# Estimate Normals from a Point Cloud

Riccardo Zappa

November 6, 2024

**Abstract**

In this project, we develop a CUDA-based algorithm to estimate surface normals for each point in a point cloud extracted from a 2D image. The key steps involve transforming the image into a point cloud, creating a KD tree for efficient neighbor search, calculating the covariance matrix from neighboring points, and estimating the normals using the eigenvectors of the covariance matrix. The project leverages parallel computing for speedup, specifically using CUDA kernels.

## 1 Introduction

Surface normal estimation is a fundamental task in computer vision and 3D geometry processing. Normals describe the orientation of the surface at each point, which is essential for tasks such as rendering, segmentation, and feature extraction.

The data used for the tests are a some .png images with a different amount of points: 100k, 250k, and 1m .

## 2 Point Cloud Generation from Image

The first step of the project is to generate a point cloud from the images retrieved using Opencv. First we have a classic sequential method to convert from "cv::Mat" to "pointCloud", then we have a simple kernel to parallelize the trasformation from pointCloud to two collections of Point that will be useful for the next passages.

```
const int MAX_DIM = 3;

    struct Point {
    float coords[MAX_DIM];  // Coordinates in MAX_DIM-dimensional space
};
```

The CUDA kernel is used to transform the image into a point cloud by mapping pixel coordinates $(x, y)$ and the depth value $z$ to 3D points $(x, y, z)$. The transformation is done in parallel for each pixel.

## 2.1 function and CUDA Kernel for Image-to-Point Cloud Transformation

The CUDA kernel operates on the image data stored as a 2D array. Each thread in the kernel corresponds to a pixel in the image and computes its respective 3D point.

```
// Convert depth map to point cloud using PCL
pcl::PointCloud<pcl::PointXYZ>::Ptr depthMapToPointCloud(const cv::Mat& depthMap) {
    // Create a new point cloud object
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new pcl::PointCloud<pcl::PointXYZ>);

    // Loop through each pixel in the depth map
    for (int v = 0; v < depthMap.rows; ++v) {
        for (int u = 0; u < depthMap.cols; ++u) {
            float depthValue = depthMap.at<uint8_t>(v, u);  // Grayscale depth value
            float Z = depthValue;  // Depth value (this can be scaled as per your nee

            if (Z > 0) {  // Ignore points with zero or invalid depth
                float X = (u - cx) * Z / fx;
                float Y = (v - cy) * Z / fy;

                // Add the 3D point to the point cloud
                pcl::PointXYZ point;
                point.x = X;
                point.y = Y;
                point.z = Z;
                cloud->points.push_back(point);
            }
        }
    }

    // Set the width and height of the point cloud
    cloud->width = cloud->points.size();
    cloud->height = 1;  // Unordered point cloud (1 row)
    cloud->is_dense = false;

    return cloud;
}


// CUDA kernel to convert pcl::PointXYZ to Point
__global__ void convertPointCloudKernel(const pcl::PointXYZ* inputPoints, Point* outp
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < numPoints) {
        outputPoints[idx].coords[0] = inputPoints[idx].x;  // x coordinate
        outputPoints[idx].coords[1] = inputPoints[idx].y;  // y coordinate
        outputPoints[idx].coords[2] = inputPoints[idx].z;  // z coordinate
        //queries
        outputQueries[idx].coords[0] = inputPoints[idx].x;  // x coordinate
```

```
        outputQueries[idx].coords[1] = inputPoints[idx].y;  // y coordinate
        outputQueries[idx].coords[2] = inputPoints[idx].z;  // z coordinate
    }
}
```

# 3 KD Tree for k-Nearest Neighbors Search

Once the point cloud is generated, the next step is to find the $k$ nearest neighbors for each point. For efficient neighbor searching, we use a KD tree structure. KD trees enable fast nearest neighbor queries by organizing points in a spatial hierarchy.

```
    struct KDNode {
    Point point;
    int left;
    int right;
    int axis;  // Splitting axis (0 to MAX_DIM-1)
};
```

