Obstacle-Avoidance Control-Candidate Evaluation Using a GPU

Peter Nordin

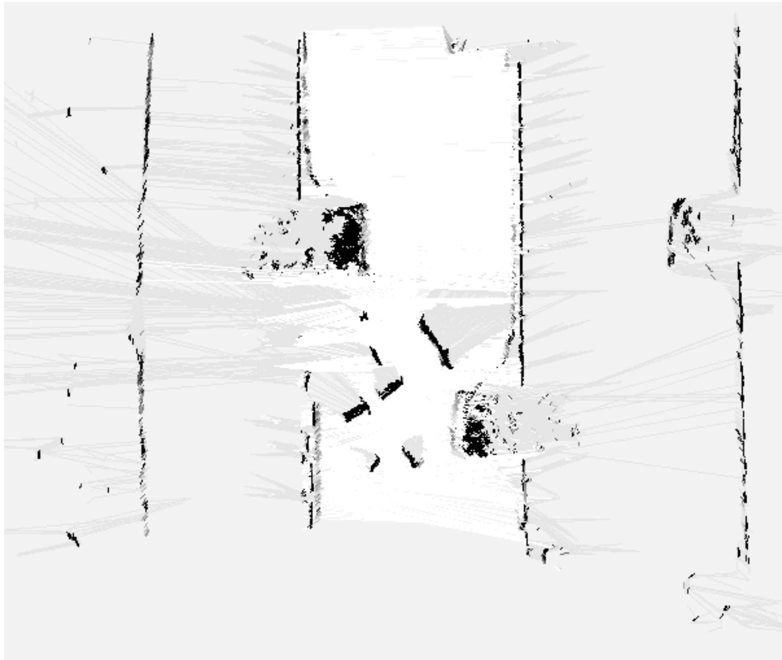
Fluid and Mechatronic Systems
Department of Management and Engineering
Linköping University

May 28, 2010

Introduction and Background

The preRunners project

One small preRunner maps ground traversability and sends this data back to a larger less agile master vehicle. The map can be used for obstacle avoidance.



Introduction and Background

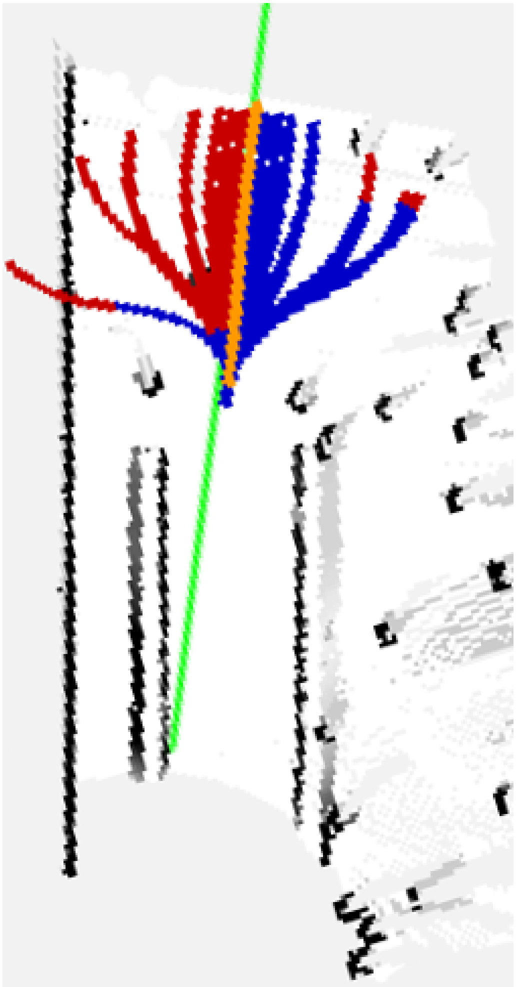
Obstacle Avoidance

- Obstacle-avoidance using traversability maps
- Short sensor range leads to sudden obstacles, no time for advanced path planning
- Obstacle-avoidance by detour from the desired path
- Primary detour candidates are lines parallel to the desired path

