Campbell, Gaston Fiore, Luke Fletcher, Emilio Frazzoli, Albert Huang, Sertac Karaman, Olivier Koch, Yoshiaki Kuwata, David Moore, Edwin Olson, Steve Peters, Justin Teo,

Robert Truax, Matthew Walter, David Barrett, Alexander Epstein, Keoni Maheloni, Katy Moyer, Troy Jones, Ryan Buckley, Matthew Antone, Robert Galejs, Siddhartha Krishnamurthy, and Jonathan Williams. A perception-driven autonomous urban vehicle. *J. Field Robot.*, 25(10):727–774, 2008.

[54] Maxim Likhachev and Dave Ferguson. Planning long dynamically-feasible maneuvers for autonomous vehicles. In *Proceedings of Robotics: Science and Systems IV*, Zurich, Switzerland, June 2008.

[55] T. Lozano-Perez and P.A. O'Donnell. Parallel robot motion planning. In *Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on*, pages 1000–1007 vol.2, Apr 1991.

[56] Ambuj Mahanti and Charles J. Daniels. A simd approach to parallel heuristic search. *Artificial Intelligence*, 60(2):243 – 282, 1993.

[57] L. Martinez-Gomez and T. Fraichard. An efficient and generic 2d inevitable collision state-checker. In *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, pages 234–241, Sept. 2008.

[58] B. Martinez-Salvador, A.P. del Pobil, and M. Perez-Francisco. Very fast collision detection for practical motion planning. i. the spatial representation. In *Robotics and Automation, 1998. Proceedings. 1998 IEEE International Conference on*, volume 1, pages 624–629 vol.1, May 1998.

[59] Matthew McNaughton and Chris Urmson. Fahr: Focused a* heuristic recomputation. In *Proceedings of the 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2009. to appear.

[60] U. Meyer. Delta-stepping: A parallel single source shortest path algorithm. In *In ESA 98: Proceedings of the 6th Annual European Symposium on Algorithms*, pages 393–404. Springer-Verlag, 1998.

[61] Isaac Miller, Mark Campbell, Dan Huttenlocher, Frank-Robert Kline, Aaron Nathan, Sergei Lupashin, Jason Catlin, Brian Schimpf, Pete Moran, Noah Zych, Ephrahim Garcia, Mike Kurdziel, and Hikaru Fujishima. Team cornell's skynet: Robust perception and planning in an urban environment. *J. Field Robot.*, 25(8):493–527, 2008.

[62] Javier Minguez and L. Montano. Nearness diagram (nd) navigation: collision avoidance in troublesome scenarios. *Robotics and Automation, IEEE Transactions on*, 20(1):45–59, Feb. 2004.

[63] Michael Montemerlo, Jan Becker, Suhrid Bhat, Hendrik Dahlkamp, Dmitri Dolgov, Scott Ettinger, Dirk Haehnel, Tim Hilden, Gabe Hoffmann, Burkhard Huhnke, Doug Johnston, Stefan Klumpp, Dirk Langer, Anthony Levandowski, Jesse Levinson, Julien Marcil, David Orenstein, Johannes Paefgen, Isaac Penny, Anna Petrovskaya, Mike Pflueger, Ganymed Stanek, David Stavens, Antone Vogt, and Sebastian Thrun. Junior: The stanford entry in the urban challenge. *Journal of Field Robotics*, 25(9):569–597, 2008.

[64] Michael Montemerlo, Jan Becker, Suhrid Bhat, Hendrik Dahlkamp, Dmitri Dolgov, Scott Ettinger, Dirk Haehnel, Tim Hilden, Gabe Hoffmann, Burkhard Huhnke, Doug Johnston, Stefan Klumpp, Dirk Langer, Anthony Levandowski, Jesse Levinson, Julien Marcil, David Orenstein, Johannes Paefgen, Isaac Penny, Anna Petrovskaya, Mike Pueger, Ganymed Stanek, David Stavens, Antone Vogt, and Sebastian Thrun. The stanford entry in the urban challenge. *Journal of Field Robotics*, 2008.

[65] Michael Montemerlo, Jan Becker, Suhrid Bhat, Hendrik Dahlkamp, Dmitri Dolgov, Scott Ettinger, Dirk Hähnel, Tim Hilden, Gabe Hoffmann, Burkhard Huhnke, Doug Johnston, Stefan Klumpp, Dirk Langer, Anthony Levandowski, Jesse Levinson, Julien Marcil, David Orenstein, Johannes Paefgen, Isaac Penny, Anna Petrovskaya, Mike Pflueger, Ganymed Stanek, David Stavens, Antone Vogt, and Sebastian Thrun. Junior: The stanford entry in the urban challenge. *J. Field Robotics*, 25(9):569–597, 2008.

[66] Intel Corporation Museum. Product category list. `http://download.intel.com/museum/research/arc_collect-/timeline/TimelineProductTypeSort7_05.pdf`.

[67] N. J. Nilsson. *Principles of Artificial Intelligence*. 1980.

[68] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*, version 2.0 edition, June 2008.

[69] Colm Ó'Dúnlaing and Chee-Keng Yap. A "retraction" method for planning the motion of a disc. *J. Algorithms*, 6(1):104–111, 1985.

[70] I. Papadimitriou and M. Tomizuka. Fast lane changing computations using polynomials. *American Control Conference, 2003. Proceedings of the 2003*, 1:48–53 vol.1, 4-6 June 2003.

[71] Benjamin J. Patz, Yiannis Papelis, Remo Pillat, Gary Stein, and Don Harper. A practical approach to robotic design for the darpa urban challenge. *J. Field Robotics*, 25(8):528–566, 2008.

