CUDA Project

Implement a paper in CUDA: “Surface normal from depth images”

Based on paper: “Adaptive Neighborhood Selection for real-time surface normal estimation from organized point cloud data using integral images”

Table of Contents

1. Introduction

When the camera's intrinsic calibration parameters are known depth images (obtained from Kinect for example) can be converted into organized point clouds. The article aforementioned[1] is about estimating surface normals from this organized point cloud data using integral images. While two methods are discussed in the article I will focus here on the algorithm that Arnaud Schenkel chose to use for his C++ implementation.

The method uses an adaptive window size to analyze local surfaces which allow us to effectively handle depth-depended sensor noise and to avoid common artifacts.

The algorithm can be summarized as follows:

- calculation of integral images
- pre-processing step to compute neighborhood size for each pixel
- computing covariance matrices using the integral images
- estimating surface normals from covariance matrix

2. Article

Before continuing I reference some background Information from the article which can prove useful in understanding the concept:

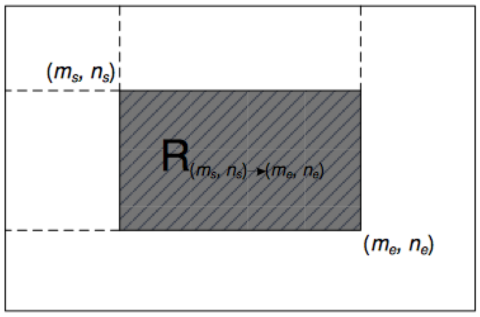


Fig. 1.

[1] S. Holzer and R. B. Rusu and M. Dixon and S. Gedikli and N. Navab, Adaptive Neighborhood Selection for Real-Time Surface Normal Estimation from Organized Point Cloud Data Using Integral Images,International Conference on Intelligent Robots and Systems, 2012.

To be able to compute the area sum of an image O each pixel element $(m,n)^T$ in the integral image I_o is defined as the sum of all elements which are inside of the rectangular area between $O(0,0)$ and $O(m,n)$

An integral image I_o corresponding to an image O makes it possible to compute the sum of all values of O within a certain rectangular region R by accessing only four data elements in memory. Computational cost are thus independent of the size of the rectangle which is useful because it allows us to use varying smoothing sizes using the same amount of memory accesses. Since the noise usually depends on the depth of the perceived data (data acquired at a far distance has a worse SNR than data acquired at close distances) we will use this depth information to estimate the smoothing area. So the size of the rectangle will depend on the smoothing which depends on the depth[2].

After calculating the integral images and estimating the smoothing area the surface normals are estimated by trying to fit a plane into the local neighborhood N_p (of which the size is determined by the smoothing map) of the point of interest. This is done by computing the eigenvectors of the corresponding covariance matrix C_p for a point p at $(m,n)^T$

$$\mathcal{C}_p = \begin{pmatrix} c_{xx} & c_{xy} & c_{xz} \\ c_{yx} & c_{yy} & c_{yz} \\ c_{zx} & c_{zy} & c_{zz} \end{pmatrix} - \begin{pmatrix} c_x \\ c_y \\ c_z \end{pmatrix} \begin{pmatrix} c_x \\ c_y \\ c_z \end{pmatrix}^\top$$

with

$$\begin{aligned} c_x &= \mathcal{S}(\mathcal{I}_{\mathcal{P}_x}, m, n, \mathcal{R}(m,n)), \\ c_y &= \mathcal{S}(\mathcal{I}_{\mathcal{P}_y}, m, n, \mathcal{R}(m,n)), \\ c_z &= \mathcal{S}(\mathcal{I}_{\mathcal{P}_z}, m, n, \mathcal{R}(m,n)). \end{aligned}$$

$$\begin{aligned} c_{xx} &= \mathcal{S}(\mathcal{I}_{\mathcal{P}_{xx}}, m, n, \mathcal{R}(m,n)), \\ c_{xy} = c_{yx} &= \mathcal{S}(\mathcal{I}_{\mathcal{P}_{xy}}, m, n, \mathcal{R}(m,n)), \\ c_{xz} = c_{zx} &= \mathcal{S}(\mathcal{I}_{\mathcal{P}_{xz}}, m, n, \mathcal{R}(m,n)), \\ c_{yy} &= \mathcal{S}(\mathcal{I}_{\mathcal{P}_{yy}}, m, n, \mathcal{R}(m,n)), \\ c_{yz} = c_{zy} &= \mathcal{S}(\mathcal{I}_{\mathcal{P}_{yz}}, m, n, \mathcal{R}(m,n)), \\ c_{zz} &= \mathcal{S}(\mathcal{I}_{\mathcal{P}_{zz}}, m, n, \mathcal{R}(m,n)), \end{aligned}$$

