Accelerating Random Forests on CPUs and GPUs for Object-Class Image Segmentation

Master's Thesis

Benedikt Waldvogel

First Examiner: Prof. Dr. Sven Behnke
Second Examiner: Dr. Simone Frintrop
Advisor: Hannes Schulz, M. Sc.

Submitted on: July 18, 2013

Declaration of Authorship

I declare that the work presented here is original and the result of my own investigations. Formulations and ideas taken from other sources are cited as such.

It has not been submitted, either in part or whole, for a degree at this or any other University.

__________________________ __________________________

Location, Date Signature

Abstract

Random forests are a machine learning method that has recently become popular in the computer vision community to solve image segmentation and object detection tasks. Existing random forest implementations are either general purpose and not efficiently applicable for image segmentation or focus only on the speed of prediction. The implementation for the Microsoft Kinect gaming platform, for instance, achieves real-time speed on a single Microsoft Xbox GPU to recognize the pose of the user. Random forest training, however, has been conducted on a large cluster with 1000 CPU cores. Generally, training on large datasets is computationally demanding and impedes scientific research since the process takes long if a computing cluster is not available or too expensive for the task at hand.

It is the goal of this master's thesis to accelerate training and prediction of random forests for object-class image segmentation on RGB-D datasets by efficiently using CPUs and the massively parallel computing power offered by GPUs. We present an implementation that runs up to 28 times faster on GPU and is capable to train a random forest in less than four minutes on a GPU; thus drastically abbreviating a process that previously took about one whole day on a CPU. Dense classification of RGB-D images in VGA resolution runs in real-time speed on a single mobile GPU.

Contents

List of Figures vi

List of Tables vii

List of Algorithms ix

1. Introduction 1

2. Related Work 3

3. Foundations 5

- 3.1. Random Forests 5
  - 3.1.1. Random Forest Training 6
  - 3.1.2. Random Forest Prediction 11
  - 3.1.3. Random Forest Variants 11
- 3.2. Visual Features for Random Forests 13
  - 3.2.1. Parameters 14
  - 3.2.2. Color Feature 14
  - 3.2.3. Depth Feature 15
  - 3.2.4. Integral Images 15
- 3.3. Measuring Image Segmentation Accuracy 17
  - 3.3.1. Void and Background Pixels 17
  - 3.3.2. Average Per-Class Accuracy Measure 18
  - 3.3.3. Pixel Accuracy Measure 18
- 3.4. Speed-up Measures 18
- 3.5. The CUDA GPU Programming Framework 18
  - 3.5.1. GPU Programming Model 19
  - 3.5.2. GPU Memory 20
- 3.6. Memory-Efficient n-Dimensional Array Calculations 26
  - 3.6.1. Row-major and Column-major Array Order 27

4. Accelerating Random Forests 28

- 4.1. Implementation of Visual Features 28
- 4.2. Accelerating Random Forest Training 28
  - 4.2.1. Breadth-first Training 29
  - 4.2.2. Random Forest Training on CPU 29
  - 4.2.3. Theoretical Computing Boundaries 31
  - 4.2.4. Random Forest Training on GPU 33
- 4.3. Accelerating Random Forest Prediction 48
  - 4.3.1. Random Forest Prediction on CPU 49

4.3.2. Random Forest Prediction on GPU . . . 49
4.4. Accelerating Random Ferns . . . 52
4.4.1. Random Fern Training . . . 52
4.4.2. Random Fern Prediction on GPU . . . 53

5. Experimental Results **54**
5.1. Datasets . . . 54
5.1.1. AIS Bonn Large-Objects dataset . . . 54
5.1.2. NYU Depth v2 dataset . . . 58
5.2. Parameters . . . 60
5.3. Random Feature Candidate Generation per Tree Level . . . 62
5.4. Random Forest Training on GPU . . . 63
5.4.1. Breakdown of Random Forest Training Time . . . 65
5.4.2. Discussion of the Results . . . 68
5.5. Random Forest Prediction on GPU . . . 69
5.5.1. Discussion of the Results . . . 69
5.6. Segmentation Accuracy . . . 70
5.6.1. Information Gain versus Normalized Information Gain . . . 70
5.6.2. Depth Filling . . . 71
5.6.3. RGB versus CIE Lab . . . 74
5.6.4. Segmentation Accuracy on the NYU Depth v2 dataset . . . 74
5.6.5. Discussion of the Results . . . 75
5.7. Random Fern Prediction on GPU . . . 75
5.7.1. Discussion of the Results . . . 76

6. Conclusion **77**
6.1. Future Work . . . 78

Glossary **79**

Bibliography **81**

A. Appendix **86**

List of Figures

3.1. Example of a binary decision tree. 6
3.2. Example of a random forest and a random fern. 12
3.3. Example of a visual feature. 14
3.4. Example of a RGB-D image. 15
3.5. Example of an integral image. 16
3.6. Example of an image in the NYU Depth v2 dataset. 17
3.7. Comparison of memory bandwidth and computing power of CPUs and GPUs. 19
3.8. Schematic diagram of a CPU and a GPU. 19
3.9. Schematic diagram of GPU threads, blocks and the grid. 20
3.10. Example of an efficient memory access on GPU. 23
3.11. Example of an inefficient memory access on GPU. 23
3.12. Example of a memory bank mapping. 24
3.13. Example of a matrix in row-major and column-major order. 27

4.1. Thread block layout of the feature response kernel. 36
4.2. Training time and cache hit rate of the feature response calculation with respect to the maximum region offset. 38
4.3. Thread block layout of the histogram aggregation kernel. 43
4.4. Schema of histogram counter reduction in shared memory. 44
4.5. Schema of histogram counter reduction in shared memory without bank conflicts. 45
4.6. Example for the mapping of a tree structure to a texture on GPU. 51

5.1. Example of three images in the AIS Bonn Large-Objects dataset 56
5.2. Comparison of depth filling variants in the AIS Bonn Large-Objects dataset. 57
5.3. Example images of the NYU Depth v2 dataset 59
5.4. Development of the loss function during a hyper-parameter search. 61
5.5. Breakdown of random forest training time per tree level. 66
5.6. Feature response calculation runtime with respect to the number of samples. 67
5.7. Histogram aggregation runtime with respect to the number of samples. . . . 67
5.8. Examples for random forest predictions with the NYU Depth v2 dataset. . 73
5.9. Relative color feature frequency per tree level. 74

A.1. Relative feature frequency per image channel pair. 86

List of Tables

3.1. Runtime comparison of various transpose kernels. 25

4.1. Training time for feature candidate generation. 35
4.2. Training time and cache hit rate of the feature response calculation with respect to sorting of features and samples. 39
4.3. Training time and cache hit rate of the feature response calculation with respect to the thread block size. 40
4.4. Training time and cache hit rate of the feature response calculation for 20000 samples from 100 images. 41
4.5. Histogram aggregation kernel runtime with respect to the number of feature and threshold candidates. 46
4.6. Histogram aggregation kernel runtime with respect to the number of samples. 46
4.7. Score kernel runtime for different impurity score functions. 47
4.8. Runtime of random forest prediction with respect to the patch size. 50

5.1. Hyper-parameter search space for the NYU Depth v2 dataset. 60
5.2. Random forest training parameters. 61
5.3. Segmentation accuracy with feature candidate generation per-level and per-node. 63
5.4. Comparison of random forest training time on CPU and GPU. 64
5.5. Date of introduction, price and thermal design power for the CPU and GPUs. 64
5.6. Power consumption of random forest training on CPU and GPU. 65
5.7. Comparison of random forest prediction runtime on CPU and GPU. 69
5.8. Comparison of random forest training time and segmentation accuracy with respect to the impurity score function. 70
5.9. Comparison of random forest segmentation accuracy for filled depth and raw depth on the AIS Bonn Large-Objects dataset. 72
5.10. Comparison of random forest segmentation accuracy for filled depth and raw depth on the NYU Depth v2 dataset. 72
5.11. Comparison of segmentation accuracy for RGB and CIELab color space . . 74
5.12. Comparison of the segmentation accuracy on the NYU Depth v2 dataset with state-of-the-art methods. 75
5.13. Comparison of prediction runtime of random forests and random ferns. . . . 76

A.1. Training time and cache hit rate of the feature response calculation for 2000 samples from 100 images. 87
A.2. Training time and cache hit rate of the feature response calculation for 20000 samples from 10 images. 87
A.3. Training time and cache hit rate of the feature response calculation for 20000 samples from 100 images. 88

A.4. Training time and cache hit rate of the feature response calculation for 2000 images from 100 images. 88

List of Algorithms

1. Depth-first training of a random decision tree 7
2. Breadth-first training of a random decision tree 8

3. Naïve implementation of the feature evaluation 30
4. Feature evaluation optimized for the CPU cache 30
5. Histogram aggregation on GPU 42
6. Random forest prediction on GPU 51

1. Introduction

Visual perception belongs to the most important aspects to acquire knowledge about the world that surrounds us. Precise information about our environment is an essential input to plan and execute further actions. The visual system in the human brain accomplishes this task with fascinating results. The computer vision community aims to achieve the same quality level since their existence and researchers have made significant progress to accomplish that goal in recent years.

Sophisticated computer vision systems have been constructed that process and analyse digital images in order to understand the content. Segmentation of the digital image is one step in the processing chain of such a system and defines the task to divide an image into multiple regions where a class label is assigned to every pixel in the image. The determined segments serve as input for subsequent processing steps such as, for example, the selection of interesting regions in an image.

Random forests are a machine learning method that has become popular in the computer vision community since Lepetit et al. [28] published their work on random forests for keypoint recognition. Since then, several publications have depicted the application of random forests and variants on computer vision tasks such as image segmentation [27, 41, 44, 43, 50, 38].

It is a common claim that random forests are classifiers which allow fast prediction and fast training. However, training is only fast if the dataset is not too large and the choice of features is sufficiently simple.

There has been a recent increase in the availability of large datasets with color and depth images as commodity cameras with depth sensor such as the Microsoft Kinect have become obtainable at low price. Such datasets demand research on new features that operate on color as well as distance information of the captured digital image.

Microsoft for example applies random forests to detect the human body parts in the Microsoft Xbox gaming platform. The technique allows users to operate games without the need for a dedicated device. Toby Sharp [52] and Shotton et al. [43] present a way of applying the processing power of a single Graphics Processing Unit (GPU) to detect the body parts in real-time using random forests. They solve the extreme computational effort of random forest training on a large computer cluster with 1000 Central Processing Unit (CPU) cores.

Companies with a broad end-user reach and according investment capabilities, such as Microsoft, might be able to apply this approach; scientific institutions such as academic research institutes, however, are often not able to invest such amounts of money to solve computationally intensive tasks.

This is one reason why the demand for an implementation increases that uses available computing resources, such as CPUs and GPUs as efficiently as possible, not only for random forest prediction but also for random forest training.

The scope of this master's thesis encompasses the design and implementation of strate-

gies to accelerate random forests by efficiently using CPUs and the massively parallel computing power of modern GPUs. We implement the visual features as proposed by Stückler et al. [50] that operate on color and depth information of RGB-D images but are computationally more complex than the features used by Toby Sharp [52] and Shotton et al. [43].

Our experimental results show that we have accomplished to accelerate random forest prediction to perform in real-time speed with less than 45 ms per image in VGA resolution on a mobile GPU and to accelerate training by a factor of up to 28. Random forest training which previously took about one whole day for one dataset now takes less than four minutes.

The remainder of this master's thesis is organized as follows: We discuss related work in Chapter 2. Chapter 3 introduces random forests as well as concepts and techniques that form the foundation of this work. In Chapter 4 we present our key contribution – the acceleration of random forests on CPU and GPU. We evaluate our implementation on two datasets and present the experimental results in Chapter 5. Chapter 6 concludes this master's thesis and outlines possible future work.

2. Related Work

Random forests are an ensemble classifier that became recently popular in the computer vision community. A series of work has been published that proposes random forests and variants for computer vision applications such as image segmentation or object detection.

Schroff et al. [41] use random forests with RGB, HOG and filter bank features for pixel-wise segmentation of images. Shotton et al. [43] use random forests in the Microsoft Kinect system to recognize the human pose from single depth images. The visual features for object-class image segmentation were inspired by Lepetit et al. [28] and calculate the average depth differences of two regions surrounding the pixel that is to be classified. For this master's thesis we implement and evaluate the visual features as proposed by Stückler et al. [50] that also use region differences but combine color and depth information in order to be applicable for RGB-D datasets.

Real-time applications as presented by Lepetit et al. [28] and Shotton et al. [43] require extremely fast prediction in few milliseconds per image. Random forests inherently allow a fast prediction, but require a long training phase when the dataset is large. Toby Sharp [52] implements real-time prediction for the Microsoft Kinect system on GPU which allows the dense labeling of 320 px × 240 px images [31] at 200 frames per second on the Microsoft Xbox 360 hardware [2]. Shotton et al. [43] use a distributed CPU implementation to reduce training time. Nevertheless, it takes one day to train three trees from one million images on a 1000 core cluster [43]. Toby Sharp [52] describes a method to implement training and prediction of random forests on GPUs which is a proprietary solution based on Microsoft Direct3D and the High Level Shader Language (HLSL). The features are chosen to target computer vision applications such as presented by Viola and Jones [54], Lepetit et al. [28] and Schroff et al. [41]. The implementation of Toby Sharp accelerates prediction on GPU by a factor of 100, while training has only a speed-up factor of eight. His measurements show that 96 % of training time is spent for histogram accumulation. Toby Sharp states that this might be a limitation of Direct3D and refers to experiments that indicate significant benefits by using CUDA, the GPU programming framework of NVIDIA.

Other work has been conducted to implement general purpose random forests on GPU that are not targeted to computer vision applications. Geary et al. [16] present an implementation of extremely randomized trees using CUDA but conclude that their method is inferior to an implementation on CPU.

Slat and Lapajne [48] also present a general purpose implementation of random forests that uses the CUDA framework. They achieve a maximal speed-up factor of five for an artificially chosen setup of 256 trees. They compare their implementation to the random forest CPU implementation in the data mining software Weka which has not been manually optimized and is written in Java.