Simulation and Evaluation

Simulation

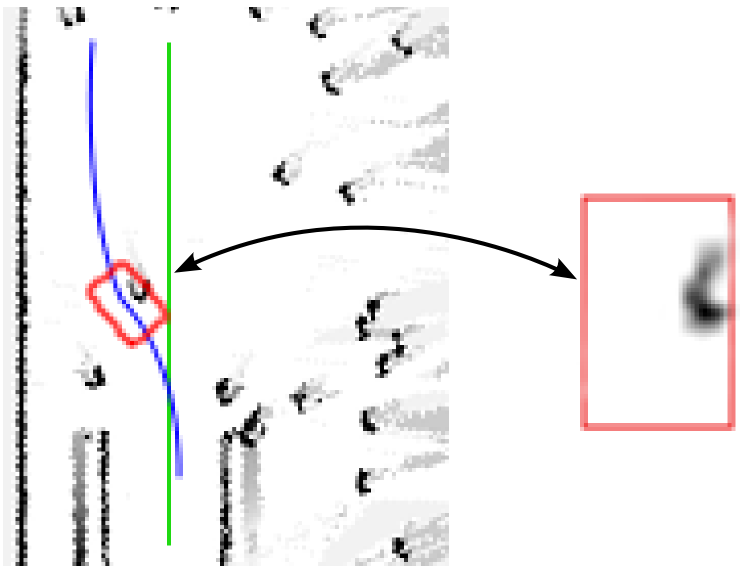
- A time discrete motion model including limitations on the steer wheel rate of change is used to simulate the robot
- Movement along the lines is controlled by a line following algorithm
- Candidates have different offset and may be given different speeds and controller parameters



Simulation and Evaluation

Evaluation

- Safety evaluation by sampling robot safety zone around poses along the simulated trajectories



- The best candidate is decided by a score function
- Score is based on how far we got without a collision, how close we were to collision and how close to the desired path we end up

Simulation and Evaluation

Limitations of the CPU Implementation

The primary limitation of the CPU implementation is the slow memory access for pose evaluation

- So slow that we can not wait for all candidates to be evaluated
- Only one new candidates are evaluated each time. It is unlikely that we will find the optimum solution
- Candidate selection is limited to different offsets, only one speed is tested
- All candidate evaluations, currently 48, (24 unique), takes between 1.4 and 3.4 seconds, (Core2 Duo E6750)

Simulation and Evaluation

Why GPU

- The OA-algorithm requires a lot of CPU-time that could be used for other things (like map management).
- The slow evaluation speed could lead to a collision if the robot speed is to high.

The algorithm has relatively little computations but much memory access. Not really suitable for GPU
However:

- The GPU likely has faster memory access
- We can simulate MANY different candidates at the same time, including different speeds and controller parameters
- If the GPU does all the heavy work, the CPU can do other things

Simulation and Evaluation

My Task

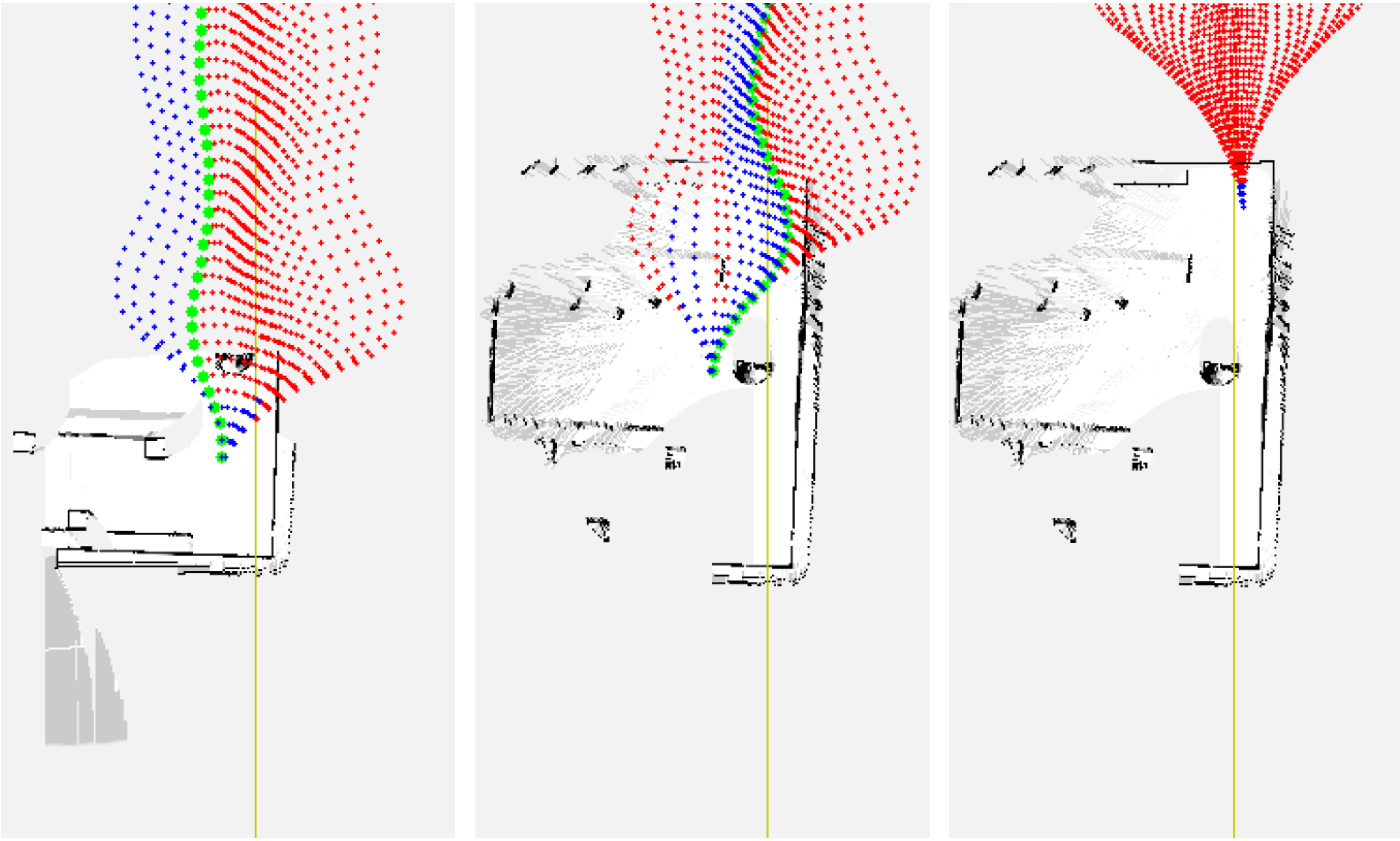
Determine:

- If this can be done on a GPU
- If it is any faster
- How to run CUDA kernel code together with other C++ code, (OpenCV and my own program)

Using CUDA Toolkit 3.0 which have better C++ support (among other things)

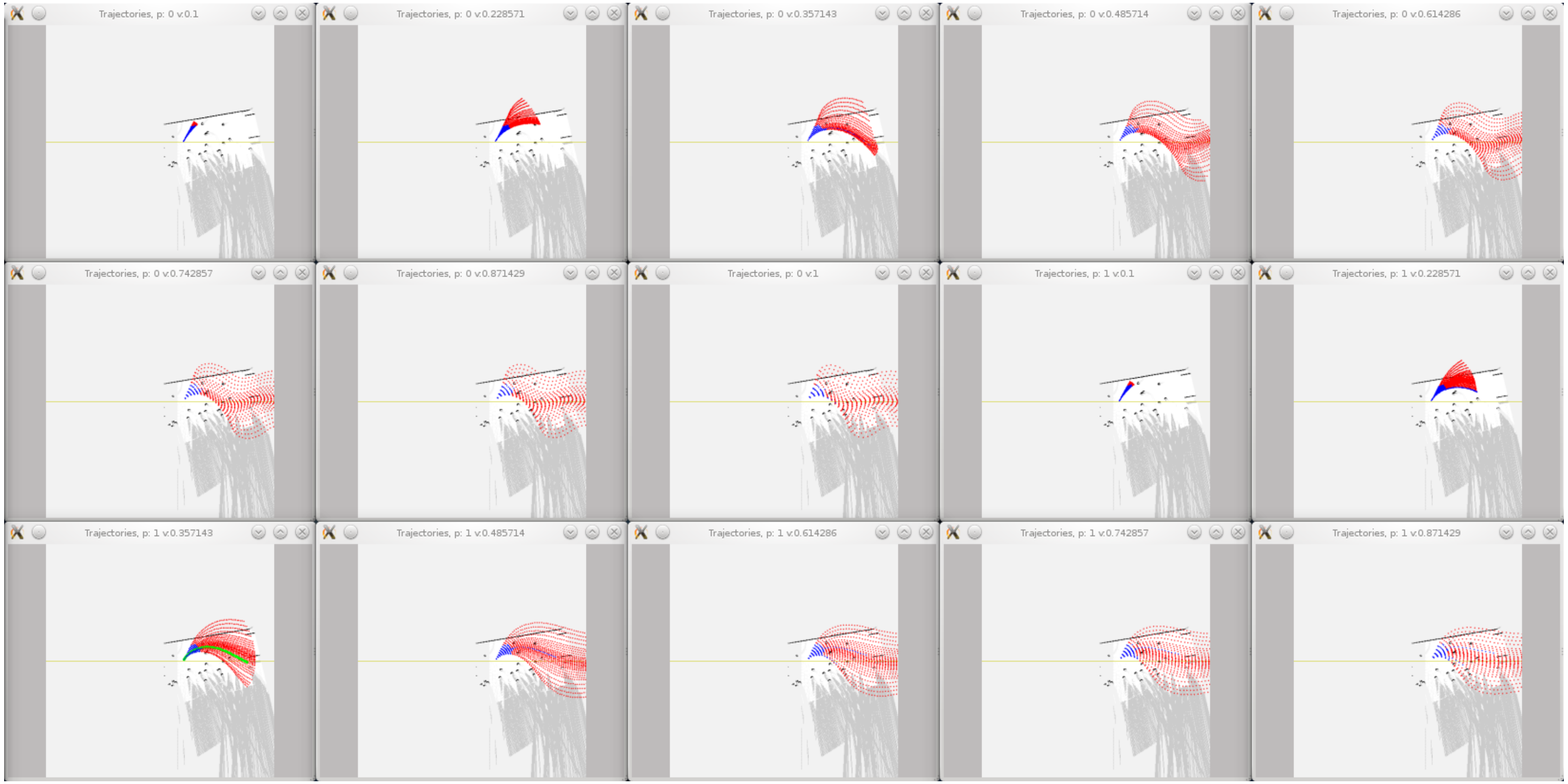
Sneak Peak

Detour when Avoiding Obstacle



Sneak Peak

Multiple Combinations of Candidate-Offsets, Speeds and Controller Parameters



The GPU Implementation

CPU to GPU Changes

- Simulation related code is basically copied and pasted.
- For pose evaluation in the map, the GPU texture-memory is really suitable. It offers Uncomplicated (x,y) pixel based memory access and local 2D cache
- All candidates are simulated the same number of steps, fast candidates always go further then slow ones. As a result the scoring function gets more complicated

The GPU Implementation

Functions and Classes

It is convenient to be able to write functions and classes in CUDA just like in C++

Listing 1: Function

```
__device__
float myNormrad(float ang)
{
    return atan2f(sinf(ang), cosf(ang));
}
```

Listing 2: Class

```
class MyCudaPose2D
{
public:
    float x,y,yaw;
    __device__
    MyCudaPose2D(): x(0.0), y(0.0), yaw(0.0) {};
};
```

The GPU Implementation

Compiling and Linking

CUDA kernels can not be called directly from normal C++ code.
