GPU based Voxel-Collision-Detection for Robot Motion Planning

Andreas Hermann, Florian Drews
Research Center for Information Technology (FZI), Karlsruhe, Germany

IAS-13, Padova, 2014
General Agenda

- Introduction of the speakers, the FZI and the audience 10 min
- Motivation
- Short intro to CUDA programming paradigms 15 min
- Collision Detection 75 min
  - Data Structures
  - Collision Checks
  - Swept Volumes
  - Planning
- Visualizer 5 min
- Coffee Break 15 min
- Hands on 90 min
  - High Level API
  - Populating the Environment Map
  - Populating the Robot Map
    - Generate a Swept Volume
  - Collision queries
  - Generate own robot models
- Discussion / Outlook 30 min
Why are we doing all this?

MOTIVATION AND RELATED WORK
Motivation for GPU based collision checking

- Many fast planners work in 2D / 2.5D
  - But our world is 3D
- Reactive behaviors may lead to deadlocks.
  - We want fast planning, not behavior based decisions.
- Performance of Motion planning is heavily dependent on computationally expensive collision checking.
  - Speed up collision checks to speed up planning.
- Planned Motions may have already lost validity at execution time.
  - Trajectories have to be validated against dynamic environments continuously.
  - Highly parallel GPU based collision detection fulfills all these requirements
Related Work: Collision Detection for Motion Planning

- 2D vs. 3D Approaches
  - Downprojection, Layered Projection [1], Full 3D, Approximation with simple geometries

- CPU vs. GPU based collision checking
  - Most current GPU Software is Mesh based [2]
  - Is a geometric approximation required?
  - Is Single- or Multi-Querying needed?

- Mesh-based vs. Voxel-based approaches [3]
  - Which are better suited for Pointcloud data?
  - How efficient is Swept Volume generation?
  - Does the input fit into Bounding hierarchies?
  - Are Distance queries / Contact point calculation possible?

Mesh based vs. Voxel based collision detection

Mesh based approaches
- Contact points and forces can be calculated
- Intersection with point cloud data not directly possible
- Creation of Swept Volumes is computationally intensive but memory efficient

Discretization based approaches
- Inherent data fusion of various sensors within the data model
- Discretization level can be freely chosen, also reduces amount of data
- Generation of Swept Volumes is only a question of available memory
- Free / Unknown / Occupied space can be explicitly modeled
Contributors

- Development in progress since August 2012
- Contributors (in temporal order):
  - **Andreas Hermann** (Work on opportunistic mobile manipulation planning, managing GPU-Voxels as a sub-project)
  - **Sebastian Klemm** (Diploma Thesis on parallel collision detection with focus on safety)
  - **Jörg Bauer** (Diploma Thesis about planners that use GPU collision detection)
  - **Florian Drews** (Masters Thesis on the GPU Octree)
  - **Matthias Wagner** (Bachelors Thesis on the Shared Memory Visualization)
General goals

- Exploit advantages of the GPU performance compared to the CPU:
  - 7x higher computing performance, 5x memory bandwidth, 10x energy efficiency
- Develop solutions to parallelize data structures and algorithms
  - Either optimized for memory efficiency or calculation throughput
- Offer support for collision detection against live pointcloud sensor data and a-priori model data
- Achieve much shorter planning times in realistic robotic scenarios through hierarchical approaches
- Support interactive replanning in dynamic scenarios
- Realize short reaction times due to safety reasons
- Offer an intuitive tool for live visualization of model-, sensor- and planning data
System Overview

1. Voxel Collision Checks
2. Data Structures
3. Populating the maps
4. Swept Volumes
5. Planning
6. Visualization
Programming Paradigms for

CUDA-BASICS
Parallel programming on a GPU

What is CUDA?
- “Compute Unified Device Architecture”
- General purpose programming framework for Nvidia GPUs
- Interfaces exist for C, Perl, Python, Java, Fortran, .NET, MATLAB
- Scalable Scheduling and Memory Management abstraction layer through the runtime