## 3.1 KD Tree Construction

The KD tree is built by recursively splitting the point cloud along one dimension at each level of the tree.

```
    // Recursive function to build KDTree
__host__ void buildSubTree(Point *points, KDNode *tree, int start, int end, int depth
    if (start >= end) return;

    int axis = depth % MAX_DIM;
    std::sort(points + start, points + end, PointComparator(axis));

    int split = (start + end - 1) / 2;
    tree[node].point = points[split];
    tree[node].axis = axis;

    buildSubTree(points, tree, start, split, depth + 1, node * 2);
    buildSubTree(points, tree, split + 1, end, depth + 1, node * 2 + 1);
}

// Function to initialize the KDTree
__host__ void buildKDTree(Point *points, KDNode *tree, int n, int treeSize) {
    for (int i = 0; i < treeSize; i++) {
        tree[i].left = tree[i].right = -1;  // Default values
    }
    buildSubTree(points, tree, 0, n, 0, 1);
}
```

## 3.2 CUDA Kernel for Neighbor Search

After constructing the KD tree, another CUDA kernel performs a $k$-nearest neighbor search for each point in the point cloud. The neighbors are needed to calculate the covariance matrix for normal estimation.

```
// Recursive device function for finding K nearest neighbors
 __device__ void findKNearestNeighbors(KDNode *tree, int treeSize, int treeNode, int
    // Base case
    if (treeNode >= treeSize) return;

    KDNode node = tree[treeNode];
    if (node.axis == -1) return;

    bool near = false;
    int max = 0;
    float dist = 0;
    int index = 0;
     // Push the current node point into neighbors array
    for (int i = 0; i < k; i++) {
        if ( distance(node.point, query) < distance(neighbors[i], query) ) {
            near = true;
            break;
        }
    }
    if (near)
    {
        for (int j=0; j < k; j++)
        {
            dist = distance(neighbors[j], query);
            if (max < dist)
            {
                max = dist;
                index = j;
            }
        }
        neighbors[index] = node.point;
    }
    if (query.coords[node.axis] < node.point.coords[node.axis]) {
         findKNearestNeighbors(tree, treeSize, treeNode * 2, depth + 1, query, neighb
    } else {
         findKNearestNeighbors(tree, treeSize, treeNode * 2 + 1, depth + 1, query, ne
    }
 }



// Kernel to perform K nearest neighbor search for all queries
__global__ void kNearestNeighborsGPU(KDNode *tree, int treeSize, Point *queries, Poin
```

```
    int index = blockIdx.x * blockDim.x + threadIdx.x;

    if (index < nQueries) {
        __shared__ Point neighbors[K_NEIGHBORS];  // Local array to store the K neare

        for (int i = 0; i < k; i++) {
            neighbors[i].coords[0] = INF;
            neighbors[i].coords[1] = INF;
            neighbors[i].coords[2] = INF;  // Assuming Point() initializes it to an i
        }
        findKNearestNeighbors(tree, treeSize, 1, 0, queries[index], neighbors, k);

        // Copy the neighbors to the results array
        for (int i = 0; i < k; i++) {
            results[index * k + i] = neighbors[i];
        }
    }
}
```

# 4 Covariance Matrix Calculation

The covariance matrix provides a measure of the local geometric structure around a point. For a point $\mathbf{p}_i$, the covariance matrix $\Sigma$ is calculated from its $k$ neighbors $\mathbf{p}_j$.

## 4.1 Mathematical Definition

The covariance matrix $\Sigma$ for a point $\mathbf{p}_i$ is given by:

$$\Sigma = \frac{1}{k} \sum_{j=1}^{k} (\mathbf{p}_j - \bar{\mathbf{p}})(\mathbf{p}_j - \bar{\mathbf{p}})^T$$

where $\mathbf{p}_j$ are the $k$ neighbors of $\mathbf{p}_i$, and $\bar{\mathbf{p}}$ is the mean of the neighboring points:

$$\bar{\mathbf{p}} = \frac{1}{k} \sum_{j=1}^{k} \mathbf{p}_j$$

## 4.2 CUDA Kernel for Covariance Matrix Calculation

For each point, the CUDA kernel calculates the covariance matrix from its $k$ nearest neighbors.

```
__global__ void computeCovarianceMatrix(Point* points, Point* neighbors, float* covar
    int pointIdx = blockIdx.x;  // Each block processes one point
    int tid = threadIdx.x;      // Thread ID within the warp

    if (pointIdx >= numPoints || tid >= kN) return;  // Out-of-bounds protection

    // Step 1: Calculate mean (parallelized by threads within the warp)
```

```
__shared__ float mean[3];
if (tid < 3) mean[tid] = 0.0f;
__syncthreads();

// Each thread handles one neighbor
Point neighbor = neighbors[pointIdx * kN + tid];

// Accumulate each neighbor's coordinates to the mean
atomicAdd(&mean[0], neighbor.coords[0]);
atomicAdd(&mean[1], neighbor.coords[1]);
atomicAdd(&mean[2], neighbor.coords[2]);

__syncthreads();

// Compute the mean for this point (only the first 3 threads handle this)
if (tid < 3) {
    mean[tid] /= kN;
}
__syncthreads();

// Step 2: Calculate the covariance matrix (each thread handles one neighbor)
__shared__ float cov[3][3];
if (tid < 9) {  // Initialize the shared covariance matrix
    cov[tid / 3][tid % 3] = 0.0f;
}
__syncthreads();

// Difference from the mean
float diff[3] = {
    neighbor.coords[0] - mean[0],
    neighbor.coords[1] - mean[1],
    neighbor.coords[2] - mean[2]
};

// Accumulate to the covariance matrix in parallel
atomicAdd(&cov[0][0], diff[0] * diff[0]);
atomicAdd(&cov[0][1], diff[0] * diff[1]);
atomicAdd(&cov[0][2], diff[0] * diff[2]);
atomicAdd(&cov[1][0], diff[1] * diff[0]);
atomicAdd(&cov[1][1], diff[1] * diff[1]);
atomicAdd(&cov[1][2], diff[1] * diff[2]);
atomicAdd(&cov[2][0], diff[2] * diff[0]);
atomicAdd(&cov[2][1], diff[2] * diff[1]);
atomicAdd(&cov[2][2], diff[2] * diff[2]);

__syncthreads();

// Step 3: Normalize and store the covariance matrix
```

```
    if (tid < 9) {  // Each thread stores one element of the covariance matrix
        int i = tid / 3;
        int j = tid % 3;
        covarianceMatrices[pointIdx * 9 + i * 3 + j] = cov[i][j] / (kN - 1);

        // Add small value to diagonal for numerical stability
        if (i == j) covarianceMatrices[pointIdx * 9 + i * 3 + j] += 1e-6f;
    }
}
```

# 5 Normal Estimation

Once the covariance matrix is computed for each point, the normals can be estimated by solving for the eigenvectors and eigenvalues of the covariance matrix.

## 5.1 Eigenvalue and Eigenvector Computation

The normal vector $\mathbf{n}_i$ at point $\mathbf{p}_i$ is the eigenvector corresponding to the smallest eigenvalue of the covariance matrix $\Sigma_i$. This eigenvector represents the direction of least variation, which is the surface normal.

$$\Sigma \mathbf{n} = \lambda \mathbf{n}$$

where $\lambda$ is the eigenvalue and $\mathbf{n}$ is the corresponding eigenvector.