Van Essen et al. [53] compare implementations for compact random forest classifiers on CPU, GPU and a Field Programmable Gate Array (FPGA). Compact random forests

are a variant of random forests that are more suitable for parallelization. However, Van Essen et al. focus on classification only and derive the training implementation from the LogitBoost classifier in Weka, which is also written in Java.

3. Foundations

This master's thesis focuses on the acceleration of random forests used for object-class image segmentation, which is a pixel-wise classification of images. In this section we cover the foundations of the accelerated machine learning algorithm, the features, notions, measures and last but not least the basics of the framework that we used to accelerate random forests on GPU.

Section 3.1 introduces the machine learning method "random forests". Section 3.2 describes the visual features that we use as input to the random forests. Sections 3.3 and 3.4 present the measures for segmentation accuracies and GPU-CPU speedup that we use in this master's thesis.

Training of random forests is a computationally intensive task, especially if complex features are used or the dataset contains many training instances. We accelerate random forests on GPU by using the programming toolkit CUDA of NVIDIA. Section 3.5 introduces the programming and memory model of the CUDA framework. Section 3.6 closes with notes on our strategy to use multi-dimensional arrays as abstraction layer in the CPU and GPU implementation.

3.1. Random Forests

Random forests, which are also known as random decision trees or random decision forests, were independently introduced by Ho [22] as well as Amit and Geman [3]. Breiman introduced the term "random forest" and compared the algorithm to Adaboost [5]. Breiman himself was influenced by the work of Amit and Geman as he states in his publication.

Random decision forests are ensemble classifiers that consist of one or many binary decision trees. Decision trees themselves are simple and commonly used models in data mining and machine learning. A decision tree consists of a hierarchy of questions that are used to map a multi-dimensional input value to a scalar output. The scalar output can be a real value (regression) or a class label (classification). In the scope of this master's thesis, we focus on decision trees and forests for classification.

Figure 3.1 depicts an example of a single decision tree that could be part of a random decision forest. To classify input x, we traverse each of the K random decision trees $\mathcal{T}_k$ of the random forest $\mathcal{F}$, starting at the root node. The root node of the decision tree is the topmost split node (also known as decision node), which is depicted as a circle. Each split node defines a test with a binary outcome (i.e. true or false). We traverse to the left child if the test is positive, otherwise we continue with the right child. Classification is finished if a leaf node $l_k(x)$ (depicted as square) is reached. The output value is obtained from the reached leaf node, which stores a single class label or a distribution $p\,(c \mid l_k(x))$ over class labels $c \in \mathcal{C}$.

The K decision trees in a random forest are trained independently. The output values or distributions for input x are collected from all reached leaves in the decision trees and

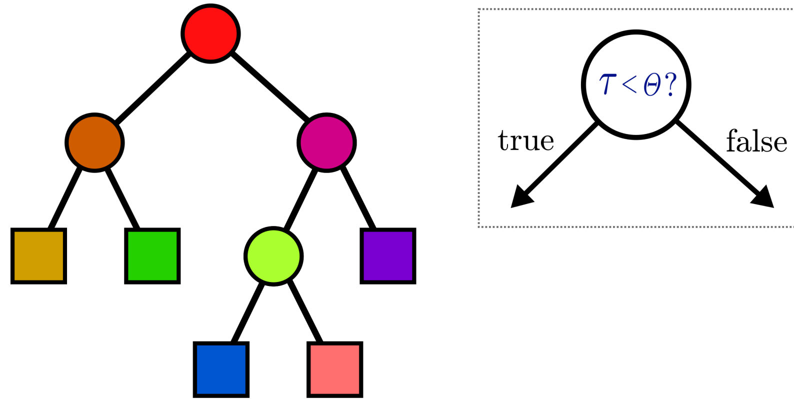

Figure 3.1.: Left: Example for a binary decision tree. Circles depict decision nodes. Squares depict leaf nodes. Right: Each decision node contains a test function in the form $\tau < \theta$ that compares a feature response τ with a threshold θ to generate a binary decision.

combined to generate a single classification. Various combination functions are possible. Common methods are the majority vote or the average of all probability distributions defined by

$$p(c \,|\, \mathcal{F}, x) = \frac{1}{K} \sum_{k=1}^{K} p\left(c \,|\, l_k\left(x\right)\right). \tag{3.1}$$

3.1.1. Random Forest Training

Key difference between a decision tree and a random decision tree is the training phase. The idea of random forests is to train multiple trees on a random subset of the dataset and a random subset of features.

A common process to train a decision tree is Top-Down Induction of Decision Trees (TDIDT), which is a greedy algorithm. TDIDT based decision tree training consists of two training phases:

1. Iteratively growing the decision tree until a stopping criterion is reached
   a) Selecting a leaf node that is not yet pure
   b) Selecting the best test that minimizes the impurity score
   c) Splitting the leaf node into left and right according to the selected test
   d) Continuing with a)
2. Prune the decision tree using a validation set

A decision tree is grown by iteratively splitting the nodes until the stopping criterion is reached. We do not split nodes with instances that all belong to only one class (i.e. the node is pure). This would not change the prediction given that we use majority voting or averaged probabilities as result combination method.

Decision trees allow an arbitrary function (i.e. feature) to be used as splitting criterion of a decision node. The function with the best parameterization is selected according to a score function. Two commonly used score functions are information gain and the Gini index [15, Section 9.2].

In contrast to normal decision trees, random decision trees are not pruned after training as they are less likely to overfit [5]. Breiman's random forests use Classification And Regression Tree (CART) as tree growing algorithm [5] and are restricted to binary trees (tree order of 2) for reasons of simplicity. Whether the decision tree is balanced depends on the dataset and the impurity score function used for training.

Tree growing finishes if a stopping criterion, such as a maximum tree depth, is reached. Additional per-node stopping criteria can be used, such as a minimum number of training instances (minimum support threshold). The chronological order of calculating node splits does not influence the decision tree structure. Thus, training can proceed in depth-first or breadth-first order as outlined in Algorithm 1 and Algorithm 2, respectively.

Algorithm 1 Depth-first training of a random decision tree

Require: $\mathcal{D}$ training instances
Require: F number of feature candidates to generate
Require: P number of feature parameters
Require: T number of candidate thresholds to generate
Require: stopping criterion (eg. maximal depth)

```
 1: D ← randomly sampled subset of 𝒟 (D ⊂ 𝒟)
 2: GrowTreeDepthFirst(D)          ▷ start growing the decision tree at the root note
 3: function GrowTreeDepthFirst(D)
 4:     if stopping criterion holds then return
 5:     (D_left, D_right) ← EvaluateBestSplit(D)
 6:     GrowTreeDepthFirst(D_left)                    ▷ recursively grow left child
 7:     GrowTreeDepthFirst(D_right)                   ▷ recursively grow right child
 8: function EvaluateBestSplit(D)
 9:     F ∈ R^{F×P} ← create random feature candidates
10:     T ∈ R^{F×T} ← create random threshold candidates for each feature
11:     I* ← ∞                                        ▷ initialize optimal impurity
12:     for all f ∈ 1..F do
13:         for all θ ∈ T_f do
14:             D_left ← {d | d ∈ D, FeatResp(F_f, d) < θ}           ▷ left split
15:             D_right ← D \ D_left                                  ▷ right split
16:             I ← ImpurityScore(D, D_left, D_right)
17:             if I < I* then                 ▷ update the best parameters f*, θ*
18:                 I* ← I; f* ← f; θ* ← θ
19:     D_left ← {d | d ∈ D, FeatResp(F_{f*}, d) < θ*}          ▷ left child samples
20:     D_right ← D \ D_left
21:     return (D_left, D_right)
```

We recommend Criminisi [7] for further reading in the topic of decision trees, random forests and its variants.

Algorithm 2 Breadth-first training of a random decision tree

Require: $\mathcal{D}$ training instances
Require: F number of feature candidates to generate
Require: P number of feature parameters
Require: T number of candidate thresholds to generate
Require: stopping criterion (eg. maximal depth)

1: $D \leftarrow$ randomly sampled subset of $\mathcal{D}$ ($D \subset \mathcal{D}$)
2: $N_{\text{root}} \leftarrow$ create root node
3: $C \leftarrow \{(N_{\text{root}}, D)\}$ ▷ initialize candidate nodes
4: **while** $C \neq \emptyset$ **do**
5: $\quad C' \leftarrow \emptyset$ ▷ initialize new set of candidate nodes
6: $\quad$ **for all** $(N, D) \in C$ **do**
7: $\quad\quad \left(D_{\text{left}}, D_{\text{right}}\right) \leftarrow \textsc{EvaluateBestSplit}(D)$
8: $\quad\quad$ **if** stopping criterion does not hold for (N, D_{left}) **then**
9: $\quad\quad\quad N_{\text{left}} \leftarrow$ create left child for node N
10: $\quad\quad\quad C' \leftarrow C' \cup \{(N_{\text{left}}, D_{\text{left}})\}$ ▷ add left child to candidates
11: $\quad\quad$ **if** stopping criterion does not hold for $\left(N, D_{\text{right}}\right)$ **then**
12: $\quad\quad\quad N_{\text{right}} \leftarrow$ create right child for node N
13: $\quad\quad\quad C' \leftarrow C' \cup \left\{\left(N_{\text{right}}, D_{\text{right}}\right)\right\}$ ▷ add right child to candidates
14: $\quad C \leftarrow C'$ ▷ continue with new set of nodes

The key part of the decision tree growing algorithm is the evaluation of the best split criterion. The best split evaluation can be abstracted into four major phase that are executed in sequential order.

1. Random feature candidate generation
2. Feature response calculation
3. Threshold selection
4. Impurity score calculation

We describe the four phases in the following section.

Random Feature Candidate Generation

Random forests have two sources of randomness. The first source is random sampling of the dataset (Line 1 of Algorithm 1). Secondly, a subset of features is sampled from the feature space to generate a set of candidate features that are evaluated to find the best split criterion. The size of the random subset is a training parameter and influences the correlation between individual trees in a forest. Increasing the size of the subsets leads to a higher correlation of the decision trees. If the subset includes all features, the decision trees have maximum correlation, i.e. the individual trees are identical. Usually this leads to a higher generalization error; on the other hand, a lower correlation of the individual trees requires training of more trees in order to keep the generalization error. Finding the best parameters with respect to the generalization error is an optimization problem which depends on the individual dataset. A common approach is to use cross-validation or out-of-bag estimates [5] to optimize parameters such as the size of the random feature subset or the number of trees to grow.

Feature Response Calculation

After we have selected a subset of random feature candidates, the best split evaluation continues to calculate feature responses for every candidate in the selected set of features (Line 14 of Algorithm 1).

Feature response calculation is not specified in detail, as it depends on the specific type of feature. We distinguish between two feature classes. The first variant extracts features from a dataset $\mathcal{D}$. Feature responses for every feature and every item in the dataset are pre-calculated in a pre-processing phase. The calculated feature responses constitute the derived dataset $\mathcal{D}'$, which is used as input data for training. An example would be a text corpus $\mathcal{D}$ that is pre-processed to extract one feature vector per document according to the bag-of-words model [21]. The set of bag-of-words vectors build the derived dataset $\mathcal{D}'$ where a feature response calculation is a simple lookup in the vector.

The second variant does not pre-calculate feature responses. Instead, the original dataset $\mathcal{D}$ is used to calculate the feature responses on-the-fly. This approach is typically used if the feature space is large, such that pre-calculating $\mathcal{D}'$ is infeasible and a sampling of the parameter space is necessary. An example is pixel-wise classification with a dataset that consists of images where the feature response depends on the neighborhood of a pixel. If size and shape of the neighborhood is a feature parameter, the pre-calculation of feature responses would imply a calculation for every pixel (in all images) times the number of

possible neighborhoods.

Another example are the visual features used by this master's thesis as described in Section 3.2. The feature space is extremely large, such that pre-calculation of feature responses is not feasible. Training a random forest from precomputed feature responses has different runtime properties, especially with respect to parallelization. Instead of pre-computing all values for all possible features, we sample the feature space at runtime and calculate the feature responses on demand.

Threshold Selection

Finding the best node split criterion requires the comparison of the feature response with a threshold (see the right hand side of Figure 3.1). It is possible to find the optimal threshold analytically. However, this is computationally infeasible for real-value feature responses and a large number of evaluated feature candidates. Instead, we follow the approach of Toby Sharp [52], Stückler et al. [50] and randomly sample the threshold space.

Impurity Score Calculation

Impurity score calculation in Line 16 of Algorithm 1 is not specified. It is possible to use any function that fulfills the requirements of an impurity score function, i.e. the output for input a is smaller than the output for input b, if and only if input a should be preferred over input b. The Gini coefficient (also known as Gini impurity) and information gain are commonly used functions.

Information gain is defined as the difference of Shannon entropy of the class distribution D in the parent node and the weighted sum of Shannon entropies in class distributions in the left child node D_{left} and right child node D_{right} over the set of classes $\mathcal{C}$ by

$$I_{\mathcal{C}}\left(D, D_{\text{left}}, D_{\text{right}}\right) := H_{\mathcal{C}}(D) - \frac{|D_{\text{left}}|\, H_{\mathcal{C}}\left(D_{\text{left}}\right) + \left|D_{\text{right}}\right| H_{\mathcal{C}}\left(D_{\text{right}}\right)}{|D|}, \tag{3.2}$$

where

$$H_{\mathcal{C}}\left(D\right) := -\sum_{c \in \mathcal{C}} p(c \mid D) \log_2\left(p\left(c \mid D\right)\right). \tag{3.3}$$

Wehenkel and Pavella [58] define the normalized information gain by Equation (3.4), which is a variant of the information gain that is normalized by the sum of the split entropy[1] and classification entropy $H_{\mathcal{C}}\left(D\right)$ [17].

$$I'_{\mathcal{C}}\left(D, D_{\text{left}}, D_{\text{right}}\right) = \frac{2 I_{\mathcal{C}}\left(D, D_{\text{left}}, D_{\text{right}}\right)}{H_{\mathcal{C}}\left(D\right) + H_{\mathcal{C}}\left(D_{\text{left}}\right) + H_{\mathcal{C}}\left(D_{\text{right}}\right)} \tag{3.4}$$

It can be shown that the normalized information gain holds the property of Equation (3.5) [58].

$$0 \leq I'_{\mathcal{C}}\left(D, D_{\text{left}}, D_{\text{right}}\right) \leq 1 \tag{3.5}$$

[1] $H_{\mathcal{C}}\left(D_{\text{left}}\right) + H_{\mathcal{C}}\left(D_{\text{right}}\right)$

The normalized information gain favors tests with a lower split entropy. Tests that split into a large and a small child have a higher normalized information gain than tests that split into two equally sized children, given that the tests yield the same information gain.