Allows a high degree of parallelization
- Current GPUs allow the execution of 67 Million Threads

Streaming Multiprocessor principle with SIMD processing (Single Instruction Multiple Data)

One needs to be aware of some programming paradigms to generate efficient CUDA programs. But in general the same as with CPUs counts: When choosing a data structure, decisions have to be made between processing speed and memory usage
Example Specification of a current CUDA GPU

Datasheet of our current GPU:

- GeForce GTX TITAN by NVIDIA
- GK110-Chip with Kepler-Architecture
- 6 GB GDDR5-Memory, maximum Bandwidth 288.4 GB/s
- 16×PCI-E 3.0 Interface (theoretical Bandwidth of 6 GB/s between CPU and GPU
- 2688 GPU-Cores in 14 Multiprocessor Units (each of the 192 Cores executes 32 Threads simultaneously)
- **86016 Threads** can run in parallel
- 4500 Teraflops with single precision calculations
- Up to 250 Watt power consumption
Thread organization on the GPU

- Worker **Threads** of a Kernel are organized in **Grids (2D)** which consist of up to 65536 **Blocks (3D)**. Each Block manages up to 1024 Threads.
- Each **Kernel** call runs on one Grid.
- The CUDA/Runtime offers API functionality to address and identify the threads within the running kernel code.
- Optimal degree of parallelization can not be determined automatically.

---

from NVIDIA Corporation: NVIDIA CUDA Programming Guide v1.0, June 2007
Multidimensional Optimization of Parallelization Parameters

Required runtime in Milliseconds for a specific task (building an Octree) depending on the number of utilized Blocks and Threads

- Left: Small Dataset (building Octree from 3 Million points)
- Right: Large Dataset (building Octree from 13 Million points)

Evaluation of optimization required for each program function running in a Kernel theoretically. This is not feasible in practice

⇒ Utilizing the shown curve also for other Kernels by linearizing it around the global minimum
Memory Architecture

Nvidia GPUs have a multilayered memory architecture:

- Large Global Memory which is time intensive to access
- Block wise Shared Memory
- Thread wise Local Memory
- Fast local Registers
- The user has to take care of Memory Synchronization
- Program design has to be optimized for memory access patterns

![Memory Architecture Diagram](image)
Kernel Calls

Typical process flow for calling a Kernel function:

- **Host Computer**
  - Preprocessing of data
  - Post processing of results

- **GPU**
  - Copy data Host ➔ Device
  - Parallel processing in CUDA Kernels
  - Copy data Device ➔ Host

→ Memory transfer between Host and Device is a bottleneck
Conclusions: Paradigms for efficient CUDA Programs

- Kernels need to follow the same path through the code, because all cores have to execute the same code
  - If only one thread executes a time intensive task, all others of the same kernel have to wait
- Pipelined program architecture with kernels that works continuously on device data to prevent bottlenecks of PCI Bus data transaction
- Memory Allocation: Lots of small memory allocation operations slow down the execution.
- Memory Coalescing: Consecutive Threads have to access consecutive memory to operate efficiently:

Addresses:

| 96 | 128 | 160 | 192 | 224 | 256 | 288 |

Threads: 0 ... 31
What and why it is happening on the GPU?

BASIC PRINCIPLES OF VOXEL COLLISION DETECTION
Basic principle of Parallel Collision Detection

Collision checking is very well suited for parallelization

Basic principle of Parallel Collision Detection

Populating the Maps

Process chain runs with multiple Kinects at 20 Hz
Collision Checker as central component for Motion Planning and Trajectory Evaluation

- Models (Environment, Robot, Motion Primitives) can be represented by different datatypes, according to functional needs
- All components are optimized for high parallel throughput
Definition of System Reaction Times

- **Module chain:**
  \[ t_{\text{Reaction}} = t_{\text{Sensor}} + t_{\text{Environment Model Update}} + t_{\text{Collisioncheck}} + t_{\text{Robot Reaction}} \]