## 5.2 CUDA Kernel for Normal Estimation

The CUDA kernel performs eigenvalue and eigenvector decomposition of the covariance matrix for each point and selects the eigenvector with the smallest eigenvalue as the normal.

```
__host__ void computeNormalsWithCuSolver(float* covarianceMatrices, Point* normals, i
    // cuSolver handle
    cusolverDnHandle_t cusolverH = NULL;
    cusolverDnCreate(&cusolverH);

    // Allocate device memory
    float *d_covarianceMatrices, *d_eigenValues, *d_eigenVectors;
    Point *d_normals;
    int *d_info;
    float *d_work;

    eChk(cudaMalloc((void**)&d_covarianceMatrices, nPoints * dim * dim * sizeof(float
    eChk(cudaMalloc((void**)&d_eigenValues, nPoints * dim * sizeof(float)));
    eChk(cudaMalloc((void**)&d_eigenVectors, nPoints * dim * dim * sizeof(float)));
    eChk(cudaMalloc((void**)&d_normals, nPoints * sizeof(Point)));
```

```
 eChk(cudaMalloc((void**)&d_info, sizeof(int)));

 // Copy data to device
 eChk(cudaMemcpy(d_covarianceMatrices, covarianceMatrices, nPoints * dim * dim * s

// Debug: Print covariance matrix passed
 float debugMatrix[9];
 cudaMemcpy(debugMatrix, covarianceMatrices, 9 * sizeof(float), cudaMemcpyDeviceTo
 std::cout << "covariance matrix passed to debug:" << std::endl;
 for (int i = 0; i < 3; i++) {
     for (int j = 0; j < 3; j++) {
         std::cout << debugMatrix[i*3 + j] << " ";
     }
     std::cout << std::endl;
 }

 // Workspace size
 int lda = dim;
 int workSize = 0;
 CUSOLVER_CHECK(cusolverDnSsyevd_bufferSize(cusolverH, CUSOLVER_EIG_MODE_VECTOR, C
                            dim, d_covarianceMatrices, lda, d_eigenValues, &workS

 cudaMalloc((void**)&d_work, workSize * sizeof(float));

 // Compute eigenvalues and eigenvectors for all matrices:the eigenvectors will be
 CUSOLVER_CHECK(cusolverDnSsyevd(cusolverH, CUSOLVER_EIG_MODE_VECTOR, CUBLAS_FILL_
                 dim, d_covarianceMatrices, lda, d_eigenValues, d_work, workSize,

 // Check if the operation was successful
 int info;
 eChk(cudaMemcpy(&info, d_info, sizeof(int), cudaMemcpyDeviceToHost));

 // Launch kernel to compute normals
 int blocks = (nPoints + BLOCK_SIZE - 1) / BLOCK_SIZE;

 computeNormalsKernel<<<blocks, BLOCK_SIZE>>>(d_covarianceMatrices, d_eigenValues,
 eChk(cudaDeviceSynchronize());
 // Copy results back to host
 cudaMemcpy(normals, d_normals, nPoints * sizeof(Point), cudaMemcpyDeviceToHost);

 // Cleanup
 cusolverDnDestroy(cusolverH);
 cudaFree(d_covarianceMatrices);
 cudaFree(d_eigenValues);
 cudaFree(d_eigenVectors);
 cudaFree(d_normals);
 cudaFree(d_info);
 cudaFree(d_work);
```

```
}

__global__ void computeNormalsKernel(float* d_covarianceMatrices, float* d_eigenValue
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < nPoints) {
        // Find the eigenvector corresponding to the smallest eigenvalue
        int minIndex = 0;
        for (int j = 1; j < dim; j++) {
            if (d_eigenValues[idx * dim + j] < d_eigenValues[idx * dim + minIndex]) {
                minIndex = j;
            }
        }
        // Copy the normal vector (corresponding to the smallest eigenvalue)
        for (int d = 0; d < dim; d++) {
            d_normals[idx].coords[d] = d_eigenVectors[idx * dim * dim + d + minIndex
        }
    }
}
```

# 6 GPU specification

The GPU hardware specifications are

| Field | Value |
|---|---|
| Device Name | NVIDIA GeForce RTX 4060 Laptop GPU |
| CUDA Driver Version / Runtime Version | 12.7 / 12.4 |
| GPU Architecture Name | Ada |
| CUDA Capability Version | 8.9 |
| Total Global Memory | 8188 MB (8585216000 bytes) |
| Multiprocessors | 24 |
| CUDA Cores per Multiprocessor | 128 |
| Total CUDA Cores | 3072 |
| GPU Max Clock Rate | 1755 MHz (1.75 GHz) |
| Memory Clock Rate | 8001 MHz |
| Memory Bus Width | 128-bit |
| L2 Cache Size | 33554432 bytes |
| Maximum 1D Texture Size | 131072 |
| Maximum 2D Texture Size | 131072 x 65536 |
| Maximum 3D Texture Size | 16384 x 16384 x 16384 |
| Maximum 1D Texture Layers | 32768 (2048 layers) |
| Maximum 2D Texture Layers | 32768 x 32768 (2048 layers) |
| Total Constant Memory | 65536 bytes |
| Shared Memory per Block | 49152 bytes |
| Registers per Block | 65536 |
| Warp Size | 32 |
| Max Threads per Multiprocessor | 1536 |
| Max Threads per Block | 1024 |
| Max Thread Block Dimension | (1024, 1024, 64) |
| Max Grid Dimension Size | (2147483647, 65535, 65535) |
| Maximum Memory Pitch | 2147483647 bytes |
| Texture Alignment | 512 bytes |
| Concurrent Copy and Kernel Execution | Yes (with 1 copy engine) |
| Run Time Limit on Kernels | Yes |
| Integrated GPU Sharing Host Memory | No |
| Support for Host Page-Locked Memory Mapping | Yes |
| Alignment Requirement for Surfaces | Yes |
| ECC Support | Disabled |
| Unified Addressing (UVA) Support | Yes |
| Device PCI Domain ID / Bus ID / Location ID | 0 / 1 / 0 |

Table 1: NVIDIA GeForce RTX 4060 Laptop GPU Specifications

# 7 Results and Conclusion

To track the results and the speedUp of the cuda project, i have created other two c++ implementation of it, one using the PCL library function and the other being a raw

implementation.

