Normal Calculation in Point Clouds with CUDA

Alexander Mock, Matthias Greshake

amock@uos.de, mgreshake@uos.de

Institut für Informatik
AG Wissensbasierte Systeme

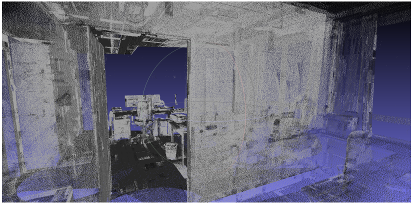
26. January 2016

UNIVERSITÄT OSNABRÜCK

Outline

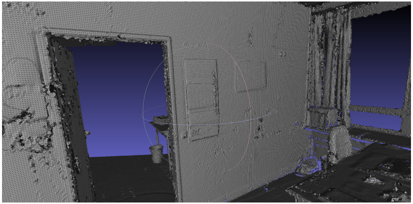
Motivation

Point Clouds



- Generated by depth sensors of 3D laser scanners
- 3D points with geometry data of the environment
- No topological connection between points
- Polygon meshes based on point clouds

Reconstruction



- Lower memory usage without loss of information
- Normal of each point is necessary to create mesh
- Pre-calculation of normals is possible
- Requires nearest neighbours of each point

Nearest Neighbor Search

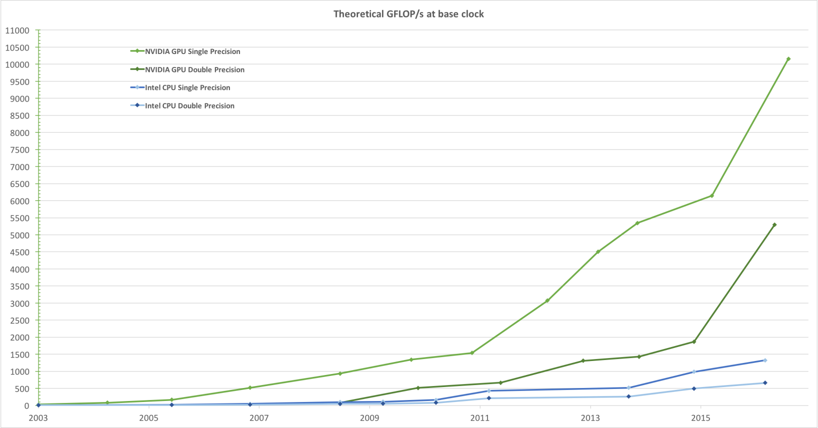
- Independent search of k nearest neighbors to each point
- Long runtimes on large datasets

Nearest neighbor search is the bottleneck of normal calculation

- Thread-based CPU implementation in the LVR Framework

$\Rightarrow$ Parallel implementation on GPU to reduce runtime

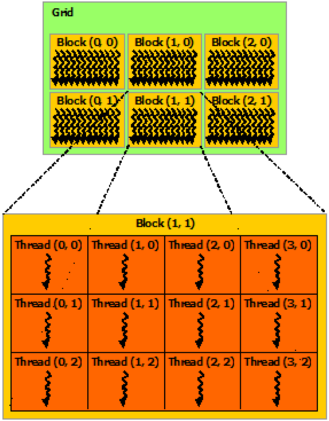
GPU vs. CPU


Theoretical GFLOP/s at base clock
NVIDIA GPU Single Precision
NVIDIA GPU Double Precision
Intel CPU Single Precision
Intel CPU Double Precision
11000
10500
10000
9500
9000
8500
8000
7500
7000
6500
6000
5500
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
2003
2005
2007
2009
2011
2013
2015

CUDA

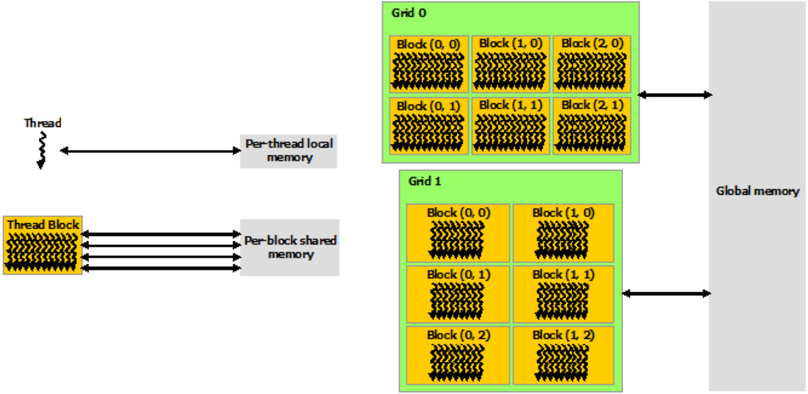
- API for parallel computing on the GPU
- Developed by NVIDIA in 2007
- Newest version: 8.0
- Supports programming languages C/C++, Fortran, Python and many more

Threads



- Parallelization via "Device Kernels" with threads
- Organized in blocks on a grid
- Block contains 1024 threads at maximum
- Blocks have access to shared memory

Memory


Thread
Per-thread local memory
Thread Block
Per-block shared memory
Grid 0
Block (0, 0)
Block (1, 0)
Block (2, 0)
Block (0, 1)
Block (1, 1)
Block (2, 1)
Grid 1
Block (0, 0)
Block (1, 0)
Block (0, 1)
Block (1, 1)
Block (0, 2)
Block (1, 2)
Global memory