[72] M. Perez-Francisco, A.P. del Pobil, and B. Martinez-Salvador. Very fast collision detection for practical motion planning. ii. the parallel algorithm. In *Robotics and Automation, 1998. Proceedings. 1998 IEEE International Conference on*, volume 1, pages 644–649 vol.1, May 1998.

[73] D. Pham, S. Asano, M. Bolliger, M.N. Day, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation cell processor. In *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, pages 184–592 Vol. 1, Feb. 2005.

[74] M. Pivtoraiko and A. Kelly. Efficient Constrained Path Planning via Search in State Lattices. In *'i-SAIRAS 2005' - The 8th International Symposium on Artificial Intelligence, Robotics and Automation in Space*, volume 603 of *ESA Special Publication*, August 2005.

[75] Mikhail Pivtoraiko, Ross Alan Knepper, and Alonzo Kelly. Differentially constrained mobile robot motion planning in state lattices. *Journal of Field Robotics*, 26(1):308–333, March 2009.

[76] C. Raphael. Coarse-to-fine dynamic programming. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 23(12):1379–1390, Dec 2001.

[77] Nathan Ratliff, Matt Zucker, J. Andrew Bagnell, and Siddhartha Srinivasa. CHOMP: Gradient optimization techniques for efficient motion planning. In *Proceedings of the International Conference on Robotics and Automation(ICRA)*, 2009.

[78] Fred W. Rauskolb, Kai Berger, Christian Lipski, Marcus A. Magnor, Karsten Cornelsen, Jan Effertz, Thomas Form, Fabian Graefe, Sebastian Ohl, Walter Schumacher, Jörn-Marten Wille, Peter Hecker, Tobias Nothdurft, Michael Doering, Kai Homeier, Johannes Morgenroth, Lars C. Wolf, Christian Basarke, Christian Berger, Tim Gülke, Felix Klose, and Bernhard Rumpe. Caroline: An autonomously driving vehicle for urban environments. *J. Field Robotics*, 25(9):674–724, 2008.

[79] Guodong Rong and Tiow-Seng Tan. Jump flooding in gpu with applications to voronoi diagram and distance transform. In *I3D '06: Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, pages 109–116, New York, NY, USA, 2006. ACM.

[80] Azriel Rosenfeld and John L. Pfaltz. Sequential operations in digital picture processing. *J. ACM*, 13(4):471–494, 1966.

[81] Martin Rufli, Dave Ferguson, and Roland Siegwart. Smooth path planning in constrained environments. In *ICRA 2009*, 2009.

[82] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[83] T. Shibata and T. Fukuda. Coordinative behavior by genetic algorithm and fuzzy in evolutionary multi-agent system. In *Robotics and Automation, 1993. Proceedings., 1993 IEEE International Conference on*, pages 760–765 vol.1, May 1993.

[84] Sanjiv Singh, Reid Simmons, Trey Smith, Anthony (Tony) Stentz, Vandi Verma, Alex Yahja, and Kurt Schwehr. Recent progress in local and global traversability for planetary rovers. In *Proceedings of the IEEE International Conference on Robotics and Automation, 2000*. IEEE, April 2000.

[85] Jarrod M. Snider. Automatic steering methods for autonomous automobile path tracking. Technical Report CMU-RI-TR-09-08, Robotics Institute, Pittsburgh, PA, February 2009.

[86] A. Stentz. The focussed d* algorithm for real-time replanning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1995.

[87] Paul Tompkins. *Mission-Directed Path Planning for Planetary Rover Exploration*. PhD thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, May 2005.

[88] Jesper L. Träff and Christos D. Zaroliagis. A simple parallel algorithm for the single-source shortest path problem on planar digraphs. In *IRREGULAR*, pages 183–194. Springer, 1996.

[89] Jesper L. Träff and Christos D. Zaroliagis. A simple parallel algorithm for the single-source shortest path problem on planar digraphs. *J. Parallel Distrib. Comput.*, 60(9):1103–1124, 2000.

[90] Arthur Trew and Greg Wilson, editors. *Past, Present, Parallel : a Survey of Available Parallel Computing Systems*. Springer-Verlag, 1991.

[91] Allen B. Tucker, editor. *Computer Science Handbook*, chapter Parallel Algorithms (10). CRC Press, second edition, 2004.

[92] Chris Urmson, Joshua Anhalt, Drew Bagnell, Christopher R. Baker, Robert Bittner, M. N. Clark, John M. Dolan, Dave Duggins, Tugrul Galatali, Christopher Geyer, Michele Gittleman, Sam Harbaugh, Martial Hebert, Thomas M. Howard, Sascha Kolski, Alonzo Kelly, Maxim Likhachev, Matthew McNaughton, Nick Miller, Kevin Peterson, Brian Pilnick, Raj Rajkumar, Paul E. Rybski, Bryan Salesky, Young-Woo Seo, Sanjiv Singh, Jarrod Snider, Anthony Stentz, William Whittaker, Ziv Wolkowicki, Jason Ziglar, Hong Bae, Thomas Brown, Daniel Demitrish, Bakhtiar Litkouhi, Jim Nickolaou, Varsha Sadekar, Wende Zhang, Joshua Struble, Michael Taylor, Michael Darms, and Dave Ferguson. Autonomous driving in urban environments: Boss and the urban challenge. *J. Field Robotics*, 25(8):425–466, 2008.

[93] J. Viitanen and J. Takala. SIMD parallel calculation of distance transformations. In *Image Processing, 1994. Proceedings. ICIP-94., IEEE International Conference*, volume 3, pages 645–649 vol.3, Nov 1994.

[94] Ofir Weber, Yohai S. Devir, Alexander M. Bronstein, Michael M. Bronstein, and Ron Kimmel. Parallel algorithms for approximation of distance maps on parametric surfaces. *ACM Trans. Graph.*, 27(4):1–16, 2008.