CUDA kernels are compiled with nvcc together with a C++ wrapper.

Listing 3: My Makefile

```
CFLAGS=`pkg-config --cflags opencv` -I/usr/local/cuda/include -g
LDFLAGS=-L/usr/local/cuda/lib64/ -lcudart `pkg-config --libs opencv`
CUDACFLAGS=-g -G --compiler-bindir=/usr/bin/g++-4.3 -lcudart
DEFINES= -DSEKVENTIAL

all : cudaOAmain

cudaOA : cudaOA.cu
        nvcc $(CUDACFLAGS) $(DEFINES) -c cudaOA.cu -o cudaOA.o

cudaOAmain : cudaOAmain.cc cudaOA
        g++ $(CFLAGS) $(DEFINES) -c cudaOAmain.cc -o cudaOAmain.o
        g++ $(LDFLAGS) cudaOAmain.o cudaOA.o -o cudaOAmain

clean :
        rm cudaOAmain
        rm *.o
```

The GPU Implementation

Initialization

- Convert OpenCV image to "raw byte array"
- Allocate Host and Device memory
- Calculate the necessary amount of threads
- Transfer:
  - Candidate offsets
  - Controller parameter sets
  - Speed alternatives
  - Current robot states
  - Simulation results
- From score, select best candidate

The GPU Implementation

CUDA Code

All candidate combinations are run in parallel

Sequential texture memory access

1. Determine candidate combination
2. Simulate movement along each candidate
   2.1 Run control algorithm
   2.2 Update pose
   2.3 Determine if collision
   2.4 Store pose
   2.5 Exit loop if collision
3. Calculate safety score