3.1.2. Random Forest Prediction

A random forest $\mathcal{F}$ classifies an input instance x of the dataset by processing it on each random decision tree $\mathcal{T}_k$ in the forest and combining the individual results. We start at the root node of each tree and continue traversing the decision trees until we reach a leaf node. The type of information which is stored in a leaf node depends on the respective application. The algorithm that is used to combine the information of all leaf nodes also depends on the application. Random forests used for our image segmentation application store probability distribution over all classes in the leaf nodes [50, 43].

Each decision tree stores the number of samples per class $s_c(l_k)$ that reached each leaf node l_k during training. The class distribution $p\,(c\,|\,l_k(x))$ for input x is then calculated as the relative frequency of training samples by

$$p\,(c\,|\,l_k(x)) := \frac{s_c\,(l_k(x))}{\sum_{c'\in\mathcal{C}} s_{c'}\,(l_k(x))}. \tag{3.6}$$

In case that the training samples Q of the training dataset $\mathcal{D}$ with $Q \subseteq \mathcal{D}$ are sampled uniformly across all classes $\mathcal{C}$ such that $p(c\,|\,Q) = |\mathcal{C}|^{-1}$, we weigh the class distribution stored in a leaf node according to the probability of class c in the original dataset $\mathcal{D}$ by

$$p\,(c\,|\,l_k(x)) := \frac{s_c\,(l_k(x))}{\sum_{c'\in\mathcal{C}} s_{c'}\,(l_k(x))}\frac{p(c\,|\,\mathcal{D})}{p(c\,|\,Q)}. \tag{3.7}$$

We use $p(c\,|\,\mathcal{F}, x)$, which is the averaged class distribution over all reached leaves in the forest $\mathcal{F}$ according to Equation (3.1), and either return the class with the highest probability

$$c^* = \arg\max_{c\in\mathcal{C}} p(c\,|\,\mathcal{F}, x), \tag{3.8}$$

or the distribution $p(c\,|\,\mathcal{F}, x)$ itself and leave it open to the client how to use the information.

In the case of object-class image segmentation, every pixel of the image represents an instance that can be classified independently. This property leads to a so called “embarrassingly parallel” problem which is easy to parallelize on CPU and GPU.

3.1.3. Random Forest Variants

In this section we briefly present two variants of random forests. Both variants are specializations that add further restrictions to the properties of the decision trees in a random forests.

Extremely Randomized Decision Forests

Extremely randomized decision forests have a restriction for the training of random forests. During training, we randomly sample a set of feature candidates that are evaluated for the best split criterion (cf. Section 3.1.1).

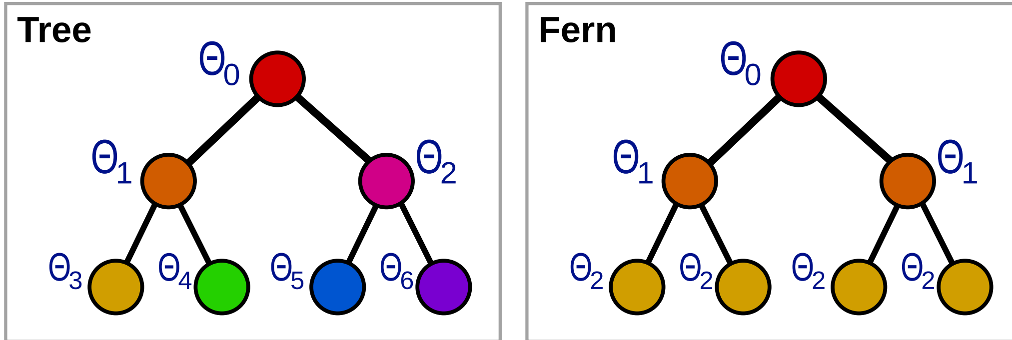

Figure 3.2.: Random tree (left) and random fern (right) (based on [7]). Random trees have different tests θ_n for every node n. Random ferns share the same tests θ_l per tree level l.

Geurts et al. [17] coined the term "extremely randomized trees" for a variant with the restriction that the training algorithm samples only a single feature candidate per node.

For prediction, we do not have to distinguish between trees of an extremely randomized decision forest and trees of a normal random forest.

Training can be simplified dramatically, if threshold sampling is omitted and the threshold value is randomly selected. Feature response calculation and impurity score calculation can be skipped since the only feature and threshold candidate will be selected anyway. However, for this master's thesis we do not distinguish between extremely randomized forests and normal random forests. Nevertheless, we can train extremely randomized forests by setting the two parameters for the number of feature candidates and threshold candidates to one.

Random Ferns

Ozuysal et al. [36] proposed a second random forest variant called "random ferns". Random ferns have the restriction that all nodes in the same tree level use the same split criterion (i.e. test parameters).

Figure 3.2 depicts an example of a random tree and a random fern. We see that a random tree has different tests on the first level (θ_1, θ_2), while the random fern applies the same test θ_1. The same principle applies for the other levels as well.

The properties of a random fern can be leveraged to improve parallelization since all instances are classified by applying the same list of tests [36, 35]. The list of tests is known in advance. In contrast to the test sequence in a random forests, the next element in the sequence does not depend on its predecessor. Thus, all feature responses can be calculated in parallel.

The original random fern training method differs from the training of a random forest. The most important distinction is the selection of the best split criterion which is done randomly as for extremely randomized decision trees. Furthermore, Ozuysal et al. developed random ferns for the application of keypoint recognition with dozens of ferns and hundreds of classes which is about one order of magnitude larger than the typical parameter settings that we observe for the applications in the scope of this master's thesis.

We base our random fern training method on random forest training such that decisions trees are constructed with the shape and the constraints of random ferns. Hence, our random fern training does not differ significantly from random forest training. Instead of selecting the test criterion which maximizes the impurity score for each node, we select the test criterion which maximizes the “common good” for all nodes in the same tree level. This implies that random ferns have to be trained in breadth-first order while random forests can also be trained in depth-first order. Furthermore, we have to evaluate the same set of feature candidates when growing the fern by one level.

Criminisi [7] claims that random ferns typically require deeper trees to achieve a comparable segmentation accuracy of random forests but have a lower risk of overfitting to the training data.

3.2. Visual Features for Random Forests

Tests in decision nodes are not restricted to a specific feature type and each node can differ with respect to the parameters. Despite the name “random forest”, the tests are usually deterministic. The only requirement is a boolean return value which is used to traverse the decision tree during training and prediction.

This master’s thesis focuses on image segmentation and classification applications as published by Schroff et al. [41], Shotton et al. [43], Stückler et al. [50]. The selection of features is inspired by Lepetit et al. [28, Section 5.3] and leads back to the theory of visual object detection published by Viola and Jones [54].

We implement two types of RGB-D image features as proposed by Stückler et al. [50, Section III.B.]. They resemble the features in [52, 43] but use depth-normalization and region averages; Shotton et al. [43] avoid the use of region averages to reduce the computational complexity.

For a given query pixel q, the image feature f_θ is calculated as the difference of the average value of the image channel ϕ_i in two rectangular regions R_1, R_2 in the neighborhood around q by

$$f_\theta(q) := \frac{1}{|R_1(q)|} \sum_{p \in R_1} \phi_1(q) - \frac{1}{|R_2(q)|} \sum_{p \in R_2} \phi_2(q). \tag{3.9}$$

Extent size w_i, h_i and relative offset o_i of the rectangular region $R_i, i \in \{1, 2\}$ in the image is normalized by the depth of the query pixel $d(q)$ such that

$$R_i(q) := R\left(q + \frac{o_i}{d(q)}, \frac{w_i}{d(q)}, \frac{h_i}{d(q)}\right). \tag{3.10}$$

This leads to the property of using smaller regions and offsets for pixels that have a larger distance to the camera (eg. pixels in the background). Figure 3.3 depicts an example of an image feature at three different query pixels in an image.

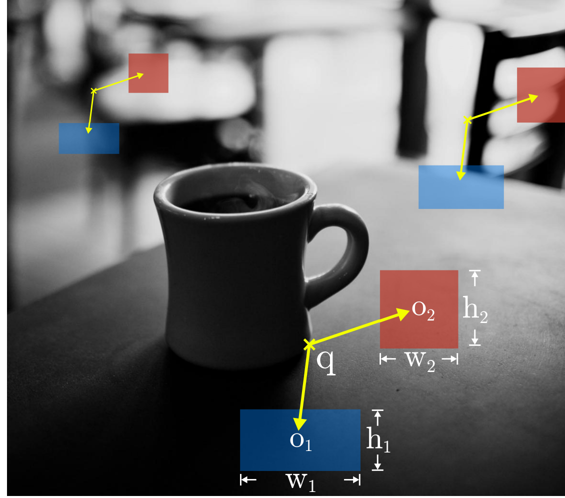

Figure 3.3.: Example of a visual feature at three different query pixels. The feature response value for a query pixel q is calculated from the difference of average values in two offset regions. The relative offset locations o_i and region extents w_i, h_i are normalized with the depth $d(q)$ at the query pixel.

3.2.1. Parameters

Each visual feature f_θ has a set of parameters θ that consists of eleven scalar values

1. Feature type (either "color" or "depth"),
2. Region 1 offset x,
3. Region 1 offset y,
4. Region 1 extent width,
5. Region 1 extent height,
6. Region 2 offset x,
7. Region 2 offset y,
8. Region 2 extent width,
9. Region 2 extent height,
10. Channel 1,
11. Channel 2,

which are randomly selected during training.

The feature type parameter specifies which of the two visual features is used. The first feature type ("color") operates only on the values of the three color channels, while the second feature type ("depth") only uses information of the depth channel. Both feature types share the remaining set of feature parameters that describe the offset and region extent of both queried image regions (cf. Figure 3.3). The following two sections describe both feature types in more detail.

3.2.2. Color Feature

The three color channels of a RGB-D input image are converted into CIE Lab color space in a pre-processing step. Each of the two region averages is calculated on one image channel,

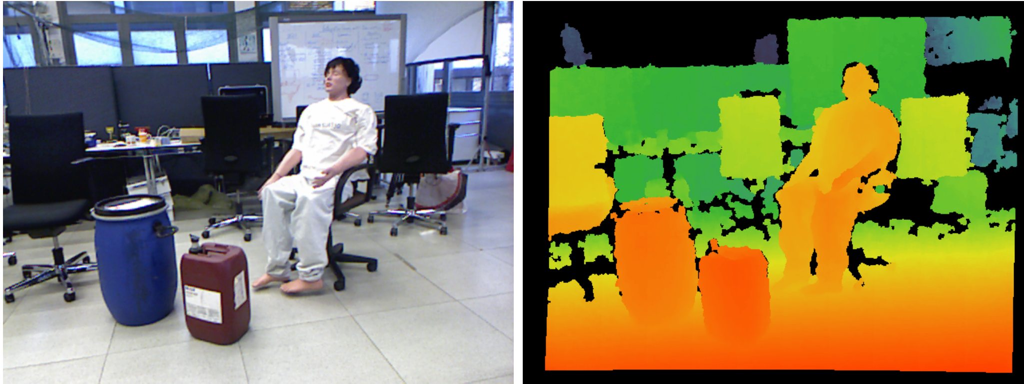

Figure 3.4.: Example of a RGB-D image captured by a Microsoft Kinect camera. Left: Color channels of the image. Right: Visualization of the depth channel. Different colors indicate various depths, where red and orange indicate a low distance to the camera. Green and blue indicate higher distance. Black is used to visualize missing depth information.

which is specified by parameter channel 1 and channel 2, respectively. Note that the two regions do not need to be calculated on the same image channel. In fact, comparisons on the same image channels are less frequently selected during training than comparisons on two different image channels as shown by a frequency analysis on a typical dataset depicted in Figure A.1 in Appendix A.

Random forest implementations need not be aware of semantic interpretation of a color image channel. The feature can easily be extended to use more than three image channels, such as an additional channel that contains the output of a Sobel or Canny edge detector.

3.2.3. Depth Feature

Depth information of a RGB-D input image could be treated as fourth channel in the image. The advantage of such an approach would be the usage of a generalized feature implementation that calculates the difference of two image regions without a conditional branch on the feature type. This approach is used by Toby Sharp [52]. However, depth information of RGB-D cameras, such as the Microsoft Kinect depth sensor, do not guarantee to deliver defined values for every pixel in the image. This can happen, for instance, if parts of the scenery are only visible for one of the two cameras. The distance to these regions cannot be measured and thus remains undefined. Undefined values are usually encoded as Not a Number (NaN) or zero depth. Figure 3.4 depicts an example of a RGB-D image with partially missing depth information.

When calculating the depth region average, we need to take care of undefined depth values. The approach used by Stückler et al. [50] the generation of a second depth image channel that is used to read the number of valid depth values in a region.

Parameter channel 1 and channel 2 are unused and always set to zero for depth features.

