ECE/CS 508: Manycore Parallel Algorithms
Final Project Report
GPU-based Collision Detection for Robot Motion Planning

Nandeeka Nayak (ndnayak2), Victor Murta (vmurta2),
Tommaso Bassetto (Riley) (tommaso3), Aakash Hegde (aakashh2)

1. Introduction and Motivation

Motion planning is the problem of finding a sequence of valid configurations for some object from a start point to a goal paint. It is an important area of research in robotics and is relevant in a wide range of applications, including automated assembly, space exploration, automated vehicles, computational biology (e.g., protein folding), gaming and virtual reality — just about any situation where you want to move a physical object from one configuration to another. The best deterministic algorithms for the motion planning are provably PSPACE-Hard, and thus all algorithms used in practice are sampling-based.

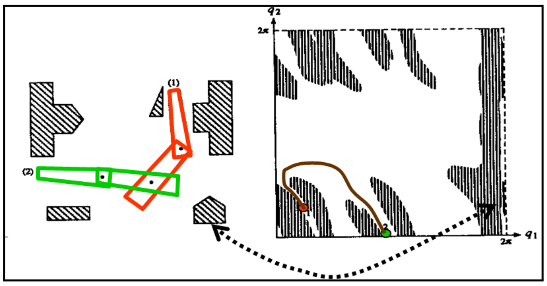
A fundamental concept in motion planning is that of the Configuration Space. The configuration space of a robot is the space of all possible configurations (or poses) that the robot can take within its environment. The configuration space is defined by the range of possible values for each degree of freedom of a robot, and we define the following subspaces of the base configuration space:

- C_{space} - set of all possible robot placements
- C_{free} - set of all valid placements
- C_{obst} - set of all invalid placements

Describing the exact boundaries of C_{obst} is an immensely difficult task, since it requires tracing obstacles in the environment and accounting for all potential collisions that a robot may encounter as it moves within the workspace. Sampling-based motion planning provides a practical alternative, and is the algorithm we explore in this project.

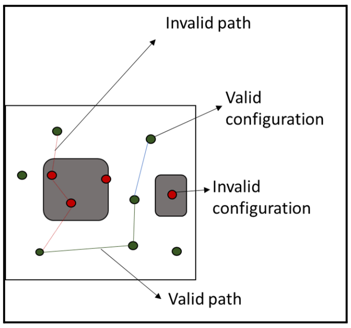
1.1. Existing Methodologies and Challenges

As mentioned before, motion planning is PSPACE-hard [1]. The best deterministic algorithm has a running time that is exponential in degrees of freedom of the robot [2]. C_{space} is high dimensional - 6-D C_{space} for a rigid body in 3-D workspace, and growing exponentially for any added degrees of freedom. Additionally, even simple obstacles have complex C_{obst}. It is, therefore, impractical to compute explicit representations of free space for robots consisting of more than 4 or 5 degrees of freedom. Many planning algorithms exist, but struggle to perform efficiently in high dimensions. In light of the computational complexity involved in motion planning, the focus has shifted towards randomized algorithms that trade-off full completeness of the planner for probabilistic completeness, resulting in a significant improvement in efficiency.



The probabilistic roadmap (PRM) planner is a popular motion planning algorithm that follows this approach. It approximates C_{free} by iteratively sampling a random points in C_{space}, checking if that point is valid, and if it is, connecting it to other nearby valid points. Once some threshold of valid points have been sampled, PRM does a basic shortest path search using standard graph algorithms to try to find a path from start to goal. This method is probabilistically complete, meaning that as the number of sampled points approaches infinity, the probability of finding a path from start to goal (if one exists) approaches 1. Probabilistic roadmaps are widely used due to their ability to generate plans quickly and effectively, even in high-dimensional spaces.

1.2. Collision Detection



Collision detection is the process of determining a robot configuration p is valid by checking if the robot is in collision with any obstacles in its workspace at that point, that is, if the robot would be physically in contact with or overlapping an obstacle at that configuration. Collision detection is used to determine whether the configuration lies in the free configuration space (C_{free}) or the obstacle space (C_{obst}). A path between two configurations is valid only if it lies completely in C_{free}, meaning that it avoids all obstacles in the workspace.