Using one kernel

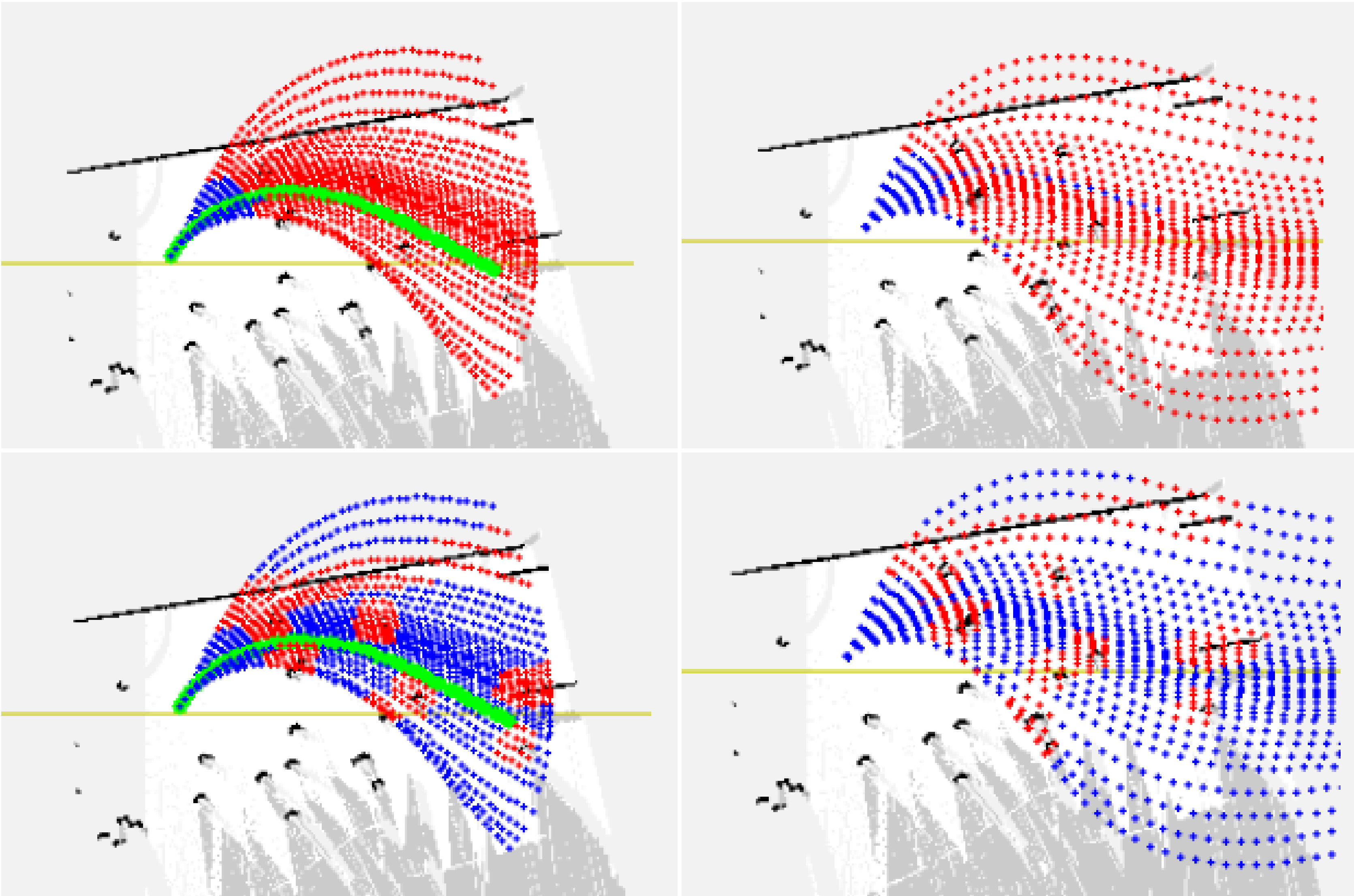
Parallel texture memory access

1. Determine candidate combination
2. Simulate movement along each candidate
   2.1 Run control algorithm
   2.2 Update pose
   2.3 Store pose
3. Check ALL stored poses for collision in parallel
4. Calculate safety score

Using three kernels

Results

Sequential vs Parallel



Results

Performance Comparison

Unique candidate offsets:	32
Speed alternatives:	8
Controler parameter sets:	2
Simulation steps:	500
Colission check skipfactor:	10
Timestep:	0.1
Total candidate combinations:	$32 * 8 * 2 = 512$

	Quadro FX 1700	GeForce GTX 260
Sequential	0.58, (0.53)	0.43, (0.38)
Parallel	0.53, (0.48)	0.16, (0.10)
Sequential Short	0.28, (0.23)	0.22, (0.16)
Parallel Short	0.53, (0.49)	0.16, (0.10)

CPU: 48 simulations in 1.4 seconds

Conclusion

- With the Quadro FX 1700 (comute capability 1.1) both methods have aproximately the same performance
- With the GeForce GTX 260, (compute capability 1.3) the parallel texture access is faster then the sequential even though all poses are evaluated in both cases
- If ALL candidates lead to an early collision, the sekvential one may be faster on the slower card

Possible Improvements

- The simulation code and some functions contain short branches, e.g. to avoid division with zero, Finding alternative mathematical models could eliminate branches.
- Currently not using shared memory, potentialy great performance improvements. However it is difficult to figure out how to efficiently program this. Texture access is unevenly distributed.