- **Worst Case:**
  \[ t_{\text{Reaction}} = 33.3 \text{ ms}^1 + 13.57 \text{ ms} + 15.59 \text{ ms} + t_{\text{Robot Reaction}} \]
  \[ t_{\text{Reaction}} = 62.46 \text{ ms} + t_{\text{Robot Reaction}} \]

1 Internal sensor lag not regarded
Parallelization of map population

Robot config

L * Matrix generation

L * M parallel Trafo & Insert

D * B parallel Trafo & Insert

L Matrices

D Matrices

L Links

M*L Points

Voxelmap

N Voxels Robot

N Voxels Environment

N Parallel Collision Checks & Log N Reductions

E * Bool

D * RGBD Sensor with B points

A priori map data

A priori map data
Taxonomy of GPU-DATA Structures

Trade-off between memory usage and computational throughput

Voxel-Map (red), Octree (green), Voxel-Lists (blue).

Suited Data-Types

Match data type to represented content:

- For large dense datasets we intersect two Voxelmaps (Environment + Swept Volumes)
  ➔ Guarantees a constant upper time bound

- For small subvolumes we intersect a Voxel-List (Robot) and a Voxelmap (Environment)
  ➔ Addressing scheme allows to efficiently look up occupancies
  ➔ Translation of the sub volume within the map by a simple offset addition:
  3D-Array like addressing scheme:

\[
I = B + \left( \left\lfloor \frac{z}{VS_z} \cdot MD_x \cdot MD_y \right\rfloor + \left\lfloor \frac{y}{VS_y} \cdot MD_x \right\rfloor + \left\lfloor \frac{x}{VS_x} \right\rfloor \right)
\]
(Multiresolution) Voxelmaps

- Represented as big array with 3D addressing scheme in GPU RAM
- Hierarchical collision checks can be achieved by instantiating multiple copies in various resolutions

+ Perfectly suited for parallelization
+ Almost constant runtime

→ Use a bounding box for collision checks, defined by low resolution collisions, to speed up calculations

\[
\text{Speedup} = \frac{T_{high} \times C_{expanded}}{T_{high} \times C_{High} + C_{expanded} \times T_{low}}
\]

with

- \( T_{high} \) = average Time for a high resolution collision check
- \( T_{low} \) = average Time for a low resolution collision check
- \( C_{expanded} \) = number of expanded grid cells
- \( C_{high} \) = number of necessary high resolution collision checks
Voxel Lists

- Minimal memory requirements
- Only relevant (occupied Voxels) are saved
- Same addressing scheme as Voxel-Maps

\[ B + \left( \left\lfloor \frac{z}{VS_z} \cdot MD_x \cdot MD_y \right\rfloor + \left\lfloor \frac{y}{VS_y} \cdot MD_x \right\rfloor + \left\lfloor \frac{x}{VS_x} \right\rfloor \right) \]

→ Easy translation relative to a Voxelmap by adding an memory offset
  - Highly efficient
  - Missing rotation has to be compensated by planner
Collision checking times between different data structures

<table>
<thead>
<tr>
<th>Cube side length [Voxels]</th>
<th>Mean num of collisions [Voxels]</th>
<th>Octree ∩ Octree [ms]</th>
<th>Octree ∩ Voxel-list [ms]</th>
<th>Voxelmap ∩ Voxelmap [ms]</th>
<th>Octree ∩ Voxelmap [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.01</td>
<td>1.46</td>
<td>0.10</td>
<td>15.31</td>
<td>4.21</td>
</tr>
<tr>
<td>8</td>
<td>0.44</td>
<td>1.46</td>
<td>0.10</td>
<td>15.31</td>
<td>4.20</td>
</tr>
<tr>
<td>32</td>
<td>37.60</td>
<td>1.44</td>
<td>0.11</td>
<td>15.32</td>
<td>4.28</td>
</tr>
<tr>
<td>128</td>
<td>1862.98</td>
<td>1.88</td>
<td>0.41</td>
<td>15.32</td>
<td>4.71</td>
</tr>
<tr>
<td>187</td>
<td>6781.29</td>
<td>1.91</td>
<td>1.04</td>
<td>15.34</td>
<td>5.61</td>
</tr>
</tbody>
</table>