3.2.4. Integral Images

The color and depth features as presented in Sections 3.2.2 and 3.2.3 require the calculation of the region sum of an image channel. A naïve implementation requires $n \cdot m$ memory

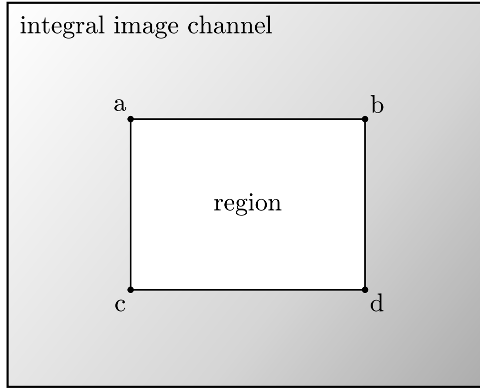

Figure 3.5.: Example of a region sum calculation that uses an integral image. The sum of a rectangular region (center) can be calculated by four memory accesses and three arithmetic operations: Region sum $= d - c - b + a$.

transactions and $n \cdot m$ arithmetic operations to calculate the sum of a $n \times m$ region.

The training of random forests requires to calculate many different region averages in the same image channel. We adopt the idea of Viola and Jones [54] and calculate integral images (closely related to summed area tables [8]) in a pre-processing step while loading the images. This reduces the computational complexity of region sum calculation to four memory transactions and three arithmetic operations as illustrated in Figure 3.5.

Numerical Limitations

The integration of images can result in large values. Consider the case of a depth channel with an average depth of 2.0 m. The largest value in an integrated 640×480 pixel image would be approximately $640 \cdot 480 \cdot 2\,\mathrm{m} = 614\,400\,\mathrm{m}$, which requires at least 19 bit for representation. Single precision floating points (IEEE 754) have a total precision of 24 bit. Given that the place before the decimal point already requires at least 19 bit, there are at most 5 bit left to represent the value after the decimal point. Depth precision is limited to at most $2^{-5}\mathrm{m} = 0.03125\mathrm{m}$ in this case. If the region is small and the depth difference between two corners of the rectangular falls below that value, the calculated region sum is rounded to 0 and the depth information is lost. Hence, we avoid the use of 32 bit floating points for integral depth channels. Instead, we use 32 bit unsigned integers to represent fixed point values with a granularity of 1 mm. Integer overflows for 640×480 pixel depth channels occur at an average depth of at least $\frac{2^{32}}{640 \cdot 480}\mathrm{mm} \approx 14\,\mathrm{m}$. For our experiments in 8 bit RGB color space, we use integer numbers for the integral image. In case of the three channels in CIE Lab color space, we measure that storing the values of the integral images in single precision floating point leads to a higher segmentation accuracy than using fix-point values with limited precision. Our implementation performs image integration in double precision with the Kahan summation algorithm [25] to reduce the propagated error while calculating the sums.

Note that the memory requirements to store the integrated color channels is four times as large since we have to use 32 bit instead of 8 bit RGB values.

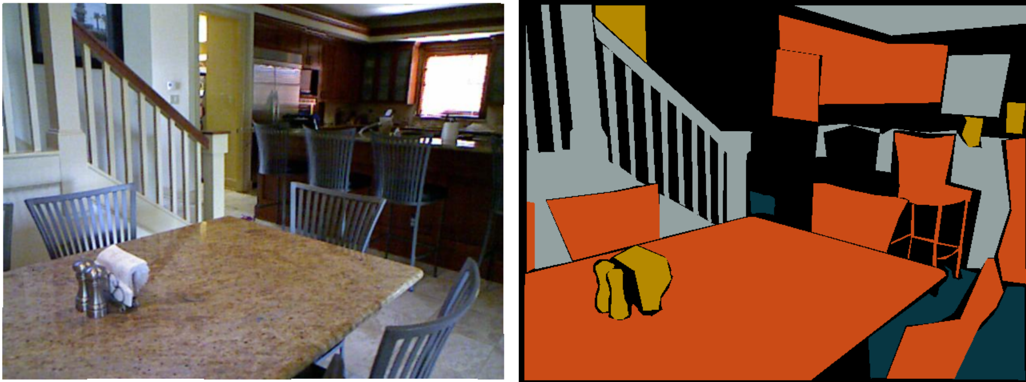

Figure 3.6.: Example of an image and the according ground truth from the NYU Depth v2 dataset. Left: RGB color image. Right: Manually labeled ground truth. Different colors are used to label four classes. Black pixels indicate "void".

3.3. Measuring Image Segmentation Accuracy

Besides the focus of this master's thesis on acceleration of random forests, we are also interested in the segmentation accuracy of random forests. For all of our experiments we output the confusion matrix, average class accuracy and pixel accuracy. The following sections explain the special classes "void" and "background" as well as how average class and pixel accuracy is measured.

3.3.1. Void and Background Pixels

Depending on the dataset, the label images ("ground truth") can contain "background" or "void" pixels [12]. "Void" pixels indicate that the actual class of the pixel is unknown and can be either background or any other label. "Void" pixels can be used to mask regions of an image that are difficult to classify and should not influence the accuracy measure. Two common cases are the border pixels between two objects and pixels that humans cannot classify or would be too difficult for the learning algorithm. Figure 3.6 shows an example of a RGB image and the according ground truth. Black pixels in the ground truth indicate "void" and are excluded in training and testing.

Datasets with small foreground objects and large background areas tend to yield high accuracies even if the classifier has a bias on the background class. Naïve classifiers which always select the background class would, for example, yield an accuracy of 90 % for images with 90 % of background.

We calculate three variants for each accuracy measure:

1. Treating "background" and "void" as usual classes
2. Excluding background pixels
3. Excluding "void" pixels

The following two sections explain how we measure class and pixel accuracy with respect to the three variants that include or exclude "void" and "background" pixels.

3.3.2. Average Per-Class Accuracy Measure

Per-class accuracy is the percentage of correctly labeled pixels for a given class. Average per-class accuracy is equivalent to the diagonal average in a confusion matrix which has been normalized, such that all values of one label add up to 1.

The first variant of the average per-class accuracy in which "background" and "void" are treated as usual classes, is invariant to the total number of pixels in each class. Hence, large classes are not favored over small classes. However, a classifier that always selects a specific class would have an accuracy of 100 % for that class.

The second variant excludes background pixels. The per-class average in this case is calculated using the accuracy of all classes except the background class.

The third variant excludes "void" pixels. The actual prediction for a pixel that is marked as "void" in the ground truth is disregarded and does not influence the accuracy measure.

3.3.3. Pixel Accuracy Measure

The pixel accuracy measure is defined by the number of all correctly classified pixels over the total number of pixels in the image. Hence, large classes (such as large background areas) have more impact in the pixel accuracy measure.

The second variant excludes background pixels and is calculated as the ratio of all correctly classified pixels over all pixels that are either non-background or correctly classified. This variant has been used for the dataset of Stückler et al. [50] that defines specific semantics for background pixels.

The third variant excludes "void" pixels and is calculated as the ratio of all correctly classified non-void pixels over all non-void pixels.

3.4. Speed-up Measures

Toby Sharp [52] compares the CPU and GPU implementation by comparing the elapsed time spent in training and testing on CPU and GPU, respectively. We think that the consideration of only the simple speed-up factor can lead to misinterpretation, especially if a powerful GPU is compared to a significantly older CPU generation. CPUs optimized for many floating point operations per second that accept, for instance a higher power consumption, can also bias the interpretation of the GPU–CPU speed-up factor.

On the other hand, a GPU might significantly reduce the required processing time of a CPU while being more expensive in acquisition or during runtime due to a higher power consumption. Depending on the application, it might be less expensive to acquire additional CPUs rather than one GPU. Thus, we extend our results with two additional measures,

1. Speed-up factor normalized over acquisition costs,
2. Speed-up factor normalized over power consumption,

as proposed by Van Essen et al. [53].

3.5. The CUDA GPU Programming Framework

Our GPU implementation is based on the CUDA programming toolkit in version 5.5.

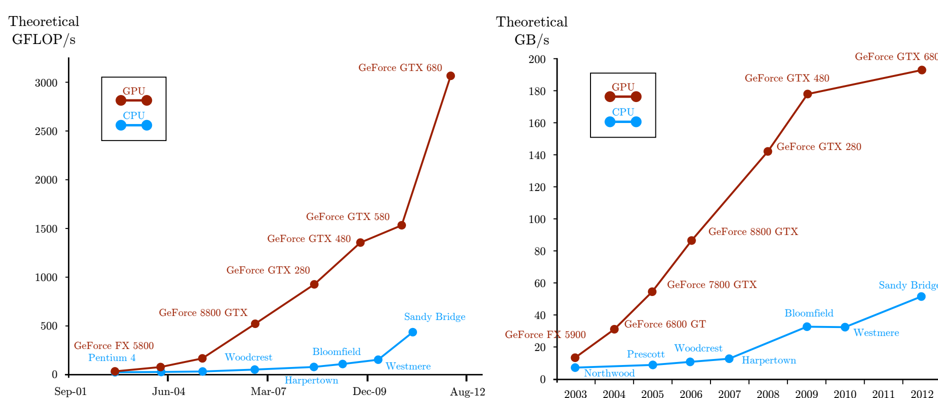

Figure 3.7.: Comparison of computing throughput of various CPUs and GPUs [37]. Left: Theoretical floating point operations per second. Right: Maximal memory throughput in gigabytes per second.

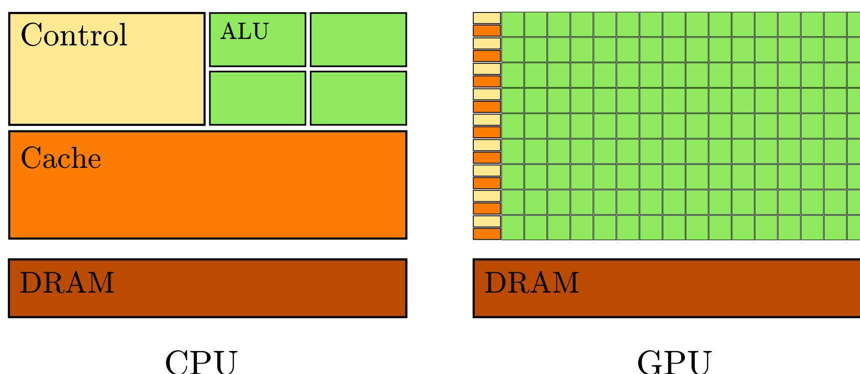

Figure 3.8.: Schematic diagram of a CPU (left) and GPU (right) [37]. More space on a GPU is devoted to ALUs. CPUs contain larger areas for caches and control logic, such as branch prediction.

GPUs and CPUs have different strengths and weaknesses. GPUs tend to be more efficient for computationally intensive applications, that can be parallelized by hundreds or thousands of independent threads. The gap between computing power and memory bandwidth, as depicted in Figure 3.7, is due to the fact that relatively more space (i.e. transistors) on a GPU is devoted to integer and floating-point units than on a CPU [37] as can be seen on the schematic high level view of a CPU and GPU on Figure 3.8.

3.5.1. GPU Programming Model

The CUDA toolkit can be used as a `C`, `C++` or `Fortran` language extension. There are wrappers for other programming languages such as `Python` and `Java` or the Microsoft `.Net` platform.

NVIDIA uses the notions "device" and "host" to distinguish between GPU and CPU, respectively. The CUDA enabled C/C++ compiler `nvcc` is used to compile to the Parallel Thread eXecution (PTX) instruction set [57]. PTX instructions describe a low-level virtual

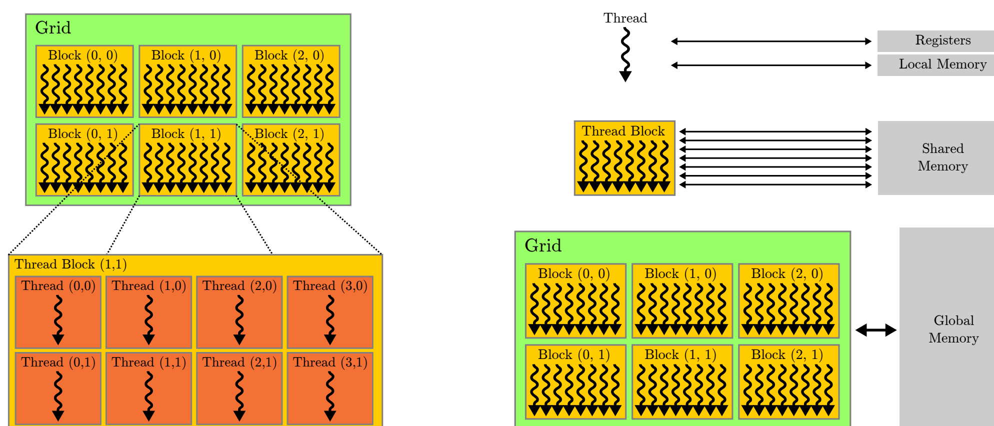

Figure 3.9.: Left: Schematic diagram of threads, thread blocks and the grid [37]. The two-dimensional grid in the depicted example consists of 2×3 thread blocks. Each block consists of 3×4 threads. Right: Schematic diagram of the CUDA memory hierarchy. Global memory is used to share data across several blocks. Threads in one block can share memory in the per-block shared memory space. Data in shared memory space is not persistent across several kernel launches. Threads have access to their own registers and local memory. Per-thread local memory and registers are neither persistent nor accessible by other threads.

machine that is independent of the exact GPU hardware. The CUDA driver translates PTX instructions into GPU specific code when the programming is loaded at runtime.

The CUDA programming model was designed so that a programmer does not need to be aware of specific hardware details. A programmer implements so called “kernel” functions, that can be executed in parallel by many threads on the GPU. Dedicated scheduling hardware on the GPU takes care of distributing the threads to the available computing units. Multiple threads execute the same code but act on different data. NVIDIA uses the term Single Instruction, Multiple Threads (SIMT) akin to the Single Instruction, Multiple Data (SIMD) class in Flynn’s taxonomy [14]. Multiple threads are logically grouped into one-, two- or three-dimensional blocks. Several blocks are grouped into a one-, two- or three-dimensional grid as depicted in the left hand side of Figure 3.9.

GPUs consist of one or many streaming multiprocessors. Each streaming multiprocessor of the Fermi architecture contains its own scheduling units, special function units, control units, registers and caches [57].

A streaming multiprocessor has several CUDA cores that perform the actual computations. The Fermi architecture defines 32 cores per streaming multiprocessor which results in a peak performance of 32 floating point and 32 integer operations per cycle and per streaming multiprocessor for devices of compute capability 2.0.