Some terminologies related to collision detection, that are used in the following sections of the document, are described below:

- Mesh: Collection of vertices that define the outline of an object and the triangles that connect those vertices
- Configuration: The spatial arrangement of the mesh (6 d.o.f.)
- Transformation: Translation and Rotation of the mesh as per the configuration
- Axis Aligned Bounding Box (AABB): Cuboid that defines the boundaries of the mesh; aligned to the axes of the space

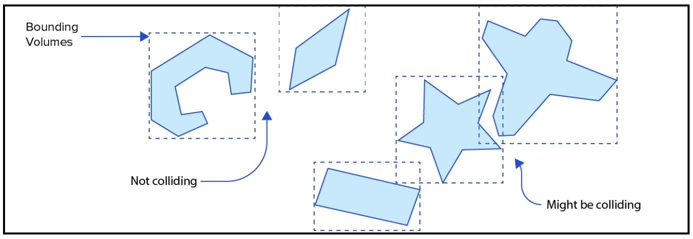
1.2.1. Basic steps in collision detection

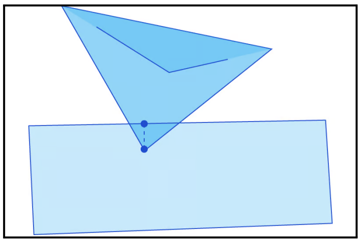
In collision detection, robots and obstacles are represented as meshes. The first step of collision detection is transformation — modifying the vertices of a robot so that the mesh matches the configuration. Then transformed mesh is checked for intersection with the obstacle meshes in two important phases:

  - ○ Broad phase
  - ○ Narrow phase

Broad phase collision detection is an initial step in collision detection that involves checking if the bounding boxes of the robot and obstacles are colliding. By checking for bounding box collisions, it is possible to quickly eliminate configurations that will definitely be collision free, allowing for fewer configurations to be checked in the more compute-intensive narrow phase..

After the broad phase collision detection, the narrow phase collision detection is used to further check for collisions in configurations that were not pruned. This involves checking if the meshes themselves of the robot and obstacles collide, which is the most computationally intensive task in collision detection. The narrow phase ensures that only collision-free configurations are used in motion planning.

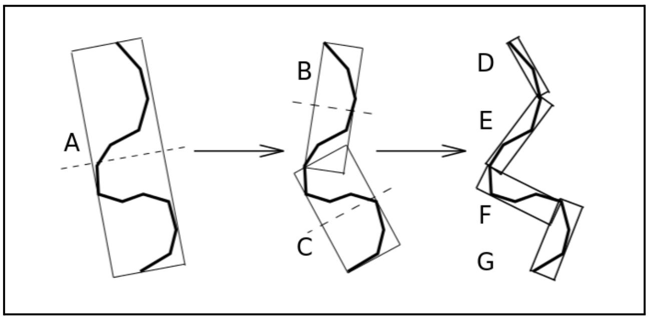




2. Sequential Collision Detection Implementation

Several sequential implementations exist to perform collision detection. One of the popular libraries that we explored and used as the baseline for this project is the FCL (Flexible Collision Library) – an open-source software library for performing collision detection between geometric models. FCL provides efficient algorithms for collision detection between convex shapes and non-convex shapes, such as meshes, using the GJK algorithm. FCL is designed to be flexible and extensible, allowing users to define their own custom collision geometries and use them with the library's collision detection algorithms. It also includes features such as distance computation and continuous collision detection, which are useful for applications such as motion planning and simulation.