In which $I_{P_x}, I_{P_y}, I_{P_z}$ are the first order and $I_{P_{xx}}, I_{P_{xy}}, I_{P_{xz}}, I_{P_{yy}}, I_{P_{yz}}, I_{P_{zz}}$ the second order for a total of nine integral images.

[2] No border policy was applied here. Since the size of the smoothing is only determined here based on the depth of the point of interest we will also smooth over object borders which can be avoided by also making the smoothing area dependent on large depth changes which are likely to be object borders.

Finally the normal equals to the eigenvector which corresponds to the smallest eigenvalue of the covariance matrix C_p. The authors acknowledge that while this method is computationally expensive the eigenvalues of the covariance matrix can be used to get information about the planarity of the neighborhood.

3. Input/Output

The input file is of PTX/TXT format. The header contains the dimensions of the map, vector, rotation vectors and matrix. The data is composed of the cartesian coordinates, the amplitude and the rgb color.

3 input files where provided for testing[3]. They respectively contain 5, 25 and 100 percent of the data taken of a Town Hall.

Since loading in the data from the PTX/TXT format into a opencv vector is a painfully slow process[4], I converted the input files into a binary format. These files are specific to the operating system since I did not serialize the data.

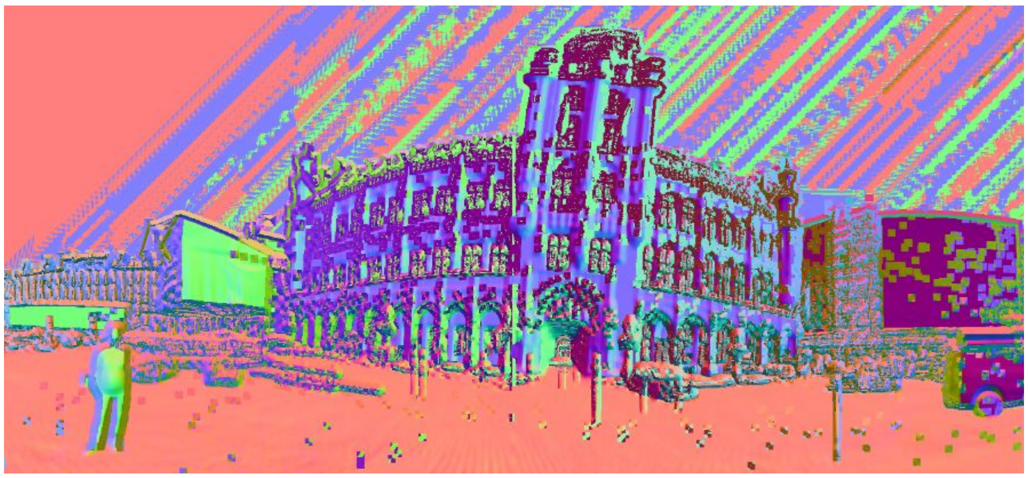


Fig. 2. Output file for the 5% PTX file of the "Town House".