We confront in the next table the execution time of the projects compared using the dimension of the point cloud passed as an input:

| | 100k | 250k | 1m |
|---|---|---|---|
| CUDA | 1020.34ms | 1715.35ms | 4261.12 ms |
| Raw C++ | 6704.6 ms | 16398.7 ms | 45152.9 ms |
| PCL Library | 1879.78 ms | 6323.51 ms | 59732.6 ms |

Table 2: execution times related to the point cloud input dimension

As we can see, there is a big improvement in the performance oof the cuda kernel related to the other two projects as the input dimension grows. In particular we have a speedUp of 6.5(raw) and 1.84(Pcl) for the 100k pointCloud, 9.55(raw) and 3.63(Pcl) for the 250k one and a consistent 10.59(raw) and 14.01(Pcl).

Now we go more in depth analysing the kernels data and execution times using a Nsight compute as a profiler.

| | Point Cloud 1 | Point Cloud 2 | Point Cloud 3 |
|---|---|---|---|
| convertPointCloudKernel | 17.44 us | 38.18 us | 106.40 us |
| kNearestNeighboursGpu | 60.77ms | 156.23 ms | 459.89 ms |
| computeCovarianceMatrixKernel | 13.07 ms | 48.10 ms | 98.50 ms |
| computeNormalsKernel | 861.54 us | 1.33 ms | 799.46 us |

Table 3: kernels execution times

For the occupancy of each kernel, we will utilize the data related to the execution with the pointCloud with 1m points.

| | |
|---|---|
| Theoretical Occupancy [%] | 100 |
| Block Limit Registers [block] | 5 |
| Theoretical Active Warps per SM [warp] | 48 |
| Block Limit Shared Mem [block] | 8 |
| Achieved Occupancy [%] | 82.26 |
| Block Limit Warps [block] | 3 |
| Achieved Active Warps Per SM [warp] | 39.48 |
| Block Limit SM [block] | 24 |

Table 4: convertPointCloudKernel

| | |
|---|---|
| Theoretical Occupancy [%] | 100 |
| Block Limit Registers [block] | 3 |
| Theoretical Active Warps per SM [warp] | 48 |
| Block Limit Shared Mem [block] | 8 |
| Achieved Occupancy [%] | 60.52 |
| Block Limit Warps [block] | 3 |
| Achieved Active Warps Per SM [warp] | 29.05 |
| Block Limit SM [block] | 24 |

Table 5: kNearestNeighboursGpu

| | |
|---|---|
| Theoretical Occupancy [%] | 50 |
| Block Limit Registers [block] | 84 |
| Theoretical Active Warps per SM [warp] | 24 |
| Block Limit Shared Mem [block] | 28 |
| Achieved Occupancy [%] | 52.16 |
| Block Limit Warps [block] | 48 |
| Achieved Active Warps Per SM [warp] | 25.04 |
| Block Limit SM [block] | 24 |

Table 6: computeCovarianceMatrixKernel

| | |
|---|---|
| Theoretical Occupancy [%] | 100 |
| Block Limit Registers [block] | 3 |
| Theoretical Active Warps per SM [warp] | 48 |
| Block Limit Shared Mem [block] | 8 |
| Achieved Occupancy [%] | 71.28 |
| Block Limit Warps [block] | 3 |
| Achieved Active Warps Per SM [warp] | 34.21 |
| Block Limit SM [block] | 24 |

Table 7: ComputeNormalsKernel

# 8 Improvements

For this cuda project there is surely room for improvements, in particular in two parts:

1) the managemente of the KD-Tree building, that is done sequentially and not leveraging on gpu. For this there is an implementation done in github by a nvidia worker that could be used as inspiration. `https://github.com/ingowald/cudaKDTree`

2)the nearest Neighbours search, in this project is done recursevely, launching a function for each thread in the kernel. There is a project done by some Phd students that can be used as reference. `https://vincentfpgarcia.github.io/kNN-CUDA/`

# References


- Point Cloud Library. *Normal Estimation*. Available: https://pcl.readthedocs.io/projects/tutorials/en/latest/normal_estimation.html.
- NVIDIA, *CUDA Programming Guide*.
- KNN, *cuda implementation* Available: https://vincentfpgarcia.github.io/kNN-CUDA/
- KD-tree, *cuda implementation* Available: https://github.com/ingowald/cudaKDTree