The scheduling unit assigns units of 32 threads, called *warps*, at a given point in time. All threads in a warp are executed in parallel, if the execution path does not diverge.

3.5.2. GPU Memory

The right hand side of Figure 3.9 depicts the three-level memory hierarchy and its relation to threads, blocks and grids.

Registers and Local Memory Space

Every thread has access to local memory and registers that are not persistent and cannot be read by other threads. The compiler decides whether a variable is put into register or local memory. The latter case can happen since the number of registers is a limited resource. Local memory resides in global device memory space with the same latency and bandwidth characteristics. This means that latency and bandwidth of registers and local memory differ by two orders of magnitude if data is not cached. However, reads and writes are cached in L1 cache such that the use of local memory does not always entail a runtime performance penalty [30].

Shared Memory Space

Threads in one block have access to the per-block shared memory. Access to shared memory is not as fast as to local memory, but significantly faster than access to global memory. Shared memory consists of 16 or 32 banks for devices of compute capability 1.x and compute capability 2.x/3.x, respectively. Each bank has a bandwidth of 32 bit per clock cycle for devices of compute capability 1.x/2.x and 64 bit per clock cycles for devices of compute capability 3.x [37]. The programmer has to manually synchronize access to shared memory by using provided methods in the CUDA framework. Data in thread-local and shared memory is not persistent after threads terminate. The programmer can use shared memory to share data between threads in the same block. Shared memory can be used, for instance, to implement software-managed caches [47] or to rearrange data, such that the data can be written more efficiently to global memory [39].

Global Memory Space

Global memory is the slowest memory space. Data in global memory is persistent, until manually released by the programmer or the CUDA driver at program termination.

Access to global memory space has to be manually synchronized by the programmer.

CUDA offers atomic operations for 32-bit or 64-bit integers in global and shared memory. However, not all atomic operations are available for older devices before compute capability 2.x.

Texture Memory Space

One-, two- or three dimensional arrays in global memory can be mapped to textures. Using texture memory has potential benefits over accessing global memory directly. Dedicated hardware on the device is used to cache data of texture space and perform address calculations. Higher bandwidth can be achieved, even if memory access is not strictly coalesced. Texture memory and texture cache perform best if the access pattern shows two-dimensional locality. Texture cache misses cause a global memory access. However, the global memory access is cached by the L2 cache. Accordingly, using texture memory does not reduce latency for an uncached memory access [37].

Constant Memory Space

A small part, 64 kB for devices of compute capability 1.x–3.x, of global device memory is reserved for constant memory. Data in constant memory is read-only. Streaming multi-

processors contain a dedicated cache for constant memory, which is read-only and shared by all functional units.

Unified Address Space

Global memory, shared memory and thread-local memory are addressable in the same memory space since PTX version 2.0. Hence, the programmer need not know the memory space of the variables.

Transferring Data between Host and Device Memory

Data in device memory space cannot be accessed by host code running on the CPU. Vice versa, device code cannot access host memory that was allocated with `malloc()` or `cudaMalloc()`. The programmer has to copy data between host and device memory space explicitly. However, since CUDA 4.0 it is possible to allocate host memory with `cudaMallocHost()` or `cudaHostMalloc()`. In contrast to `malloc()` or `cudaMalloc`, it allocates page-locked memory (also known as “pinned memory”) that can directly be accessed from device code.

Streams

Recent hardware generations allow the programmer to transfer data from host to device, device to host or device to device asynchronously. The programmer creates multiple streams and passes it as argument when launching kernels and memory transfers. If no stream is specified as argument, the default stream is used and actions are executed sequentially. Depending on the hardware, one or many memory transfers can be executed while a kernel is running. If a kernel launch and a memory transfer happen to be in different streams, they can be scheduled by the hardware to run in parallel. The programmer has to synchronize manually and has to be aware of exact semantics [37].

Hiding Memory Latency

Latency of uncached global memory accesses is at least one magnitude larger than the execution of arithmetic operations.

The latency of an global memory access on the Fermi GPUs amounts between 400 and 800 clock cycles, while an arithmetic operation takes approximately 20 clock cycles.

The memory latency penalties for CPUs are similar. We use `mem-latency` [49] to measure the CPU memory latency which is 90 ns ($\approx$ 225 clock cycles) on an Intel Core i7–920 with memory clocked at 1066 MHz.

Hiding that latency can be crucial to use all of the available processing power of CPUs and GPUs. Modern CPUs are equipped with a large L1 cache and L2 cache that transparently helps to mitigate the performance penalties. Nevertheless, manually optimizing for CPU cache hit rate is still important for memory-bound algorithms [11].

Hiding memory latency is possible, if other operations can be executed while an operations stalls for memory access. The following notions and principles are necessary to understand to be able to efficiently optimize memory throughput of GPU code.

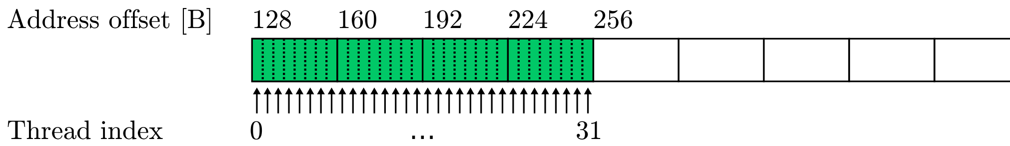

Figure 3.10.: Example of efficient global memory access (best-case). Each of the 32 threads request 4 bytes of memory (green, dotted rectangles). Addresses are consecutive such that the memory transfer can be coalesced by the GPU. Only one transfer of 256 B is necessary (green area).

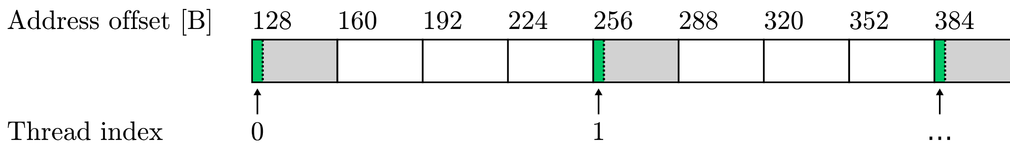

Figure 3.11.: Example of inefficient global memory access (worst-case). Each thread requests 4 bytes of memory (green, dotted rectangles). Addresses are *not* consecutive, so memory transfer cannot be coalesced. Every thread causes a separate memory transfer (gray areas) of the smallest possible word (32 B in the example).

Efficient Global Memory Access

Global memory has a limited bandwidth. The bandwidth is only fully saturated if several bytes, forming a word, are transferred at once in one transaction. Memory throughput is wasted if not all bytes of a transferred bytes are payload, for instance if only single bytes are transferred.

To make full use of all available bandwidth, i.e. reduce the number of memory transactions, the programmer is required to design the kernels in such a way that global memory accesses follow a certain pattern. Dedicated GPU hardware detects if several threads in a warp access memory at consecutive addresses. Instead of inhibiting a separate memory transfer for every thread, the requested addresses are coalesced and data is transferred in the largest possible word (128 B in the example).

Figure 3.10 shows an example for memory accesses that can be coalesced into a single transfer of 256 B by the GPU. Figure 3.11, in contrast, shows the worst-case scenario for memory accesses by multiple threads. Coalescing is not possible and a total of 32 transfers with the smallest possible word (32 B in the example) are necessary.

Avoiding Shared Memory Bank Conflicts

Addresses of shared memory access do *not* need to be contiguous in order to achieve maximal throughput [39]. However, the access pattern has to be designed to avoid bank conflicts in order to maximize the throughput.

The GPU transparently maps addresses in shared memory space to the rows in the shared memory banks. The word size (i.e. row size) of shared memory banks depends on the hardware platform and is either 32 bits for compute capability 1.x/2.x or 64 bit for

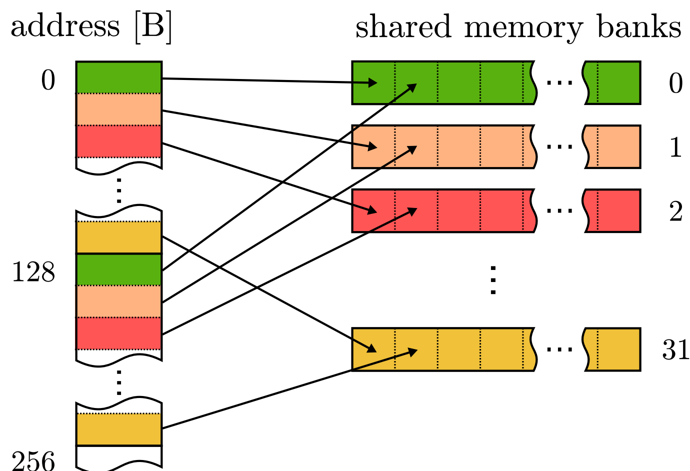

Figure 3.12.: Example of one possible mapping from shared memory address space to the 32 shared memory banks. Successive 32 bit words are mapped to successive banks as indicated by the coloring. The words at address 0 and 128 map to the same bank but in different rows.

compute capability 3.x.

Banks are organized such that successive words (either 32 bit or 64 bit) are mapped to successive banks. Figure 3.12 shows an example for the mapping with 32 banks and 32 bit words.

Each bank can read or store one word per two cycles. Hence, the maximum total shared memory bandwidth can be only achieved if all banks are used in parallel. If two or more threads read data from the same address (i.e. the same bank), the value is broadcast to the threads. If two or more threads write data to the same address, the value of one thread is written. The winning thread is undefined.

If two or more threads access the same bank for reading or writing, but at different rows (i.e. different addresses), a bank conflict occurs. The GPU has to serialize the memory transactions which results in loss of memory performance.

A common case of shared memory bank conflicts is column-wise access of a matrix by multiple threads. Ruetsch and Micikevicius [39] elaborate on a matrix transpose example that requires such accesses and discuss strategies how to avoid bank conflicts.

CUDA enabled GPUs contain a hardware event counter for bank conflicts. Programmers can use the profiler of NVIDIA [10] to retrieve the number of bank conflicts that happened during the execution of a kernel.

Partition Camping of Global Memory

Global memory is distributed over several chips, called partitions. Partitions in global memory are comparable to banks of shared memory. A similar effect such as bank conflicts can be observed when load and store operations to global memory are not equally distributed but happen to occur only on one or a few partitions. Aji et al. [1], Ruetsch and Micikevicius [39] call this effect “partition camping” and measure up to seven-fold speed-ups by mitigating partition camping in certain use cases.

Kernel Optimization Variant	Effective Memory Throughput [GB/s]		
	GTX 280	GTX 480	Tesla K20c
Simple Copy	120	136	139
Shared Memory Copy	85	209	267
Naïve Transpose	3	26	50
Coalesced Transpose	23	64	98
Bank Conflict Free Transpose	**23**	**140**	**129**
Coarse-grained Transpose	23	144	139
Fine-grained Transpose	98	138	129
Diagonal	**78**	**133**	**124**

Table 3.1.: Effective memory bandwidth of the transpose matrix sample application shipped with the CUDA SDK. Compiled with CUDA 5.0 and measured on a GeForce GTX 280, GeForce GTX 480 and Tesla K20c for comparison. Matrix size: 2048×2048 (64×64 tiles), tile size: 32×32, block size: 32×8. We set the number of repetitions to 5000 in order to achieve stable results. The "Diagonal" variant is optimized to avoid Partition Camping, while the "Bank Conflict Free Transpose" is only optimized to avoid Bank Conflicts [39]. The results show that partition camping avoidance is important on a GTX 280 but does not improve bandwidth on the newer GPUs GTX 480 and Tesla K20c.

To the best of my knowledge, there is no official NVIDIA documentation available that describes partition camping and how to avoid it. We run the transpose matrix sample application (cf. [39]), shipped with the CUDA 5 Software Development Kit (SDK), on a GeForce GTX 280, GeForce GTX 480 and Tesla K20c. Table 3.1 shows that optimizing for partition camping (kernel variant "diagonal") does not improve the effective memory throughput on the GTX 480 and Tesla K20c. In fact, we measure a small slowdown in comparison to the "bank conflict free transpose" variant that is optimized to avoid bank conflicts but does not attempt to avoid partition camping. According to rumors [26], GPUs of the Fermi architecture mitigate "partition camping" by hashing global memory addresses.

Occupancy

Occupancy is defined as the ratio of active warps over the maximum number of warps supported on a streaming multiprocessor. The maximum number of supported warps depends on the hardware platform and can be looked up in the *CUDA C Programming Guide* [9, Table 9. Feature Support per Compute Capability].

The maximum number of active warps depends on the kernel resource consumption. Limited resources are shared memory and the number of registers. The CUDA compiler attempts to minimize register consumption to increase the maximum number of active threads. The CUDA Software Development Kit is shipped with an occupancy calculator that programmers can use to calculate the theoretical occupancies.

Note that higher occupancy does not guarantee higher performance. Compute-bound algorithms do not benefit from an increased occupancy if all cores are already busy at any

given point in time.

Increasing the occupancy can have a negative impact. This can happen if the number of instructions increases or more memory transactions are necessary due to register spill. Volkov [55] measured a 60 % performance increase for the multiplication of two large matrices in the CUBLAS library, while occupancy decreased from 67 % to 33 %.

Thread-Level Parallelism (TLP)

Thread-level parallelism hides memory latency by scheduling many threads in parallel. If an instruction of a thread is waiting for a memory transaction to complete, instructions of another thread are executed in the meantime. Thread-level parallelism requires more threads to be ready for execution than currently active threads. The maximum number of resident threads is limited by hardware resources and can be looked up in the *CUDA C Programming Guide* [9, Table 9. Feature Support per compute capability].

Instruction-Level Parallelism (ILP)

Parallelism can be increased by reordering instructions such that operations that depend on each other are interleaved with independent operations.

Listing 3.1 shows an example with instructions that are not optimized to improve instruction-level parallelism. The instruction in Line 2 of Listing 3.1 depends on the result of Line 1 and has to wait until Line 1 finishes.