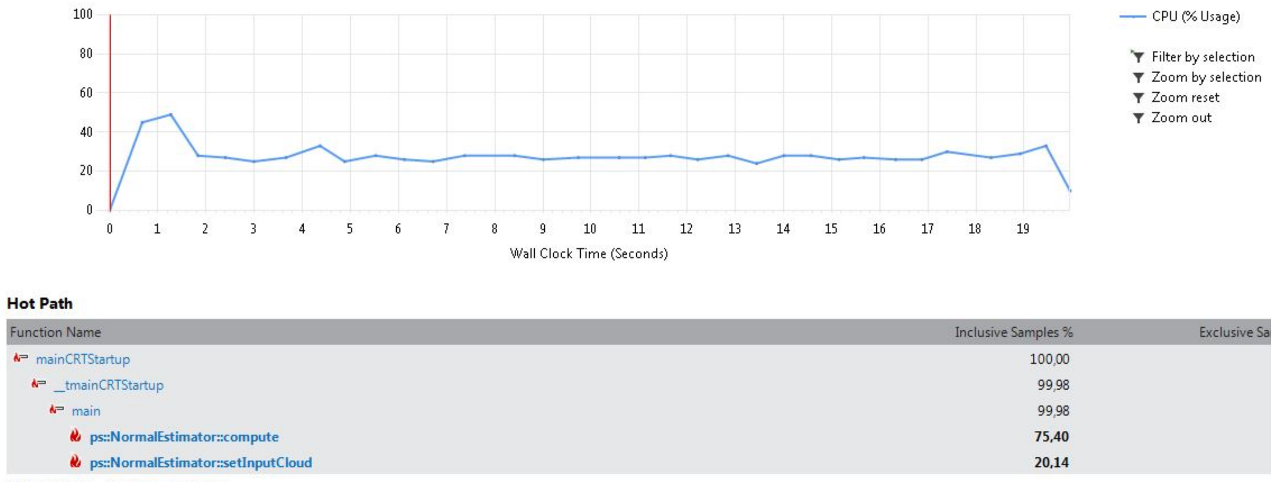
[3] http://lisaserver.ulb.ac.be/~arnaud/Hotel_de_ville/

[4] 57.95s on a Intel Core 2 Quad Q9550 2.83 Ghz for the PTX file containing only 5% of the data.

4. Profiling

Profiling the existing C++ implementation showed (Fig. 3.) that **75 percent of the time is spent inside the compute method** while 20 percent is spent in setInputCloud. The later method is used to compute the integral images while the former is used to for some pre-processing (estimating the smoothing area) and estimating the normals from the covariance matrix.

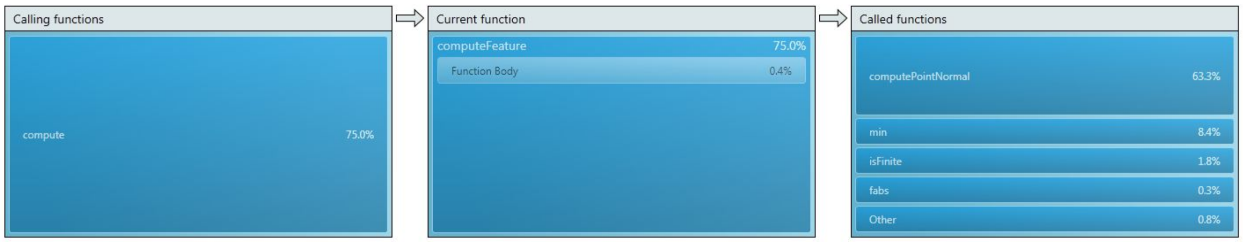
Since most of the work is happening inside `compute` I decided to focus my parallelising effort on that part of the program.



Hot Path

Function Name	Inclusive Samples %	Exclusive Sa
mainCRTStartup	100,00	
__tmainCRTStartup	99,98	
main	99,98	
ps::NormalEstimator::compute	**75,40**	
ps::NormalEstimator::setInputCloud	**20,14**	

Fig. 3. Results from profiling (CPU Sampling) the original serial C++ code.



Calling functions		Current function		Called functions	
compute	75.0%	computeFeature	75.0%	computePointNormal	63.3%
		Function Body	0.4%	min	8.4%
				isFinite	1.8%
				fabs	0.3%
				Other	0.8%

Fig. 4. Detailed view inside the `compute()` function.