Benchmark: Median time (100 runs) in ms for computing the collisions (∩) between an environment map and a randomly placed occupied cube with a given side length in voxels. The environment consists of 3D scans, that occupied 150.684 voxels in a map of 884 × 1004 × 187 voxels.
How we implemented a dynamic structure on the GPU

LOAD BALANCED GPU OCTREE
GPU Octree (I)

Problem:
Dynamic data structure contradicts parallel data processing in CUDA
- Dynamic memory management performs bad in CUDA
- Dynamic Thread-Spawning is possible with “Dynamic Parallelism” but low performance
- Parallelization for > 1000 threads

➔ Port of CPU code to GPU is not sufficient. Redesign is needed!

from NVIDIA Corporation: NVIDIA CUDA Programming Guide v6.0, Feb 2014
GPU Octree (II)

Our solution:
- Pre-sorting point-data when inserting new data (Binning)
  - Determining required memory and allocating it at once
- Load balanced decent into parts of the tree with various depth
  - Probabilistic sorting of Work-Queues results in homogenous workload per task
- Code optimization:
  - Excessive use of CUDA Intrinsics
  - Coalesced Memory access
- Calculation intensive free space determination via Raycasting is done with a lower resolution than the representation of the occupied voxels.
- Periodical rebuilding of the tree to free unused memory

[J. Pan, D. Manocha - "GPU-based parallel collision detection for real-time motion planning." Algorithmic Foundations of Robotics IX. 2011]
GPU Octree (III)

Probabilistic and deterministic implementations are available:

- Voxels have **1 Byte occupancy** in log-odd representation
  - Sufficient for most applications
- Deterministic occupancy: occupied, free, unknown

- Voxel addressing via 60-bit Morton code:
  - Represents the path from root to specific node
  - Addressable area: \(~10 \text{ km side length with } 1 \text{ cm resolution}\)

- Child nodes as array:
  - Only **one pointer** per node needed instead of 8
  - 40-bit pointers \(\rightarrow\) Maximum size of Octree: **1 TB**
Octree Collision Detection

- Hierarchical approach:
  - Simultaneous tree traversal and pair-wise node comparison
  - Possible collision detected if both nodes are occupied
  - Expand nodes for higher precision
  - Coarse resolution results in faster computation
Parallel Traversal with DFS

- Traversal: breadth-first search (BFS), depth-first search (DFS)
  - DFS: less memory and less synchronization needed

- Parallelize: Multiple DFS independent of each other
- Unknown structure of Octree ➔ Possible Load imbalance

- Adapt and extend load balancer of gProximity
- Approach: Balance work on demand (#idle threads > X)

Parallel Load Balancing (I)

- Redistribute stack items (Octree nodes) equally
- Nodes imply different amount of work
- Heuristic approach

\[ \text{level}(n) > \text{level}(m) \rightarrow \text{work}(n) > \text{work}(m) \]

→ Equal work distribution, needing less balance steps, resulting in a higher degree of parallelism
Parallel Load Balancing (II)

1. Each stack counts items per level
2. Parallel prefix sum starting with biggest work items
3. Parallel distribution by computed offset $i$

$stack(i) := i \mod \#stacks$

$pos(i) := \left\lfloor \frac{i}{\#stacks} \right\rfloor$
Parallel DFS with Load Balance

Adaptions of this approach are useful for other tasks like:

- Load-balanced collision detection with Voxel map
- Top-down propagation for big Voxel updates
- Restoring the Octree invariant after inserting new data
  - Parent node combines status of child nodes
- Extracting represented Voxel data of Octree
  - Needed for visualization via inter-process communication
Interactive Collision Detection and Visualization
Environment Observation with Kinect and PTU
Importance of Evaluation Direction