The hierarchical bounding box strategy used in FCL is a technique for accelerating collision detection between objects by using a hierarchy of bounding boxes. The bounding boxes are arranged in a tree structure, with the root containing a single bounding box that encloses the entire object, and the children containing smaller bounding boxes that enclose subsets of the object. FCL checks for collisions between pairs of bounding boxes at each level of the hierarchy, recursively descending into the children of intersecting boxes until it reaches the smallest level of the hierarchy, where it performs detailed collision tests between individual geometric primitives. This strategy provides a significant speedup in collision detection for complex objects by efficiently excluding large numbers of objects from collision tests, reducing the overall number of collision tests required and improving the performance of the collision detection algorithm.



Overall, FCL is a powerful and widely used library for collision detection, with a large community of users and contributors. Its flexibility and extensibility make it a popular choice. However, it is not designed to take advantage of the massive computational capabilities of accelerators or GPUs.

3. Parallel Collision Detection Implementation

Accelerating collision detection using GPUs is the primary intent of this project. GPU acceleration is expected to significantly speed up collision detection due to the nature of the problem and the characteristics of GPUs. One of the main advantages of using GPUs for collision detection is that mesh operations are naturally suited to GPU architecture. The large number of processing units can work in parallel to perform the same operation on a large array of points or triangles simultaneously. This parallelism allows for efficient processing of large volumes of data, which is a crucial requirement for collision detection.

Moreover, collision detection is characterized by largely independent tasks, meaning that each pair of objects can be checked for collision independently. This independence makes it an ideal candidate for GPU acceleration, as GPUs are designed to handle massive amounts of independent parallel computation. By offloading the collision detection computations to the GPU, the CPU can focus on other aspects of the simulation, such as physics calculations or user input.

3.1. Kernel Design

Our code has the following signature:

- Input: Configuration array as a point in 6d-space (x, y, z, pitch, roll yaw), Robot mesh (triangles + vertices), Obstacle mesh (triangles + vertices)
- Output: An array of booleans. It has the same length as the configuration array and is True when a configuration is valid.

The first step in our kernel is a transformation step. We take in a configuration and the original robot mesh, and we apply the transformation to get a new robot mesh. After this, we compute an axis-aligned bounding box (AABB) that fully encloses the transformed robot. We then calculate the AABB of the obstacle and do an overlap check between the two cuboids (we call this section the broad phase). If this check passes, we know the configurations cannot overlap.

However, the AABBs overlapping is not a guarantee that the configurations collide. A narrow phase compares all of the triangles in the robot's mesh with all of the triangles in the obstacle's mesh. Triangle-triangle intersection has two phases. In the first, the kernel computes the planes defined by each of the triangles. If a triangle is fully on one side of the other's plane, then the triangles cannot intersect. Otherwise, the kernel computes the intersection line of both planes, the region of the line contained within each triangle, and whether these line segments overlap. If no triangles overlap, then the configuration is guaranteed to be valid.

This algorithm has a number of early-exit points. First, if one triangle is fully on one side of the plane of the other, then there is no need to compute the intersection line, since the triangles can never overlap. Second, if two triangles are in collision, there is no need to check any other triangles in the configuration. The configuration is automatically invalid.

The organization of the tasks described below is as follows. First, the fused broad phase kernel transforms the robot, generates the AABBs for each robot, and runs the broad phase. This kernel parallelizes over configurations (i.e., each thread checks one configuration). Next, a separate narrow phase kernel checks each configuration not eliminated by the broad phase to determine which configurations are valid and which ones result in a collision. Since each pair of triangles can be tested independently, this kernel uses a warp to test each configuration; parallelizing over both configurations and triangles within a configuration.

An optimization we made to significantly improve the performance of our kernels was to coalesce the global data accesses. Specifically, in the original design, the inputs were stored in arrays-of-structs where each element in the array represented one vertex, a tuple of three coordinates, or triangle, a tuple of three references to vertices, in the mesh. Unfortunately, in this data layout, reads and writes are uncoalesced. Reorganizing the data into a struct-of-arrays, one for each dimension in the vertex or vertex in the triangle, allowed us to much more effectively use the available bandwidth.

We also made other optimizations such as moving the original robot and obstacle meshes into constant memory as it is used in the generation of every transformed robot mesh, and storing intermediates in shared memory instead of global memory.