Listing 3.2 is an reordered version of the code in Listing 3.1. Independent instructions are reordered into blocks without changing the semantics of the program. The three instructions of Line 1–3 can be put into the execution pipeline without waiting for the previous memory transaction to finish.

Listing 3.1: Original instruction order

```
x = x + a
x = x + b // stalls
y = y + a
y = y + b // stalls
z = z + a
z = z + b // stalls
```

Listing 3.2: Reordered instructions

```
x = x + a
y = y + a // no latency stall
z = z + a // no latency stall
x = x + b
y = y + b // no latency stall
z = z + b // no latency stall
```

3.6. Memory-Efficient n-Dimensional Array Calculations

One strategy for implementing algorithms on GPU is to use n-dimensional arrays as data abstraction layer. Goal is to design n-dimensional arrays such that the algorithm can be reformulated into mathematical operations on the arrays. Massively parallel hardware, such as a GPU, show good performance if the operations can be split into many independent subtasks. Such a design contrasts with the object oriented design paradigm where, for instance, the programmer uses nested lists or sets of objects instead of n-dimensional arrays. Still, a list of elements with scalar properties (i.e. a `struct`) can easily be mapped to a matrix where rows correspond to elements and columns correspond to properties. A nested list of structures (i.e. a list of lists) can be mapped to a three-dimensional array

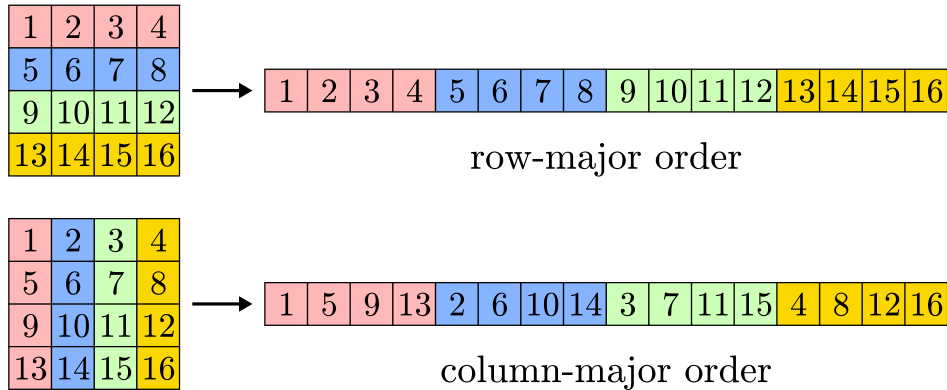

Figure 3.13.: Example for the mapping of a matrix to linear address space. Upper half: The 4×4 matrix is mapped in row-major order. Lower half: The 4×4 matrix is mapped in column-major order. Colors indicate the order of serialization.

and so forth.

Our random forest implementation on GPU strongly depends on n-dimensional arrays, such as matrices. We extracted a library from *Matrix library for CUDA in C++ and Python* [42], called `ndarray`, to implement multi-dimensional arrays on CPU and GPU. The interface is similar to the `ndarray` class in `NumPy` [33].

3.6.1. Row-major and Column-major Array Order

Multi-dimensional arrays have to be layout in a one-dimensional structure in memory. Two possible strategies are available for a two-dimensional array. Row-major order stores elements of one row consecutively in memory. Column-major order, in contrast, stores all elements in a column consecutively in memory. Row-major order is the default array ordering in languages such as `C`, `Java` or `Python`, while column-major order is the default ordering in computer languages such as `FORTRAN` or `MATLAB`. Figure 3.13 shows an example for row-major and column-major order in two-dimensional space. Both variants have benefits and disadvantages, which depends on the specific use case, such as the order of iteration. A user of our multi-dimensional array library `ndarray` can choose at compile-time which order to use for a specific array.

Row-major and column-major order can be generalized to higher dimensions. Given a d dimensional array of size $N_1 \times N_2 \times \cdots \times N_d$, we denote N_1 as first and N_d as last dimension of the array. In row-major order, elements in the last dimension are contiguous in memory. Locality of elements decreases with decreasing order of dimension. The opposite is true for column-major order, where elements of the first dimension are contiguous in memory and elements of the last dimension have minimal locality in memory.

Using an n-dimensional array to store data is not only relevant for implementing algorithms on GPU but is also relevant for parallelization on CPU. Consider the case that multiple threads iterate over a list of objects. Caching and memory pre-fetching improve if the objects are mapped to a matrix and are stored in column-major order. Intel recommends to prefer structure of arrays over an array of structures to help the compiler use vector instructions [24, Section 5.3].

4. Accelerating Random Forests

Main focus of this master's thesis lies on the acceleration of random forests. We assume the reader to be aware of the random forest machine learning algorithm, which we covered in Section 3.1.

In this section we introduce our implementation and the acceleration of random forests on CPU and GPU. Section 4.1 starts with general notes on the implementation of visual features as introduced in Section 3.2. In Section 4.2 we continue with random forests training on CPU and GPU. Section 4.3 introduces our implementation of the prediction with random forests on CPU and GPU.

4.1. Implementation of Visual Features

Visual features for object-class image segmentation, as used for the implementation in the context of this master's thesis, are calculated using the neighborhood of a pixel in the image. The feature response is calculated as the difference of two region averages that are offset around the query pixel. If at least one of the two regions lies partially or entirely outside the image, we use NaN to indicate an undefined value. NaNs have the advantageous property to propagate in floating point operations, i.e. arithmetic operations yield NaN if at least one NaN is involved in the calculation. The result of a comparison with NaN is defined to be always false.

The two region offsets and extents are normalized by the depth of the query pixel. If the depth information is missing at the position of the query pixel (cf. Section 3.2.3), the two regions cannot be normalized and the feature response is also set to NaN. Depending on how the decision node test is implemented, the decision tree is traversed in the same manner for undefined feature responses. Undefined values yield a negative test in our implementation and always flow to the right child. Hence, decision forests in our implementation can learn to treat pixels differently which are close to an image boundary or have no valid depth.

4.2. Accelerating Random Forest Training

The first step of this work is to implement the algorithm to train forests on CPU. This is necessary to later evaluate the speed improvement gained by implementing the algorithm on GPU. We base our CPU implementation on the random decision forest implementation in the *Tuwo Computer Vision Library* [32]. We refactor, parallelize and heavily optimize the implementation to make better use of the CPU cache. We then add our implementation for the normalized color and depth feature as described in Section 3.2.

4.2.1. Breadth-first Training

Decision trees can be trained in depth-first or breadth-first order. Apart from the change in the generated sequence of random numbers, decision trees that are trained in depth-first order have the same structure as trees that are trained in breadth-first order. Training a decision tree in depth-first order means that the current branch of the tree is grown recursively until the stopping criterion is reached. In contrast, training in breadth-first order grows the tree by splitting nodes in the currently bottommost level before proceeding with the next level. The original implementation in *Tuwo* executes training in depth-first order. We refactor that part of the code to grow the trees in breadth-first order, for two reasons. Firstly, training of random ferns *requires* decision trees to be trained in breadth-first order. Hence, our implementation can be adopted for training of random ferns with only a few modifications. Secondly, training in breadth-first order gives us a higher degree of freedom with respect to the generation of random feature candidates. We can choose to generate a new set of feature candidates for every node or only once per tree level. The latter variant saves training time for deep trees with several thousands of nodes. We will see in Section 5.3 that this approach does not influence segmentation accuracy, given that we generate a sufficient amount of feature candidates.

4.2.2. Random Forest Training on CPU

Before we started optimizing, we expected that about 20 % of the code accounts for 80 % of the runtime according to the Pareto principle[1]. We used *Google Performance Tools* [19] to profile our application and found that even a smaller fraction of code accounts for the vast majority of execution time. The following section focuses on optimization efforts concerning these parts of the code.

Random forests in the *Tuwo Computer Vision Library* [32] are single-threaded and individual trees are trained independently (cf. Section 3.1). *Thread Building Blocks* [51] or *OpenMP* [34] can be easily applied to parallelize training of individual trees. This is efficient if the number of trees is larger or equal to the number of available CPU cores. However, computing power is wasted if we only train few trees, but many CPU cores are available. This is the case for the computer vision applications mentioned in [43, 50]. Efficient parallelization of single tree training is a goal of this work.

Profiling indicates that training time is dominated by evaluating the best splits (Line 8 of Algorithm 1). Calculating the best split for a single node scales linearly with the number of samples, the number of random features and the number of random thresholds (cf. Toby Sharp [52, Section 5.1]). This holds only true if the number of samples and evaluated features is large. Otherwise, the overhead of parallelization and maintenance of data structures becomes relatively large.

The split evaluation is a three-level nested loop over a set of random features $\mathbf{F}$, a set of random thresholds per feature $\mathbf{T}$ and the set of all training samples D. Algorithm 3 shows a straight-forward feature evaluation implementation. The theoretical runtime complexity of Algorithm 3 is $O(F \cdot T \cdot |D|)$.

While Algorithm 3 is straight-forward to implement, CPU cache usage is not optimal. Large datasets do not fit into cache and the CPU fetches data from main memory for every

[1] https://en.wikipedia.org/wiki/Pareto_principle

Algorithm 3 Naïve implementation of the feature evaluation

Require: D samples
Require: $\mathbf{F} \in \mathbf{R}^{F \times P}$ random feature candidates
Require: $\mathbf{T} \in \mathbf{R}^{F \times T}$ random threshold candidates for each feature

1: **for all** $f \in 1..F$ **do**
2:     **for all** $\theta \in \mathbf{T}_f$ **do**
3:         initialize new histogram
4:         **for all** $d \in D$ **do**
5:             calculate feature response
6:             update histogram
7:         calculate impurity score according to histogram
8: **return** histogram with best score

evaluated feature. Main memory accesses are one or two magnitudes slower than loading data from CPU cache [11]. We confirm this assumption by profiling the application. The profiling result shows that the CPU stalls and waits for memory while feature responses are calculated (Line 5 of Algorithm 3).

To optimize our CPU implementation, we concentrate on optimizing the cache hit rate to improve the feature response calculation which we found to be the remaining bottleneck of random forest training.

Algorithm 4 Feature evaluation optimized for the CPU cache

Require: D samples
Require: $\mathbf{F} \in \mathbf{R}^{F \times P}$ random feature candidates
Require: $\mathbf{T} \in \mathbf{R}^{F \times T}$ random threshold candidates for each feature

1: initialize histograms for every feature and threshold
2: **for all** $d \in D$ **do**
3:     **for all** $f \in 1..F$ **do**
4:         calculate feature response
5:         **for all** $\theta \in \mathbf{T}_f$ **do**
6:             update according histogram
7: calculate impurity scores for all histograms
8: **return** histogram with best score

We reformulate Algorithm 3 and avoid the iteration over the dataset in the inner loop. This yields Algorithm 4 which iterates over the dataset in the outermost loop (Line 2). Note that the theoretical complexity of $O(F \cdot T \cdot |D|)$ is the same for both algorithms, as we only reordered the execution. However, the CPU needs to fetch every instance d in the dataset D only a single time. Feature responses are calculated while an instance (d) is loaded. We observe an increased cache hit rate for Line 4 in the profiling output.

The disadvantage of Algorithm 4 is a more complex data structure that is necessary to accumulate the histograms. Accessing the histogram data structure in Line 6 happens often as it is executed in the innermost loop. Again, profiling shows a large overall penalty due to a decreased cache hit rate, if the memory accesses are scattered with low locality.

To increase the cache hit rate, we need to structure the histogram counters such that

the data locality increases with the access frequency. Histogram counters for a given feature over all thresholds are updated most often because it is executed in the innermost loop. We layout the histogram data structure in a way that all counters for a given feature and threshold are located at consecutive memory addresses. This makes caching and pre-fetching more efficient. As already mentioned, the implementation becomes more complicated as a re-aggregation step in Line 7 is required. However, profiling confirms that the re-aggregation step marginally affects overall runtime performance because it is only executed once. The speed improvements due to higher cache hit rates outweigh the overhead of re-aggregation by far.

Sorting of Samples

Cache hit rate of the feature response calculation increases, if two consecutive instances $d_i, d_{i+1} \in D$ belong to the same image. We measure an improved cache hit rate if we pre-sort the entire dataset by image. The sorting need not be perfect, therefore it suffices to sort fixed-sized blocks of the dataset, which scales linearly in the size of the dataset.

Parallelization

We can easily parallelize Algorithm 4 since the calculations in Line 3-6 do not depend on each other. We use TBB [51] to partition the dataset D and run the calculations in parallel on all available CPU cores. The implementation leverages the entire computing power of the CPU even if only a single decision tree is trained.

4.2.3. Theoretical Computing Boundaries

We estimate the percentage of the theoretical computing power of a CPU that is used by our implementation. To simplify the calculation, we calculate a lower bound estimate of the time to evaluate the best split criterion by only considering the histogram update step of the split evaluation for the root node. The CPU executes the histogram update $|D| \cdot F \cdot T$ times (Line 6 of Algorithm 4).

Listing 4.1 shows the simplified source code of the histogram update. GCC 4.6.3 compiles the code of Listing 4.1 to eight X86 assembly instructions [18] with optimization flags set to "`-O3 -march=native`".

Listing 4.1: Simplified implementation of histogram update

```
1 extern double histograms[2];
2 void updateHist(double featResp, double thresh) {
3     int offset = (int)(featResp < thresh);
4     histograms[offset]++;
5 }
```

We set the parameters to $|D| = 10^6$ samples from 500 images, $F = 2000$ random features and $T = 50$ thresholds. Split evaluation of the root node takes 115 s on a quadcore Intel Core i7–920[2] clocked at 2.67 GHz. We see in the profiling output that only one instruction is executed per cycle. Pipelining and vectorization are not used because either the compiler

[2]Hyperthreading is disabled

is not capable to optimize the code using vectorization instructions, or the CPU stalls while waiting for memory.