Collision detection between Octree and Voxelmap:

- **Left:** Scanning of Voxelmap and look-up in Octree
- **Right:** Octree traversal and look-up in Voxelmap
- Exploits Octree hierarchy in homogenous areas of same status

→ Performance gain for sparsely covered Octrees
Example of Sparse Data of a realistic Scene

Point cloud FZI40M with **40.4 million points**
- Dimensions: **101.5 m x 97.2 m x 24.7 m** in **2 cm-Resolution**
- Voxelmap memory requirements at least **28 GB**
- GPU-Octree representation **300 MB**
Building an Octree out of a point cloud (FZI40M) for different voxel sizes:

- Moving data to GPU and sorting it takes a major part
- Scales linear with input data size
- Comparable to state of the art GPU approaches
Memory Efficiency

Memory usage compared to OctoMap for point cloud FZI40M:
- Memory usage increases with the number of points due to less data locality

→ Memory savings of at least factor 3.4 compared to OctoMap
Runtime for Collision Checking

Runtime and scaling of collision checking between two Octrees
- Validates random robot trajectory in < 3 ms
- Less accuracy yields performance improvement
- Work for load balancing has an influence of up to 50%
Collision checking between environment (FZI40M) and random robot trajectory

- Combination of different data structures
- Runtime in Milliseconds depending on voxel size
  - Up to one order of magnitude speed-up due to Octree
Skalierung und Zusammensetzung der Laufzeit beim Einfügen von Sensordaten

- Laufzeit für das Einfügen neuer Sensordaten
  - inklusive Freiraumberechnung
  - Sensor-Reichweite: 5,25 m.
  - Ohne die Auflösung des Freiraums zu reduzieren, kann bei einer Voxelgröße von drei Zentimetern noch die volle Datenrate der Kinect verarbeitet werden
Insertion of Sensor Data

- Insertion of Kinect sensor data with a range of up to 5.25 m
- Includes free space computation which is most expensive
- Online processing of sensor data
- Speed-ups of up to two orders of magnitude compared to OctoMap
Omnidirectional sensor carrier

Test scenario:
- Manually driven
- SLAM running with laser range finders
- Cameras extrinsically “calibrated”
- Data captured into rosbags and
- Octree generation during data playback on GPU workstation

- Omnidirectional driving
- 6 ASUS Xtion RGBD cameras
- High Performance Embedded computer
- 2D-Laser-Rangefinder localization
Video of 3D-Map generation
Restrictions of the presented approach

- High-precision localization of the mobile platform is required to construct consistent 3D-maps
  - Synchronization between 3D-Sensor data and platform localization is needed to insert the data correctly into the global map
  - This requires exact knowledge and modeling of the systems timing parameters
  - Pose jumps due to Loop closing points in a SLAM algorithm destroy the map \(\Rightarrow\) SLAM can not be used but only localization against a a-priori known map
- Alternative concept:
  - Use an a-priori generated environment model for global planning
  - Insert Live-Data into another map for local collision detection and reactive planning. Sensor data fades out after a given time horizon
How do we generate larger queries to optimize GPU load? Check more than just a single robot configuration!

SWEPT VOLUMES
Swept Volumes

- Swept Volumes represent the spatial volume that is covered by a motion.
- Sweeps can consist of whole motion plans, parts of plans, or only motions of single robot links.
- Depending on the underlying data structure, they may consist of sub-volumes, to distinguish time or other criteria.

Generation of Sweeps is computationally intensive via 3D-Meshes, but comes at no computational cost with Voxel-Representations.
Virtual Corridor

- Generate a Swept Volume by keeping the results of path planning in the Voxel-Map to create a virtual corridor for the robot.
- Monitor the Swept-Volume during execution for penetrating obstacles
- Detect future collisions and calculate time to impact

1 current robot pose (viewn from top)
2 swept volume
3 obstacle
4 removed volume of completed motion
Interleaved Planning and Execution

- Virtual Robot precedes real Robot
- Leaves a virtual corridor
- Corridor is monitored for dynamic obstacles
Swept-Volume monitoring with online replanning of 10 DOF

Initial planning took 120 collision checks, 13 ms each
Monitoring of an already planned sweep is straightforward, but how do we generate a plan without checking single configurations to achieve **maximum parallelization**?