4. Results

Through this project we tried a number of different implementations, each building on the previous one to improve performance.

4.1. Benchmarks

We generated two types of data sets, which we refer to as compact and sparse.

The sparse datasets were created by generating configurations where the translations were randomly selected from within a box 1000 times the size of the robot and the rotations generated purely randomly. This type of dataset was created to see how effective the broad phase was at clearing valid configurations, since most configurations generated this way would be valid,, as well as to see how the narrow phase performed when it had only a small number of configurations to check.

The compact datasets were created by generating configurations where the translations were randomly selected from within a box 1/8000000 the size of the robot and the rotations generated purely randomly. This type of dataset was created to see how effective the narrow phase kernel was when it was given a larger amount of work, as most configurations generated this way would be in collision.
We created the following datasets:

- Dataset 1 - 10,000 sparse configurations
- Dataset 2 - 10,000 compact configurations
- Dataset 3 - 100,000 sparse configurations
- Dataset 4 - 100,000 compact configurations

4.2. Experimental Set-Up

All experiments were run locally on a desktop PC running Ubuntu 22.04, with an Intel i7-8700K CPU and an NVIDIA GeForce GTX 1080 GPU. Benchmark values were achieved by running the same kernel 5 times and taking the minimum value. Times given include preparation of any data structures and copying to and from the device to host. The times do not include reading the configuration file from disk.

Because profiling is so time-intensive, we elected to use datasets 1 and 2 (which each include only 10,000 configurations) to develop our kernels. We elected to only use datasets 3 and 4 to collect final profiling and performance numbers. In this way, we were able to balance both the need for accurate profiling results and a fast testing cycle. The definitions of all profiling variables can be found at [4], though we define any statistics we use in the rest of this section.

4.3. Fused the Transformation, AABB Check and Broad Phase

The fused transformation, AABB check, and broad phase achieves relatively good utilization of the GPU. The multiprocessor activity describes “the percentage of time at least one warp is active on a specific multiprocessor”. On dataset 1, the average multiprocessor activity is 98.41%, while on dataset 2, the average multiprocessor activity is 98.63%.

Additionally, we see that all threads in the warp are being actively utilized. The warp execution efficiency describes the “ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor”. On dataset 1, the fused kernel achieves an average warp execution efficiency of 99.82%, and on dataset 2, it achieves an average warp execution efficiency of 99.83%. We, therefore, decided to focus on optimizing the narrow phase.

The naive narrow phase kernel implementation has a number of issues. Most importantly, this implementation relied entirely on global memory and did not do so efficiently. The global load transactions per request describes the “average number of global memory load transactions performed for each global memory load”. Similarly, the global store transactions per request describes the “average number of global memory store transactions performed for each global memory load”. On dataset 1, the naive implementation requires an average of 4.000 global memory load transactions per request and 875.4 global memory store transactions per request. On dataset 2, the naive implementation requires an average of 15.59 global memory load transactions per request and 2931 global memory store transactions per request.

4.4. Coalescing Memory Accesses via Shared Memory During the Narrow Phase

To optimize the narrow phase, we introduce two concurrent optimizations. First, we restructure our input data to be laid out as a struct-of-arrays. Additionally, to force all intermediate data to also be laid out in this manner, move all intermediates to shared memory and lay them out as a struct-of-arrays.

While this does not help us improve the global memory transactions per request, we reduce our reliance on global memory, and achieve very high shared memory transactions per request. On dataset 1, we achieve an average of 1.000 shared memory loads per request and 0.9995 shared memory stores per request. Similarly, dataset 2, we achieve an average of 1.000 shared memory loads per request and 0.9924 shared memory stores per request.

Unfortunately, the narrow phase still suffers from a major problem: control divergence. While the large number of early-exit points in the algorithm have the potential to dramatically improve performance, the kernel is not able to take advantage of this benefit. We see this in the profiling results. Unfortunately, dataset 1 has an average warp execution efficiency of 3.45%, while dataset 2 has an average warp efficiency of 36.96%.