5. Serial Code analysis

pseudo code for `compute()`

```
//compute depth change map
//compute distanceMap

for (int rowIdx = border; rowIdx < mapHeight; rowIdx++){
        for (int colIdx = border; colIdx < mapWidth; colIdx++){
            int index = rowIdx * mapWidth + colIdx;
              // check if depth is finite => bad point
              float smoothing = min(distanceMap[index], nss[5]);
              if (smoothing > 2f){
                   if (!initCovarianceMatrix){
                        initCovarianceMatrix() // computes integral images[6]
                   }
                   setRectSize(smoothing, smoothing); //set rectangular reg.
                   computePointNormal(colIdx, rowIdx,index, normal_out[index]);
              } else {
                   // => bad point
              }
        }
}
```

pseudo code for `computePointNormal()`

```
// compute # finite elements within given rectangle
unsigned int count = integral_image_xyz.getFiniteElementsCount(...);
if (count == 0){ // bad point return }

vector<cv::Matx<float,3,1> first_order_elem = integral_image_xyz.getFirstOrderSum(...);
vector<cv::Matx<float,6,1> second_order_elem = integral_image_xyz.getSecondOrderSum(...);

cv::Vec3f center(first_order_elem[0], first_order_elem[1], first_order_elem[2]);
cv::Matx <float, 3,3> covarianceMatrix;

covarianceMatrix.val[0] = second_order_elem[0];
covarianceMatrix.val[1] = covarianceMatrix[3] = second_order_elem[1];
covarianceMatrix.val[2] = covarianceMatrix[6] = second_order_elem[2];
covarianceMatrix.val[4] = second_order_elem[3];
covarianceMatrix.val[5] = covarianceMatrix[7] = second_order_elem[4];
covarianceMatrix.val[8] = second_order_elem[5];
covarianceMatrix -= (center / count * center.transpose());

cv::Mat eigenValue, eigenVector;
cv::eigen(covarianceMatrix, eigenValue, eigenVector); //returns sorted eigenvalues & vectors

float nx = eigenValue.at<float>(2,0);
float ny = eigenValue.at<float>(2,1);
float nz = eigenValue.at<float>(2,2);

flipNormal();
```

[5] nss is a factor which influences the size of the area used to smooth normals

[6] the name `initCovarianceMatrix()` is somewhat confusing here; see footnote 9 for more information.

Parallelizing this code I faced a number of problems since cuda doesn't support any standard library or opencv calls. Also data is represented in vectors and openCV primitives which cuda doesn't understand.

Joachim Kopp implemented[7] a Jacobi algorithm in C++ for his paper on 3x3 hermitian eigen decompositions. The algorithm is optimized for small 3x3 symmetric matrices as opposed to Lapack which openCV uses for eigendecomposition.

Since the Point Cloud Library [8](PCL) does not expose the raw data I slightly modified the code in order to forward integral image data to CUDA.

6. CUDA Implementation

Used `__constant__` memory for parameters that remain fixed (same for all threads).

I didn't use any shared memory as there is no memory overlap between the threads[9].

pseudo code for __global__ void `computePointNormal_cudaKernel(...)`

```
int colIdx = blockIdx.x * blockDim.y + threadIdx.x;
int rowIdx = blockIdx.y * blockDim.y + threadIdx.y;
int index = rowIdx * (gridDim.x * blockDim.x) + colIdx;

if (rowIdx > border && colIdx > border && rowIdx < (height-border) && colIdx < (width-border)){
    if // check if depth is finite => bad point
    else {
       float smoothing = (fminf)(distanceMap[index], nss);
       if (smoothing > 2.0f){
           setRectSize(smoothing, smoothing);
           unsigned int count = integral_image_xyz.getFiniteElementsCount(...);
           if (count == 0){ // bad point return }
           double *first_order_elem = getFirstOrderSum(...);
           double *second_order_elem = = getSecondOrderSum(...)
           double center[3] = center(first_order_elem[0], first_order_elem[1], first_order_elem[2]);
           double covarianceMatrix[3][3] = { {...}, {...}, {...} };

           delete[] first_order_elem;
           delete[] second_order_elem;

           double center_n[3][3] = {{(center[0]/(count)*center[0]),(center[0]/(count)*center[1]),...

           covarianceMatrix[0][0] -= center_n[0][0]; // center / count * center_t (tranpose)
           covarianceMatrix[0][1] -= center_n[0][1];
           covarianceMatrix[0][2] -= center_n[0][2];
```