**OPTIMIZED PLANNING APPROACH**
Concept of motion primitive platform planning

The most time consuming step in the planning pipeline is the Voxelization of the robot model

- Try to keep the model constant and translate its Voxel-List within the map.

Problem: Translations are straight forward accomplished by offset addition, but rotations result in discretization errors.

- Use a rotated Swept-Volume model and move it around while distinguishing subvolumes of rotation sections
D*-Lite Planner

- Planning happens on equidistant 8-Grid
- Nodes in the grid represent feasible and infeasible rotation angles of the robot
- Adjoining regions become graph nodes and graph edges are added between neighbor grid-nodes when they share common collision free angles.
- During the graph search nodes with similar orientation are preferred to suppress superfluous platform rotations

D*-Lite Planner – Reuse of planned sections

- Free space assumption
- Reuse of already planned path sections
- Significant speed up, as fewer new nodes have to be expanded, compared to planning from scratch

The closer to the start point the new obstacle lies, the bigger is the saved planning time.

~ 100 ms saved by replanning

~ 600 ms saved by replanning
Planning in narrow passages
Use Cases

- Evaluating Optimal Manipulation Poses
- Evaluating Goal-Space-Regions
- Opportunistic Planning in different abstraction levels
- Whole Body Motions are evaluated with „traditional“ OMPL - Planner
Video Demonstration of instant replanning
Video Demonstration of instant replanning
Narrow Passage (Scenario 1)

- Box on the table blocks the robot arm. Robot has to turn before driving through the narrow passage.
- A path has been found for cell sizes 0.04m and 0.08m.
- With 0.04m the number of cells is four times higher.
  - More collision checks required
  - Higher number of graph nodes causes more time for graph creation and search
- g-value of the goal node consists of the Euclidean distances between the grid points and the cost values for rotation and translation.
  - Neither the voxelsize nor the cell size is affecting the path length drastically.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.04</td>
<td>0.01</td>
<td>2651</td>
<td>2.472</td>
<td>3.621</td>
<td>5.73</td>
</tr>
<tr>
<td>0.04</td>
<td>0.02</td>
<td>2634</td>
<td>1.114</td>
<td>2.241</td>
<td>5.77</td>
</tr>
<tr>
<td>0.04</td>
<td>0.04</td>
<td>2670</td>
<td>0.546</td>
<td>1.692</td>
<td>5.95</td>
</tr>
<tr>
<td>0.08</td>
<td>0.01</td>
<td>745</td>
<td>0.608</td>
<td>0.787</td>
<td>6.17</td>
</tr>
<tr>
<td>0.08</td>
<td>0.02</td>
<td>720</td>
<td>0.292</td>
<td>0.471</td>
<td>5.86</td>
</tr>
<tr>
<td>0.08</td>
<td>0.04</td>
<td>757</td>
<td>0.139</td>
<td>0.322</td>
<td>5.99</td>
</tr>
</tbody>
</table>
Due to the fast resolution complete sampling, our planner finds smooth paths without additional smoothing overhead.
Planning in Point-Cloud data (Scenario 3)

- Environment data taken with a rotating Laser range finder
- Narrow passage was only found with small grid and voxel size

Proof of concept with realistic data
Comparison to OMPLs RRTConnect

- In the first, harder, scenario our planner is faster than in the second.
- Whereas the RRT planner needs most time for the first scenario.

Planning time mainly scales with path length, but not with difficulty of the problem.

Planning time is an extrapolation of #CollisionChecks times the average collision checking time (0.240 s for 1000 mesh based collision checks in random environment).