4.5. Reducing Control Divergence via Coarsening During the Narrow Phase

We resolve this by devoting a warp to each configuration. While this does not eliminate control divergence caused by one triangle being fully on one side of the other triangle's plane, this does eliminate control divergence due to a collision leading to an early exit. The profiling results demonstrate that coarsening significantly improves our warp execution efficiency. Our optimized version achieves 81.13% efficiency on dataset 1 and 74.97% on dataset 2.

However, coarsening at the warp granularity has a major down-side: even if the very first pair of triangles results in a collision, the coarsened kernel performs 32 triangle-triangle collisions before it executes the early exit. Therefore, while this optimization provided a 2.167x and 2.294x speedup in the end-to-end execution time on datasets 1 and 3, respectively, it resulted in a 3.019x and 2.279x slowdowns for datasets 2 and 4, respectively. The coarsening allowed us to effectively early exit when a large number of configurations were removed by the broad phase (datasets 1 and 3), but added extra work when more configurations were in collision (datasets 2 and 4).

4.6. Remaining Issues With These Kernels

At first glance, the collision detection kernel seems very well suited to a massively parallel machine like a GPU: a large number of configurations need to be generated and processed independently. However, practical implementations of collision detection use a variety of early exit strategies, from the separate broad and narrow phases to early exit paths within the narrow phase. These early exit strategies lead to high variation in the time required to perform the check for different configurations, leading to extremely low utilization.

To test this, we profiled the best kernel implementation on datasets 2 and 4, which have 100,000 configurations each. In theory, this is more than enough to fill the GPU. However, because of the extremely long tail, the achieved occupancy of the narrow phase kernel, or the “ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor”, was only 2.997% on dataset 2 and 24.30% on dataset 4.

4.7. Fully Fused Kernel

In an attempt to further minimize thread divergence, as well as decrease global reads, a fully fused kernel — deemed the Mega Kernel — was designed. The Mega Kernel has warps operating on a single configuration collaboratively, similar to the previous approach, but does broad phase and narrow phase in a single kernel, eliminating an entire kernel’s worth of reads from global memory. The Mega Kernel also introduced two new techniques in an attempt to improve performance.

The first new technique introduced was the addition of two more steps to the broad phase instead of just checking the AABBs of the robot and obstacle. The first step checks the AABBs as before; if those do not overlap, then the entire warp continues to the next configuration. If they do overlap, we advance to step two.

In step two, an AABB for each individual triangle of the robot is created, henceforth referred to as a mini AABB. The robot mini AABBs are checked against the full sized AABB of the obstacle – any mini AABBs that are in conflict are cached for the next step.

In step three, the mini AABBs of the robot are checked against the mini AABBs of the obstacle, which have been precomputed on the CPU. Any overlapping mini AABB pairs are marked for narrow phase collision detection, bringing us to the second technique: caching. The potentially conflicting triangle pairs are stored into shared memory to be checked later.

By identifying which individual triangle pairs may be in conflict, we are able to enormously save on the number of narrow phase collision checks that need to be done. Since we found that narrow phase was by far the most time consuming part of our kernel, it only made sense to minimize this in whatever way possible.