We therefore neglect pipelining and vectorization features of the CPU and estimate the total execution time for Line 6 of Algorithm 4 as

$$\text{estimated total time} = \frac{\text{total instructions}}{\text{instructions per second}} \tag{4.1}$$

$$= \frac{8 \cdot |D| \cdot F \cdot T}{4 \cdot 2.67\ \text{GHz}} \tag{4.2}$$

$$= \frac{8 \cdot 10^6 \cdot 2000 \cdot 50}{4 \cdot 2.67 \cdot 10^9\ \text{Hz}} \approx \boxed{75\,\text{s}}. \tag{4.3}$$

Under the constraint that neither pipelining nor vectorization are used, we estimate 75 s as lower boundary of the computation. According to this simplified estimate, our CPU implementation has a computing resource efficiency of at least $\frac{75\,\text{s}}{115\,\text{s}} \approx 65.2\,\%$. Given that we use a lower bound estimate and leave out the feature response calculation, we assume our implementation to be close to the theoretical computing boundaries of the CPU. Our implementation can be further improved by using vector instructions. However, we anticipate the overall speed-up to be less than one order of magnitude since vector units of current CPUs can operate on at most four 64 bit floats per clock cycle [13].

Lower Bound Estimate on GPU

We calculate a similar estimate for the GPU with one instruction per cycle. NVCC compiles the code of Listing 4.1 to 24 instructions[3]. The Nvidia GeForce GTX 480 has 480 CUDA cores and is clocked at 700 MHz[4].

Hence, our lower bound estimate is

$$\text{estimated total time} = \frac{\text{total instructions}}{\text{instructions per second}} \tag{4.4}$$

$$= \frac{24 \cdot |D| \cdot F \cdot T}{480 \cdot 700\,\text{MHz}} \tag{4.5}$$

$$= \frac{24 \cdot 10^6 \cdot 2000 \cdot 50}{480 \cdot 700 \cdot 10^6\,\text{Hz}} \approx \boxed{7\,\text{s}}. \tag{4.6}$$

The lower bound estimate for the Nvidia GeForce GTX 690 with 3072 CUDA cores and a base clock rate of 915 MHz[5] is

$$\text{estimated total time} = \frac{24 \cdot 10^6 \cdot 2000 \cdot 50}{3072 \cdot 915 \cdot 10^6\,\text{Hz}} \approx \boxed{0.9\,\text{s}}. \tag{4.7}$$

Given the constraint of one instruction per clock cycle, we anticipate a speed-up of one or two order of magnitudes by accelerating random forest training on GPU. In practice, the actual speedup depends on additional factors such as memory throughput, caching effects, memory limits and GPU occupancy (cf. Section 3.5).

In the following section, we introduce the acceleration of random forest training on GPU.

[3] According to the disassembly output of `Nsight`

[4] http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-480/specifications

[5] http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-690/specifications

4.2.4. Random Forest Training on GPU

Evaluation of the optimized random forest training on CPU (Section 4.2.2) shows that the vast majority of training is spent for the evaluation of the best split feature. This is to our benefit when accelerating random forest training on GPU. We can restrict the implementation efforts to the relatively short feature evaluation algorithm presented in Section 4.2.2. We do not need to change the remaining CPU implementation. The code to manage the decision tree structure, for instance, is the same for the CPU and the GPU accelerated version.

For our GPU implementation we choose the CUDA framework. For basic concepts and notions of CUDA we refer to Section 3.5 and the *CUDA C Programming Guide* [9]. The CUDA framework allows us to keep most of the object oriented parts of the *Tuwo Computer Vision Library* [32]. We decide at runtime whether a calculation is offloaded on GPU or is executed on CPU. This allows us to easily measure and compare the speed difference. During development and testing, we execute the code on both GPU and CPU to ensure that the calculated results do not differ.

NVIDIA has not published sufficient details about the GPU architecture in order to optimize the runtime analytically. Instead, we use an experimental approach and carefully follow the basic optimization concepts as presented in Section 3.5. In general, we primarily optimize the code for efficient global memory accesses. In a second step we refactor the code to avoid shared memory bank conflicts if profiling indicates such events.

Degrees of freedom exist for various choices, such as the layout of data structures, which GPU memory spaces to use (cf. Section 3.5.2) or how to transfer data between CPU and GPU. The exact implementation of the algorithm is tightly coupled with these choices. Changing the order of a two-dimensional matrix from row-major to column-major (cf. Section 3.6.1), for instance, would either require a change in the order of iteration or the use of shared memory to ensure coalesced memory access patterns.

In our experimental approach, we implement multiple variants and benchmark the runtime on a small subset of the training dataset. Benchmarking the implementations with the entire training dataset is not feasible because the training takes too long and would hinder the development process.

Split evaluation can be divided into the following four phases that are executed in sequential order.

1. Random Feature Candidate Generation
2. Feature Response Calculation
3. Histogram Aggregation
4. Impurity Score Calculation

Each phase depends on results of the previous phase. We cannot execute two or more phases in parallel in consequence. However, the CPU can prepare data for the launch of the next phase, while the GPU is busy executing the current phase.

Passing results from a phase to its following phase can be a bottleneck since the computing resources of the GPU are not used at this time. Copying data between CPU and GPU amongst the phases would block the computation and slow down the whole training. In case that a node contains few samples and the entire split evaluation takes only a few milliseconds, the relative overhead to copy data between host and device would be large.

Hence, our design decision is to keep the data entirely in GPU memory, such that no memory copy is required. Note that results of each phase have to be stored in global device memory, as it is the only memory space on GPU that is persistent across multiple kernel launches (cf. Section 3.5.2).

Changing the layout of these data structures requires a subsequent performance benchmark of both affected phases. This is important since such changes can improve writing of results in one phase while the reads slow down in the following phase, or vice versa. When we do not change the data structures, we optimize and benchmark each phase independently. This simplifies and speeds up development.

Each phase is implemented in a separate kernel function, while we extract common code to device methods. NVCC inlines device methods resulting in no performance penalty of a function call.

The following sections introduce the GPU implementation of the four training phases.

Random Feature Candidate Generation

Evaluation of our CPU implementation indicates that a significant amount of training time is used for generating random feature candidates. Profiling shows that the total time for feature generation increases per level as the number of nodes increases and the time for generating features remains constant.

We subdivide the random feature candidate generation into two parts. The first part randomly generates feature parameters. In the second part, we randomly sample a fixed number of threshold candidates (cf. Section 3.1.1).

We assume the number of feature candidates F to be much larger (eg. hundreds or thousands) than the number of parameters T (eg. dozens). Hence, we decide to use one thread on the GPU per generated feature candidate. As the number of feature candidates F can exceed the maximum number of threads per block with a maximum of 1024, 1536 and 2048 for compute capability 1.2/1.3, 2.x and 3.x, respectively [37], we need to launch several thread blocks.

Generation of random numbers is implemented by the CUDA library `CURAND`. We do not observe a significant overhead or bottleneck of random number generation.

Random Parameter Generation

The first step in the feature candidate generation is to randomly select feature parameter values. We describe the features used for this master's thesis in Section 3.2. We generalize the depth and color feature parameters by using a set of 11 scalar one-byte values, listed in Section 3.2.1. The parameter generalization allows us to store F feature candidates in a $F \times 11$ matrix. We can either use row-major or column-major order.

Threshold Sampling

The second step in the feature candidate generation is the selection of one or many thresholds per feature candidate. Random threshold candidates can either be obtained by randomly sampling from a distribution or by sampling feature responses of training instances. Stückler et al. [50] implement the former approach, by uniformly sampling from an interval, which is known in advance for the given feature and image channel. *Tuwo Computer*

	Training time [ms]		
Features	Feature Generation	Feature Sorting	Total
40 000	13.3	3.0	16.3
20 000	6.6	2.7	9.3
2 000	1.5	2.6	4.2
200	1.1	2.5	3.6
20	0.9	2.2	3.1

Table 4.1.: Training time for feature candidate generation on a GeForce GTX 480. Random feature generation scales linearly for many feature candidates. However, the timings indicate a relatively large constant overhead for feature sorting.

Vision Library [32] implements the latter approach, which is more flexible if features or image channels are changed. Since our implementation is based on *Tuwo*, we stick to the approach of sampling thresholds from feature responses of training instances.

We design the data structure such that the T threshold candidates for every feature candidate are stored in a $F \times T$ matrix. Again, the matrix can be in row-major or column-major order.

All T thresholds for a given feature are sampled by the same GPU thread that generates the feature parameter. Splitting parameter generation and threshold sampling into two separate kernels would require an extra read of feature parameters from global memory. Instead, the number of feature parameters is sufficiently low to be kept in registers.

Training Time

The number of feature candidates is fixed during training. Thus, the random feature candidate generation has a constant runtime. Random feature candidate generation scales linearly in the number of feature candidates in theory. Table 4.1 shows timings of generation and sorting of random feature candidates. Feature generation scales linearly for many features (i.e. 20000 versus 40000). In case of few features (i.e. 20 versus 200) it does scale sub-linearly. We assume a constant overhead for setting up random number generation on GPU. The measurements also indicate that feature sorting does not scale linearly with respect to the number of features in the range between 20 and 40000 features.

After the feature candidate generation is finished, we store the result in two matrices, i.e. 11 feature parameters for the F features in a $F \times 11$ matrix and the T thresholds for every feature in a $F \times T$ matrix. We pass the pointers of the two matrices to the next training phase, which is the feature response calculation.

Feature Response Calculation

When we optimized the CPU implementation, we saw that training time could be significantly improved by re-arranging the loops in order to improve caching (cf. Section 4.2.2). We apply a similar optimization strategy on the GPU. Spatial locality of image channel accesses is at least as important on GPU as on CPU. Measurements show a significant speed increase for the GPU version of Algorithm 3 in comparison to the GPU version of Algorithm 4.

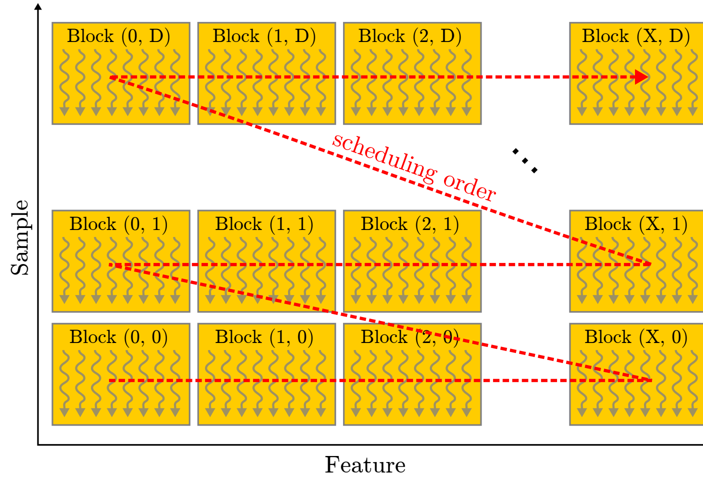

Figure 4.1.: Two-dimensional grid layout of the feature response kernel for D samples and F features. Each block contains n threads. The number of blocks in a row, X, depends on the number of features. $X = \lceil \frac{F}{n} \rceil$. Feature responses for a given sample are calculated by the threads in one block row. The arrow (red dashes) indicates the scheduling order of blocks.

Experimenting with various variants shows that it is most efficient to use one thread to calculate the feature response for a given feature and a given training sample.

The GPU schedules thread blocks by using a row-major order of the grid [39]. Execution order is determined by calculating the Block ID *bid*. In the two-dimensional case, it is defined as

$$bid = \text{blockIdx}.x + \underbrace{\text{gridDim}.x}_{\text{blocks in row}} \cdot \underbrace{\text{blockIdx}.y}_{\text{sample ID}} . \tag{4.8}$$

The number of features can exceed the maximum number of threads in a block with a maximum of 512 and 1024 threads for compute capability 1.x and 2.x/3.x, respectively, such that we need to split feature response calculation into several thread blocks. We use the x coordinate in the grid for the feature block to ensure that all features are evaluated before the GPU continues with the next sample. The y coordinate in the grid assigns training samples to thread blocks. Figure 4.1 shows the thread block layout for the feature response calculation. A row of blocks calculates all feature responses for a given sample. A column of blocks calculates the feature responses for a given feature over all samples. The dotted red arrow indicates the order of thread block scheduling. Threads reconstruct their feature id f using block size, thread and block ID, by calculating

$$f = \text{threadIdx}.x + \underbrace{\text{blockDim}.x}_{\text{threads in block row}} \cdot \underbrace{\text{blockIdx}.x}_{\text{block index in grid row}} . \tag{4.9}$$

Section 4.2.4 describes that the 11 scalar feature parameters are stored in a $F \times 11$ matrix. We use column-major order, such that the feature generation kernel as well as the feature response kernel read and write parameters with coalesced global memory access.

Attributes for samples are stored in a $D \times 4$ matrix for D samples and parameters

- Image number (32-bit integer),
- Depth (32-bit float),
- Image x offset (32-bit integer),
- Image y offset (32-bit integer).

Threads in a row of blocks access the same elements in the sample matrix, in a non-coalesced pattern. Using constant memory space would reduce the number of memory transactions in theory, as data is broadcast to all threads that access the same address. However, accessing sample data in global memory is cached and multiple threads accessing the same address already leads to a high cache hit rate. This is presumably the reason why we do not observe a performance difference by storing sample data in constant memory. We avoid the use of constant memory as it is limited to 64 kB for compute capability 1.x-3.x [9] and would restrict scalability with respect to number of samples.

After sample data and feature parameters are loaded, the kernel calculates a single feature response for either a depth feature or color feature.

To measure runtime overhead of region average calculation, we modify the feature response kernel to calculate the sum of all parameters instead of the actual region averages. Using all parameters in the calculation prevents the compiler from removing memory loads and stores that we actually want to measure. Feature response calculation for 2000 features and 20000 samples takes 62 ms on a GeForce GTX 480. The modified version, without region averages, takes only 16 ms which lets us estimate the overhead to 46 ms. Note that this experiment does not entirely factor in pipelining effects where the GPU executes arithmetic instructions while other operations are stalled on memory access.

Feature calculation is described in detail in Section 3.2. It consists of accessing image data at four pixels per region in an integrated channel of an image. It is followed by simple arithmetic operations to calculate the region sums and their difference.