[ICRA 2012: J. Pan, S. Chitta, and D. Manocha, “FCL: A general purpose library for collision and proximity queries.”]
RRT planning of an 15 Degree of Freedom model

- In average 7169 Collisionchecks are needed (Min: 103, Max: 38240)
- Required time (avg.): 7,719 sec. (Min: 0,127 sec., Max: 40,997 sec.)
- Average time to plan: 7,795 sec. (Min: 0:128 sec., Max: 41:554 sec.)
- Path smoothing took another 2,1 sec. In average

→ Single Collisionchecks between Robot Model (37862 voxelized points) and environment voxels (3,2 Million) took only 1 Millisecond
More planning results within simulated environment

Unified GPU Voxel Collision Detection for Mobile Manipulation Planning

A. Hermann, F. Drews, J. Bauer, S. Klemm, A. Roennau, R. Dillmann

Supplementary Video for IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2014)
How to render the Collision Checking data to the screen without Host interaction

SHARED MEMORY VISUALIZATION
Shared Memory Visualization on GPU

Previous approach with ROS rviz:
- Do collision checks on GPU with CUDA
- Copy results to Host-RAM
- Visualize it via OpenGL on the GPU
  ➔ Bad performance with larger scenarios

New approach:
- Shared Memory connects CUDA and OpenGL
- OpenGL Shaders are used to visualize all data types
- Rendering of several Million Voxels with High Performance
- Downside: No CAD surface models yet
  ➔ Live-Visualization of sensor point clouds and planning processes
Shared Memory Visualizer Features

- Change rendered Voxel-resolution on the fly
- Toggle single maps visibility on the fly
- Visualize collisions
- Draw arrays of markers to e.g. visualize planned paths
- On-Click information about Voxels is given (position, occupancy, map)
- Full parameterization of colors, gradients, camera poses via XML file
Enough talking, lets see the code…

HANDS ON!
The environment

- The software is an **interchangeable library (ICL)** running within FZIs **ic_workspace** → *ic* means, it can run in whatever environment, e.g. as a ROS-node, as a MCA-module or as a standalone program.
- We build it with FZIs **ic_maker**, which is a comfort layer on top of cmake, comparable to ROS catkin
- **gpu_voxels** depend on the basic library **icl_core** (OS independent comfort layer that implements logging, config, …)
- There exist a lot more ICLs, making autonomous cars drive, Marsrovers explore, walking machines walk…

Licenses:
- The ic_workspace and the ic_maker with its FindScripts ships under a **BSD license**.
- **icl_core** and the **gpu_voxels** come with a **CDDL license**
- The example Maps (*all.pcd, holl.pcd*) are kindly provided by **Jan Oberländer (FZI)** and may **NOT be redistributed**!
Directory structure

ic_workspace
  └── build
  └── doc
  └── export
  └── icmaker
    └── CMakeModules
  └── ide
    └── ic
      └── vc8
  └── packages
    └── gnu_voxels
      └── models
      └── src
        └── examples
        └── gnu_voxels
          └── kernels
          └── logging
          └── shaders
        └── icl_core
        └── python
          └── script

Here are the binaries
This is the standard export dir if you want to link against gnu_voxels
Here lie the Cmake Find Scripts
All software packages go here
Our GPU-Voxels library
Example model files
Example code
Shared Mem Visualizer
Main Library
General Helper Functions
Logging Defines
The GPU Octree
Helpers to represent kinematic structures
Unit tests
Definition of the Visualization Interface
The GPU Voxelmaps
Basis Library
XML parser for config files
Demonstrated components

- Creation of maps
  - Octree
  - Voxelmap
- Insertion of Data
  - Live Kinect data
  - A-priori map data
- Creation of a robot model
  - Voxelization of STLs with Binvox
  - Defining model parameters
- Animate the robot
  - Interpolate a motion
- Generate a Swept-Volume
- Collide the different models with each other
- Demonstrate Free-Space calculations
Programs