Example

Parallel kNN Search

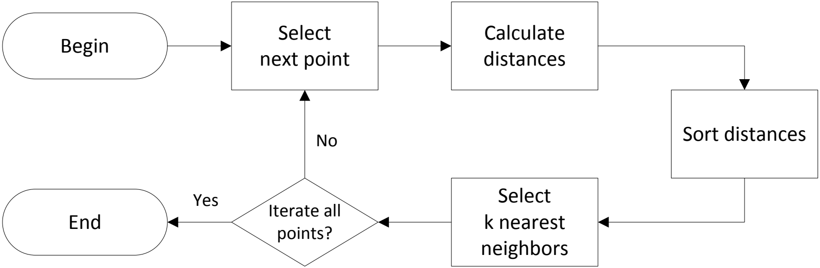
Naive Search

- Highly parallelisable
- Low memory usage
- Quadratic runtime
-

kd-Tree

- Only partial parallelisable
- Additional memory for tree representation necessary
- Linear runtime for small k
-

Basic Idea


Begin
Select next point
Calculate distances
Sort distances
Select k nearest neighbors
Iterate all points?
No
Yes
End

Implementation

for all points in data **do**
  calculate distances to all other points
  repeat
    count neighbors in radius of ε
    adapt $\varepsilon = \varepsilon \cdot (1 \pm \eta)$
  until number of neighbors in radius $\varepsilon = k$
  get points in radius of ε
end for

Implementation

for all points in data **do**

calculate distances to all other points

repeat

count neighbors in radius of ε

adapt $\varepsilon = \varepsilon \cdot (1 \pm \eta)$

until number of neighbors in radius $\varepsilon = k$

get points in radius of ε

end for

Distance Calculation

$$\begin{pmatrix} 6 & 0 & 1 & 6 \\ 2 & 5 & 7 & 9 \\ 4 & 5 & 9 & 4 \end{pmatrix}$$

Distance Calculation

$$\begin{pmatrix} 6 & 0 & 1 & 6 \\ 2 & 5 & 7 & 9 \\ 4 & 5 & 9 & 4 \end{pmatrix}$$

1. Select point from matrix

Distance Calculation

$$\begin{pmatrix} \color{red}{6} & 0 & 1 & 6 \\ \color{red}{2} & 5 & 7 & 9 \\ \color{red}{4} & 5 & 9 & 4 \end{pmatrix}$$

$$\Downarrow$$

$$\begin{pmatrix} 0 & 6 & 5 & 0 \\ 0 & 3 & 5 & 7 \\ 0 & 1 & 5 & 0 \end{pmatrix}$$

1. Select point from matrix
2. Determine absolute to each point dimension-wise

Distance Calculation

$$\begin{pmatrix} \color{red}{6} & 0 & 1 & 6 \\ \color{red}{2} & 5 & 7 & 9 \\ \color{red}{4} & 5 & 9 & 4 \end{pmatrix}$$
$$\Downarrow$$
$$\begin{pmatrix} 0 & 6 & 5 & 0 \\ 0 & 3 & 5 & 7 \\ 0 & 1 & 5 & 0 \end{pmatrix}$$
$$\Downarrow$$
$$\begin{pmatrix} 0 & 46 & 75 & 49 \end{pmatrix}$$

1. Select point from matrix
2. Determine absolute to each point dimension-wise
3. Calculate $d = x^2 + y^2 + z^2$

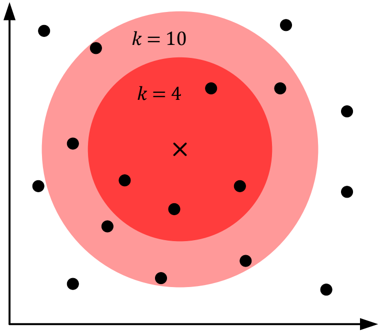
Implementation

for all points in data **do**
  calculate distances to all other points
  repeat
    count neighbors in radius of ε
    adapt $\varepsilon = \varepsilon \cdot (1 \pm \eta)$
  until number of neighbors in radius $\varepsilon = k$
  get points in radius of ε
end for

Implementation

for all points in data **do**
  calculate distances to all other points
  repeat
    count neighbors in radius of ε
    adapt $\varepsilon = \varepsilon \cdot (1 \pm \eta)$
  until number of neighbors in radius $\varepsilon = k$
  get points in radius of ε
end for

Distance "Sorting"



- conventional sorting algorithms not applicable with parallel GPU computing
- evaluate a distance ε based on radius search
- count k inside radius ε
- adapt ε iteratively with learning rate η

Kd-tree

GPU-Implementation

Array based left balanced kd-tree

GPU

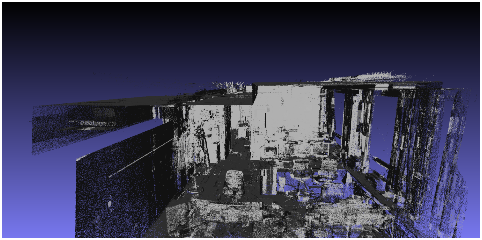
Kd-tree search

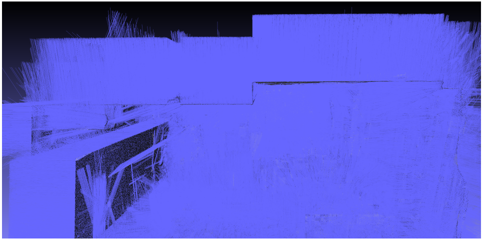
Kd-tree Knn

Normal Calculation

Normal Flip

Results in LVR-Pipeline





Runtime results

Conclusion