[7] http://www.mpi-hd.mpg.de/personalhomes/globes/3x3/
[8] http://www.pointclouds.org
[9] if the rectangle size is large enough there is an overlap in the integral image data. This could probably be used to achieve better performance.

```
       (...)

      double eigen_vector_cuda[3][3] = {{0,0,0},{0,0,0},{0,0,0}};
      double eigen_value_cuda[3] = {0,0,0};

      eigen_jacobi(covarianceMatrix, eigen_vector_cuda, eigen_value_cuda);

      //transpose eigen vectors
      //sort eigen values and eigen vectors

      float nx_cuda = eigen_vector_cuda[2][0];
      float ny_cuda = eigen_vector_cuda[2][1];
      float nz_cuda = eigen_vector_cuda[2][2];

      //flip normals

      normal[index].x = nx_cuda;
      normal[index].y = ny_cuda;
      normal[index].z = nz_cuda;
    }
  }
}
```

I also wrote a smaller kernel for the calculation of the depth map.

7. Occupancy and Memory

Initially I had a low occupancy (around 16-40% percent) independent of the chosen block size. The bottleneck proved to be the high number of registers (around 40) allocated by the compiler. After reducing the number of registers to 25 (using the “maxregcount” compiler flag) the occupancy rose to 75% with a block size of 361. Performance increased by 12%.

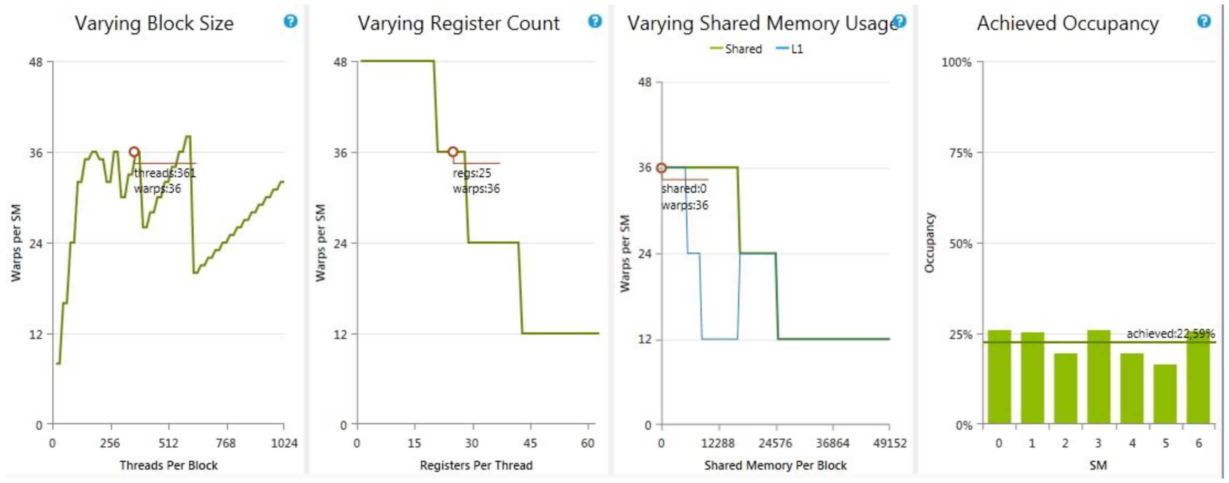


Fig. 4. Nsight Occupancy Graphs

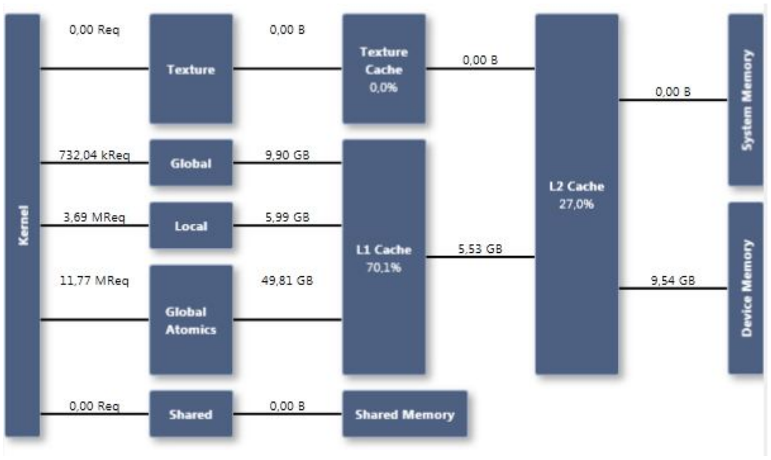
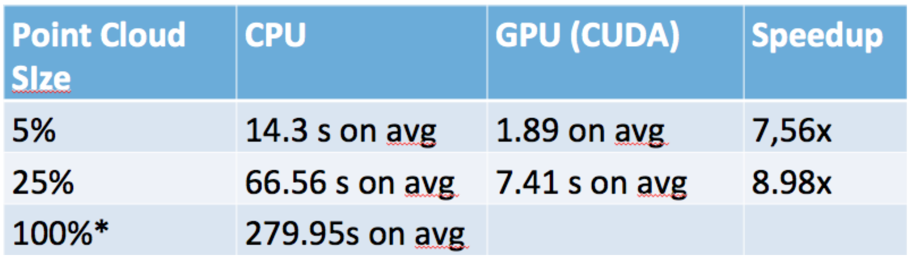