- In the “examples” directory you will find some code to test the functionality:
  - robot_vs_environment – High Level API
  - sweptvolume_vs_environment – Collision Detection against a Sweep
  - octree_provider – Visualize the free space calculations in live data
    (Requires ROS and Octomap to be installed, sorry)
- gpu_voxels_visualizer:
  - Shared memory visualizer. Press “h” while your mouse is inside the window, to get a list of Shortkeys
- All programs are in the build/bin folder, and need the environment variable `GPU_VOXELS_MODEL_PATH` be set to the folder, that contains the example files (`gpu_voxels/models`)
Open ends that need to be tied…

- Linker-Errors when using a specific instantiation of the Octree Template parameters, that is not yet a DEFINE
  ➞ If templatization would reach highest level API, user would need to compiled his software via NVCC and not via GCC
  ➞ Compilation times are a lot more convenient this way
- N-Tree is not yet a N-Tree
- API Functionality: Not all Low Level calls are yet available in high level interface. Some high-level calls are not yet implemented on lower levels
- Long start at first run after compilation is needed for CUDAs Just in Time compilation
- Visualization currently transfers too much information (whole map, with all types)
- Careful with points that lie outside of the represented volume
- Doxygen Docu not complete yet
So, what did we do? And what’s next?

CONCLUSION AND OUTLOOK
Conclusions

- We realized a software library that takes advantage of the high parallel calculation abilities of CUDA GP-GPUs.
- Datatypes and algorithms were developed that support this high degree of parallel access.
  - GPU-Voxels offers 3 different datatypes for various use cases.
  - Deterministic and probabilistic implementations are available to represent sensor data as well as a-priori data.
- Special planners that want to take advantage of the massively parallel Voxel based collision detection have to be designed to query collision checks not in a serial manner but in parallel.
- Swept-Volumes are a geometrical representation of whole plans, sub-plans or single motions that can be handles highly efficient by our library. Sections of the volume can be distinguished via a bitvector.
Practical application

- Research is partly funded by the BMBF Project ISABEL
- Partners are infineon, KUKA, celisca, macio, Fraunhofer IFF and FZI
- The goal is the automated material transport and machine-loading in clean rooms of chip production facilities
- Production process is undergoing constant changes
  - Utilization of standard automation processes with static machines for part-handling is not practical
  - Mobile Manipulation is the preferred way to go
  - Environment is partly unstructured and shared with human co-workers
- Dynamic reactions of the robot (Mobile Platform and Manipulator-Arms) are needed.
Outlook on current and future research

- Implement non-grid-based Planner for 2D Problems in 3D world
- Implement high DOF Motion-Primitive Planner for Whole Body Motions
- Realize Distance Queries in Octree
- Maintain [www.gpu-voxels.org](http://www.gpu-voxels.org)
- Implement full featured demo on our mobile manipulator
Low-Power Embedded CUDA Platform

Nvidia Jetson Boards are available on the market:
- Tegra K1 Chip combines
  - Low-Power ARM-CPU-Architecture
  - CUDA GPU
- Low price
- Targets automotive and automation sector
- Comparable platforms with more power also exist

Three boards are available at the FZI
- Performance needs to be evaluated
- Probably utilize it as preprocessor for 3D Cameras in a distributed Sensor-Network
Publications & Copyrights

Our Publication in this field of work:


Image Copyrights: If not stated otherwise, all images in these slides were created at the FZI and are published in the cited works.
Thank you.

More info available at: www.gpu-voxels.org

We would be happy if you can use it for your projects and support our future library development with ideas and implementations!

More Videos at: www.youtube.com/FZIchannel

FZI Living Lab Service Robotics
Dipl.-Inf. Andreas Hermann
Tel.: +49 721 9654-242 | hermann@fzi.de

FZI Forschungszentrum Informatik
Haid-und-Neu-Str. 10-14
76131 Karlsruhe
Germany
www.fzi.de

The research leading to these results has received funding from the German Federal Ministry of Education and Research (BMBF) under grant agreement no. 01IM12006C (ISABEL).