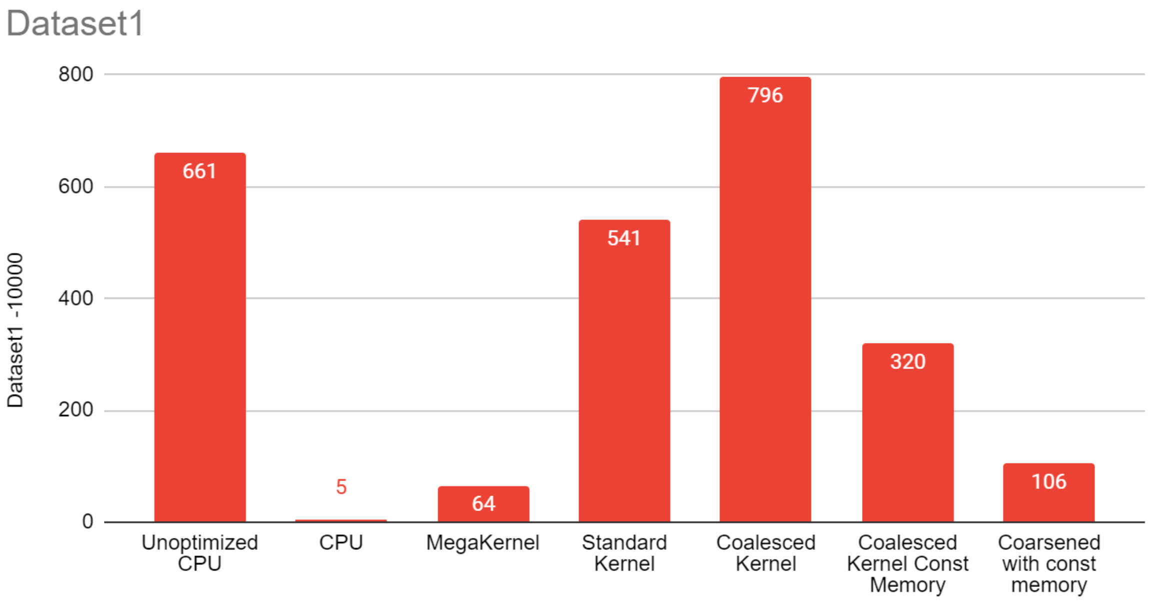
To reduce warp divergence, we do not perform narrow phase detection on the identified triangle pairs until we have either identified enough that each thread in the warp can perform narrow phase, or until we have found all potentially conflicting pairs for an entire configuration.

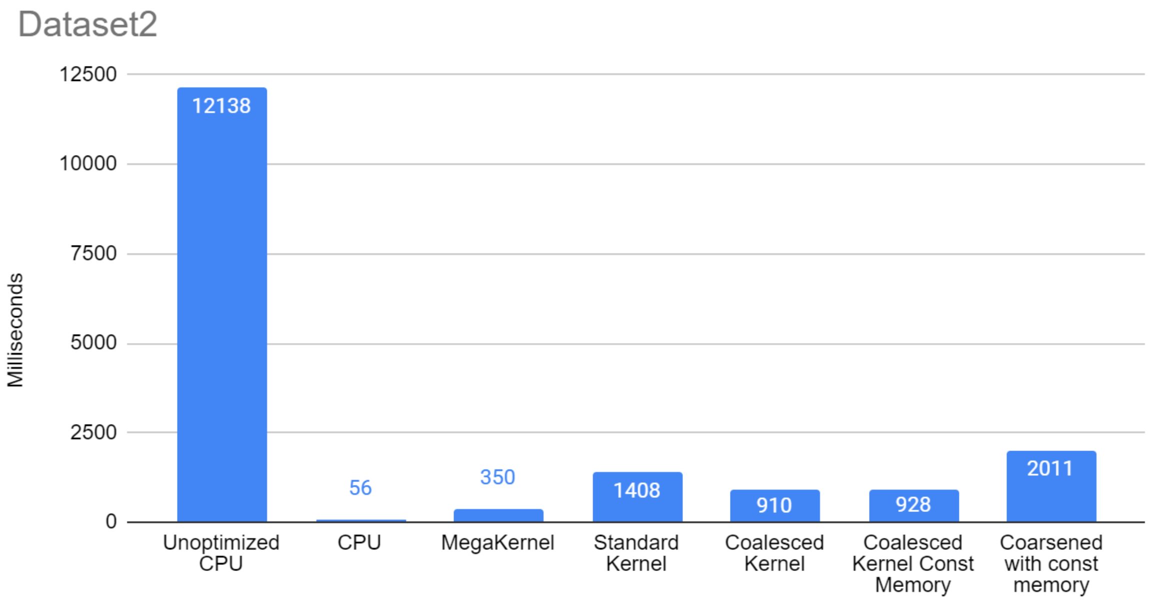
This was inarguably our best performing kernel, with speedups between 1.656x and 3.831x compared to the next best GPU kernel depending on the dataset. The three main reasons for this are:

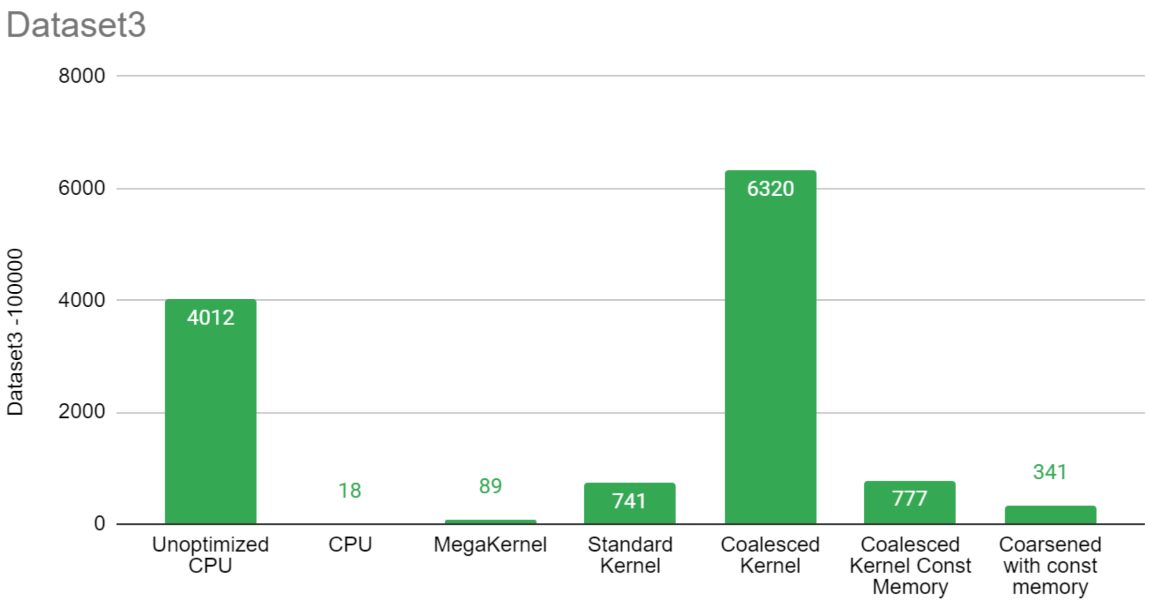
1. Fewer triangle-triangle intersections. By identifying potentially violating triangles in advance, we were able to reduce the number of triangle-triangle intersections to a fraction of a percent of the original number.
2. Fewer global reads and writes. By only launching a single kernel, we eliminate the need to communicate the results from one kernel to another, saving on overhead.