Fig. 5. Nsight Memory Statistics

8. Results

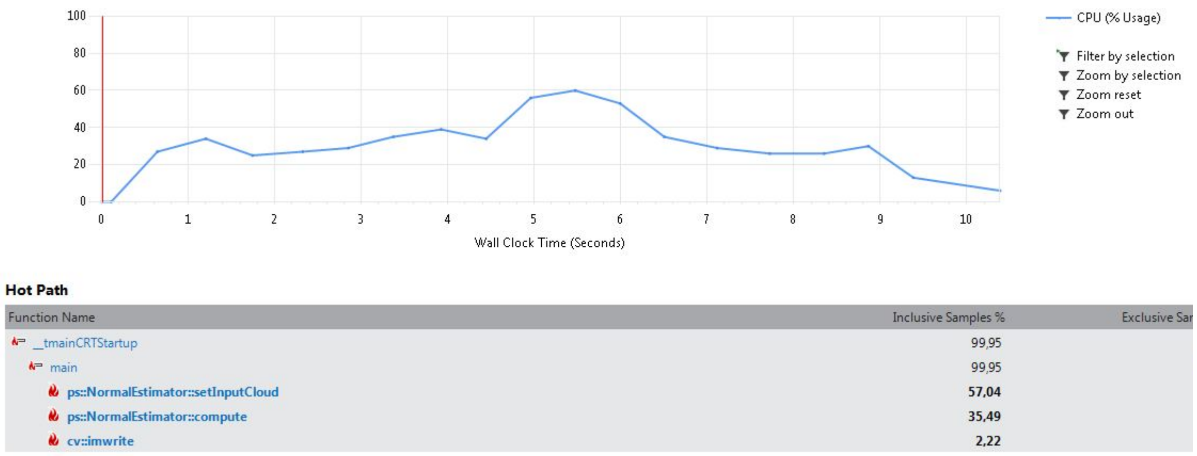


Point Cloud SIze	CPU	GPU (CUDA)	Speedup
5%	14.3 s on avg	1.89 on avg	7,56x
25%	66.56 s on avg	7.41 s on avg	8.98x
100%*	279.95s on avg		

Table 1. Timing measurements for `compute()` method[10]

Update: *In my original presentation I made an measurement error*[11]*. I accidentally re-calculated the integral images before starting the cuda kernel. Realizing my error I re-ran the timing experiments and updated the results and images accordingly.*

On average is get an **8x increase in performance** between the serial version and parallel version of the `compute()` method. We can also see the performance increases for the larger file probably due to a larger number of total threads available for scheduling.



Hot Path

Function Name	Inclusive Samples %	Exclusive Sar
__tmainCRTStartup	99,95	
main	99,95	
ps::NormalEstimator::setInputCloud	57,04	
ps::NormalEstimator::compute	35,49	
cv::imwrite	2,22	

Fig. 5. Results from profiling (CPU Sampling) the CUDA implementation.

We can see that the compute method is no longer the bottleneck in this system. Future work would thus concentrate on parallelizing the calculation of the integral images which is provided by the NPP library.

[10] There is no data for the 100% PTX (>600MB) file because my graphics card (GTX 460 1GB) cannot handle the memory requirements. In future implementations this could be solved by splitting the data.

[11] The method `initCovarianceMatrix` is already called in the `setInputCloud()` method. I mistakenly recalled this function in my CUDA implementation rendering all my previous timing measurements invalid.