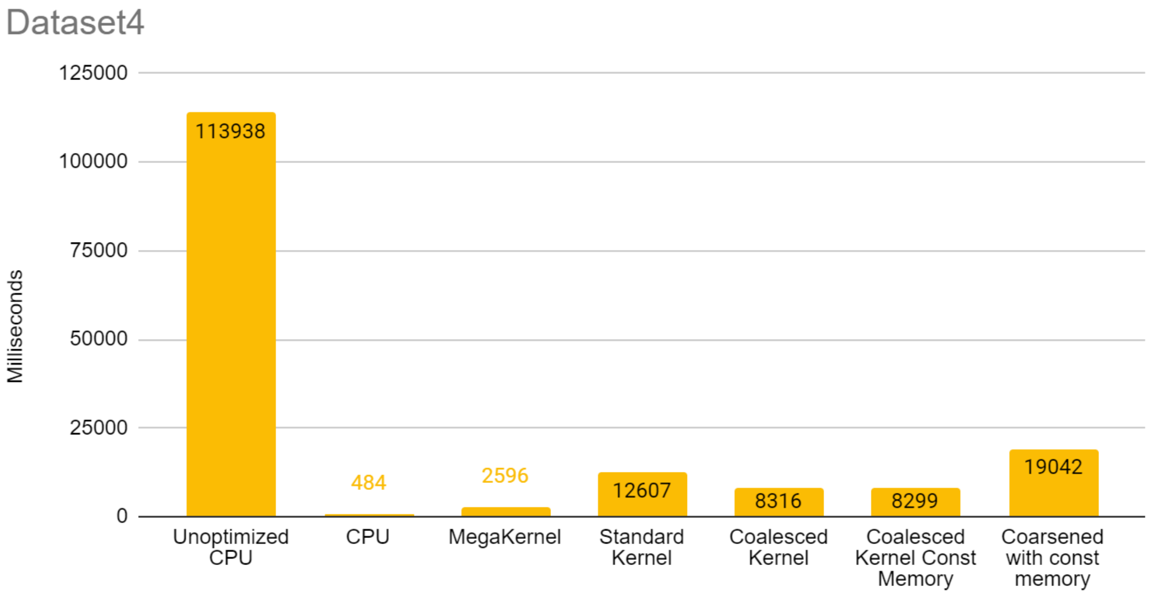
3. Less warp divergence. By pre-identifying triangle pairs with a higher probability of collision, we decrease the divergence number of threads that exit early when performing the narrow phase collision detection, thus lowering idle time.

4.8. Comparison to CPU Baseline









For a variety of reasons, we were not able to make our implementation competitive with the CPU implementation. First, we underestimated the speed of the CPU implementation. Initially, we accidentally ran the CPU version without compiler optimization. As we see in the table below, the compiler optimization improved the baseline by multiple orders of magnitude.

Second, after this optimization, preparing the data for use by the GPU (a constant overhead for all datasets of 45-50ms), required more time than the entire CPU implementation on datasets 1 and 3 and a comparable amount of time for dataset 2. In the end, no matter how good our GPU implementation ended up being, even if we were to overlap all copying with compute, this made it impossible for us to provide speedup over the GPU implementation.

5. Conclusion

Collision detection for robot motion planning seems like an ideal problem to parallelize and obtain significant speedup using GPUs because of several factors such as the independent computations, data reuse and the volume of data. But contrary to our assumption, we were not able to beat the execution times of the optimized serial implementation that solves this problem with our parallel implementation.

Some of the optimizations and techniques we used in our parallel kernels were as follows:

- Data with high reuse potential were moved to constant and shared memory. Some data was moved to registers as well, such as in the broad phase kernel.
- Coalescing of data accesses.
- Kernel fusion to ensure better utilization of the compute capabilities for the broad phase kernel.
- Coarsening at warp level during the narrow phase

Unfortunately, the following factors were observed to limit the speedup we could obtain with our parallel approach:

- Data movement time – the movement of data from the CPU to the GPU was more that what was required for the entire CPU kernel on multiple of our datasets
- Control divergence – the variety of early exit opportunities introduce a lot of control divergence and bring down the achieved occupancy significantly

To conclude, although it seems like collision detection for robot motion planning is an ideal candidate for parallel computing using GPUs, it is not so straightforward. We observed that our implementation was no competition to the optimized serial solution.

6. References

Link to our GitHub repository: https://github.com/vmurta/robo-check

[1] J. Reif and M. Sharir, "Motion planning in the presence of moving obstacles," 26th Annual Symposium on Foundations of Computer Science (sfcs 1985), Portland, OR, USA, 1985, pp. 144-154, doi: 10.1109/SFCS.1985.36.
[2] J. Canny, "A Computational Approach to Edge Detection," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. PAMI-8, no. 6, pp. 679-698, Nov. 1986, doi: 10.1109/TPAMI.1986.4767851.
[3] L. E. Kavraki, P. Svestka, J. . -C. Latombe and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," in IEEE Transactions on Robotics and Automation, vol. 12, no. 4, pp. 566-580, Aug. 1996, doi: 10.1109/70.508439.
[4] https://docs.nvidia.com/cuda/profiler-users-guide/index.html