We conduct a second experiment by modifying the original feature response kernel in a different way. Instead of fetching data from the image, we return a dummy value calculated as sum of the x and y region offset and the image channel number. Again, this is necessary to prevent the compiler from removing feature response calculation entirely. Feature response calculation of this modified variant takes 18 ms. This is 2 ms more than the variant that loads all parameters and stores the result to global memory. Despite the fact that both measurements can only serve as rough estimates, we conclude that accessing image data constitutes the majority of feature responses calculation runtime. In the following section we present the technique that we use to optimize image data accesses in order to reduce the runtime of feature response calculation.

Images in Texture Memory

Image data in our early experiments was stored in global memory. We tested interleaving of image channels as well as storing image channels consecutively in memory. We achieve the best results by mapping images to texture memory which reduces runtime by 35 % in our tests. The speed improvement is due to a high texture cache hit rate. Image accesses of feature response calculations for a given sample have inherently high two-dimensional spatial locality. This suits the texture cache well since it is explicitly designed for access patterns with two-dimensional locality. Texture cache is a limited resource and its size

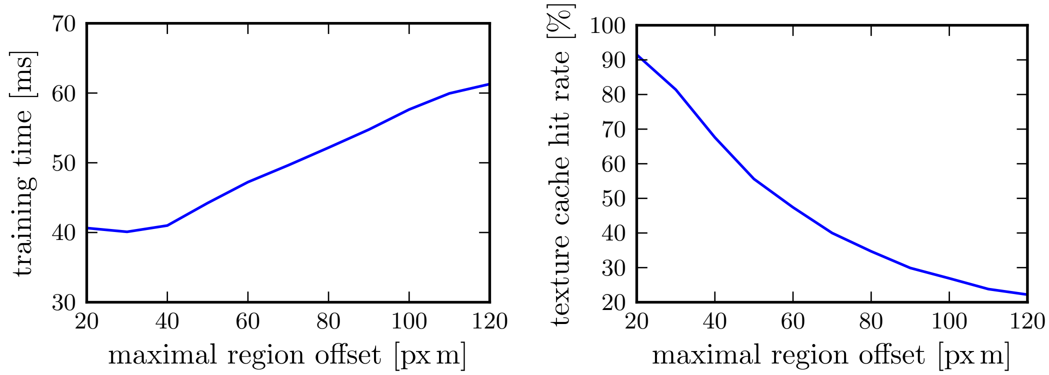

Figure 4.2.: Training time (left) and texture cache hit rate (right) of feature response kernel with respect to various region offsets. Feature response calculation for 2000 features, 2000 samples per image, 10 images and maximum region extent of 20 px m × 20 px m on a GeForce GTX 480. Texture cache hit rate (right) decreases and causes longer training times (left) for feature response calculation.

depends on the hardware in the range between 6 kB and 8 kB [37]. A texture cache of 8 kB can store, for instance, a patch of size 45 px × 45 px in a 32 bit image channel[6]. If the region offsets are too large, the GPU can only cache a subset of accessed pixels in the neighborhood of a query pixel. Figure 4.2 depicts training time penalty incurred by increasing the maximum region offset.

Increasing texture cache hit rate is our primary optimization goal to improve feature response calculation which we cover in the following section.

Sorting Features and Samples

We mentioned in Section 4.2.2 that sorting the samples improves cache hit rate of our CPU implementation. We measure a similar effect when samples are sorted on the GPU. We use *Thrust – Parallel Template Library* [23] to sort feature candidates in ascending order by

1. Feature Type,
2. Channel 1,
3. Channel 2,
4. Region 1 offset y,
5. Region 1 offset x.

Sorting by feature type reduces branch divergence in the feature response calculation. This is important since the GPU does only execute threads in parallel that follow exactly the same execution path.

Sorting by feature type and the two channels increases the probability that the same channels are accessed in consecutive region average calculations. Improving spatial locality helps to increase the hit rate of the texture cache and the L2 cache.

[6] $\sqrt{\frac{8\,\text{kB}}{32\,\text{bit/px}^2}} \approx 45.3\,\text{px}$

Sorting		Training time	Texture cache	L2 cache
Features	Samples	[ms]	Hit rate [%]	
Yes	No	122.6	19.1	54.3
No	Yes	72.4	10.4	97.7
Yes	Yes	**62.2**	22.2	96.7

Table 4.2.: Training time and texture cache hit rate of feature response calculation for 2000 features, 2000 samples per image, 10 images, maximum offset of 120 px m and maximum region extent of 20 px m × 20 px m on a GeForce GTX 480. Sorting features leads to a significantly increased texture cache hit rate. Sorting samples by image ID leads to a significant increase of L2 cache hit rate for read requests from texture cache.

Additionally sorting by the y and x coordinate (in that order) of the first region offset helps to increase two-dimensional locality in the feature response calculation. We do not measure a speed improvement by adding the second offset as sort criterion.

We adjust random feature generation such that features are already sorted by feature type, channel 1 and channel 2. We do not generate features such that they are already sorted by the additional criteria region 1 offset y and offset x because we found that parallel feature generation would become too complex to implement for an arbitrary number of feature candidates. Our implementation idea was to layout a two-dimensional grid of cells, which is used by individual threads to sample an offset from. However, it is not possible to perfectly fit the grid for any feature candidate generation parameterization such as the number of features or the maximum region offset. An imperfectly fitted grid would result in a biased distribution of offsets, which is why we neglect this idea and follow the approach to sort the samples afterwards.

Table 4.2 shows profiling measurements of the feature response kernel with and without sorting of samples and features. The results show that both sorting samples by image ID and sorting features increases cache hit rate and reduces training time. Sorting samples is only necessary once before training a tree. However, sorting features is required after new random features are generated. Parallel sorting on GPU, as implemented in *Thrust – Parallel Template Library* [23], takes 2.6 ms for 2000 features on a GeForce GTX 480. Sorting time is constant with respect to number of features, while feature response calculation time depends on the number of samples. Hence, the relative overhead of sorting increases when the number of samples decreases. We implement a heuristic to only sort features when the expected speed improvement is larger than the cost of sorting. We measure the break-even point of speed gain and sorting cost at about 10000 samples. Features are not sorted in this case, which is possible as the selection of the best feature is invariant to the order of feature evaluation.

Multiple Samples per Thread Block

The fact that we sample hundreds or thousands of pixels per image leads to the idea to exploit the spatial locality between samples to increase texture cache hit rate. Uniformly sampling 2000 pixels from each training image, for instance, means that we randomly draw

Samples	Features	Training time	Texture cache	L2 cache
per thread block	per thread block	[ms]	Hit rate [%]	
1	128	55.0	22.1	97.8
4	32	51.5	26.0	97.3
8	16	50.5	28.2	97.2
16	8	50.2	30.2	97.5
16	16	49.3	35.2	97.3
32	8	**48.9**	35.2	97.6
32	4	50.2	31.3	97.7

Table 4.3.: Training time and texture cache hit rate of feature response calculation for **2000 samples per image**, **10 images**, 2000 features, maximum offset of 120 px m and maximum region extent of 20 px m × 20 px m on a GeForce GTX 480. Feature response values are stored to a $D \times F$ matrix for D samples and F features in **row-major** order. Training time decreases with more samples per thread block, because texture cache rate increases. Performance peaks at 32 samples and 8 features per thread block.

$\sqrt{2000} \approx 44.7$ samples per row and per column. For a 640 px × 480 px image, this translates to an average horizontal distance of $\frac{640\,\text{px}}{\sqrt{2000}} \approx 14.3\,\text{px}$ and an average vertical distance of $\frac{480\,\text{px}}{\sqrt{2000}} \approx 10.7\,\text{px}$. In this case, average distance between two samples is lower than distance of two queried regions around a query pixel for a maximum offset of 120 px m.

We modify the feature response kernel to calculate feature responses for multiple samples in one thread block. The occupancy calculator of NVIDIA predicts highest theoretical occupancy for either 128 or 256 threads per block. Table 4.3 shows that texture cache hit rate increases as expected and training time decreases, if one thread block calculates feature responses for several samples. We measure peak performance at 32 samples and 8 features per thread block.

The test setup with 10 images and 2000 samples per image does not necessarily reflect average real world behavior of random forest training. As a tree is grown and gets deeper, every node contains less samples. This implies that the number of samples per image is reduced with respect to tree depth, given uniform sampling. Table 4.4 shows training time and cache hit rate for the same total number of samples[7] but distributed across 100 images. Performance peaks at 4 samples and 32 features per thread block.

The effect becomes more visible when we further reduce the number of samples per image. The average distance between two samples increases in this case, given that samples are distributed uniformly.

Table A.1 in Appendix A shows that texture cache and L2 cache hit rate decrease significantly, if more samples are calculated per thread block. Performance again peaks at 4 samples and 32 features per thread block.

[7] $200\ \frac{\text{samples}}{\text{image}} \cdot 100\ \text{images} = 2000\ \frac{\text{samples}}{\text{image}} \cdot 10\ \text{images} = 20000\ \text{samples}$

Samples	Features	Training time	Texture cache	L2 cache
per thread block	per thread block	[ms]	Hit rate [%]	
1	128	57.5	24.9	92.8
4	32	**56.8**	22.7	92.4
8	16	61.7	17.8	93.1
16	8	66.6	14.1	92.1
16	16	61.9	20.0	91.7
32	8	69.7	14.9	88.0
32	4	75.8	10.9	88.4

Table 4.4.: Training time and texture cache hit rate of feature response calculation for **200 samples per image**, **100 images**, 2000 features, maximum offset of 120 px m and maximum region extent of 20 px m × 20 px m on a GeForce GTX 480. Feature response values are stored to a $D \times F$ matrix for D samples and F features in **row-major** order. Training time increases with more samples per thread block, because texture cache rate decreases. Performance peaks at 4 samples and 32 features per thread block.

Column-major Order

Calculating feature responses for several samples per thread block has a second positive effect besides the increase of cache hit rate. Until now we assumed to write feature responses in row-major order to the $D \times F$ result matrix for D samples and F features. However, as we will see, histogram aggregation benefits if feature responses are stored in column-major order.

Given that feature responses are calculated for multiple samples per thread block we can swap the x and y thread indices in a block, in order to achieve coalesced writes to the $D \times F$ matrix in column-major order.

Table A.2 in Appendix A shows that 4 samples per thread block suffice to achieve the maximum global memory store efficiency [10] of 100 %. Tables A.3 and A.4 in Appendix A show that calculating 128 features and only one sample per thread block is penalized due to a suboptimal global memory store efficiency of 25 %.

We choose to use a fixed configuration for the feature response kernel with 32 features and 4 samples per thread block. It achieves optimal global memory store efficiency for both row-major and column-major order of the result matrix. Texture cache hit rate and training time is a good compromise for both cases, with few and many samples per image.

Future work could investigate a dynamic thread block configuration. We anticipate a possible improvement by selecting the best configuration depending on parameters such as sampling depth and maximal region offset.

After the feature response calculation is finished, we pass the memory address of the feature response matrix to the next training phase.

Histogram Aggregation

Before we can calculate impurity scores we have to aggregate feature responses into per-class histograms. Feature responses are real-valued which means that finding the best threshold is a sub-problem of finding the best split criterion. We follow the approach of

Stückler et al. [50], Toby Sharp [52] and use multiple quantization levels as described in Section 3.1.1, instead of trying to find the best threshold analytically. For each feature, each class, and each quantization level (i.e. threshold) we need two counters for the left child and the right child. We maintain per-class histogram counters in a four-dimensional matrix of size $F \times T \times C \times 2$ for F features, T quantizations levels (thresholds) and C classes.

Our CPU implementation aggregates histogram values directly after feature response values are calculated (cf. Algorithm 4). Splitting feature response calculation and histogram aggregation into two kernels makes it possible to have different orders of iteration.

CUDA allows us to implement histogram counter updates in a simple function, without taking the detour of a vertex shader as proposed by Scheuermann and Hensley [40], Toby Sharp [52].

Algorithm 5 outlines the histogram aggregation phase on GPU.

Algorithm 5 Histogram aggregation on GPU

Require: $\mathbf{F} \in \mathbf{R}^{D \times F}$ feature responses for D samples and F features
Require: $\mathbf{T} \in \mathbf{R}^{F \times T}$ random threshold candidates for each feature
Require: $\mathbf{C} \in \{1..C\}^D$ class for each sample

```
1: H ← 0_{F,T,C,2}                          ▷ initialize histogram counters
2: for all f ∈ 1..F do
3:     for all θ ∈ T_f do
4:         for all d ∈ 1..D do
5:             c ← C_d                       ▷ class c of sample d
6:             if F_{d,f} ≤ θ then
7:                 H_{f,θ,c,0} ← H_{f,θ,c,0} + 1   ▷ increment left histogram counter
8:             else
9:                 H_{f,θ,c,1} ← H_{f,θ,c,1} + 1   ▷ increment right histogram counter
```

Histogram Counters in Shared Memory

Updating the histogram counter in $\mathbf{H}$ (Line 7 and Line 9 of Algorithm 5) is executed $F \cdot T \cdot D$ times. Accordingly, updating $\mathbf{H}$ directly in global memory for every sample, every feature and every threshold is computationally too demanding.

Note that we iterate over all samples in the innermost loop in our algorithm. This allows us to keep the histogram counters in shared memory. Every thread block needs to allocate at least a matrix of size $C \times 2$ to store left and right counter values for C classes. As shared memory is a limited resource, we can only keep histogram counters for one feature and one threshold in shared memory. We implement iteration over features and thresholds as thread blocks in a two dimensional grid on GPU as depicted in Figure 4.3. Each thread block slices samples into partitions such that all threads in the block can aggregate histogram counters in parallel.

Iteration over features and samples in the histogram aggregation kernel is in the transposed order of the iteration in the feature response calculation. In consequence, reading feature responses from global memory would be a bottleneck if the feature responses would be stored in row-major order. We saw in Section 4.2.4 that we were able to design the feature response calculation phase such that the result can be stored in column-major order

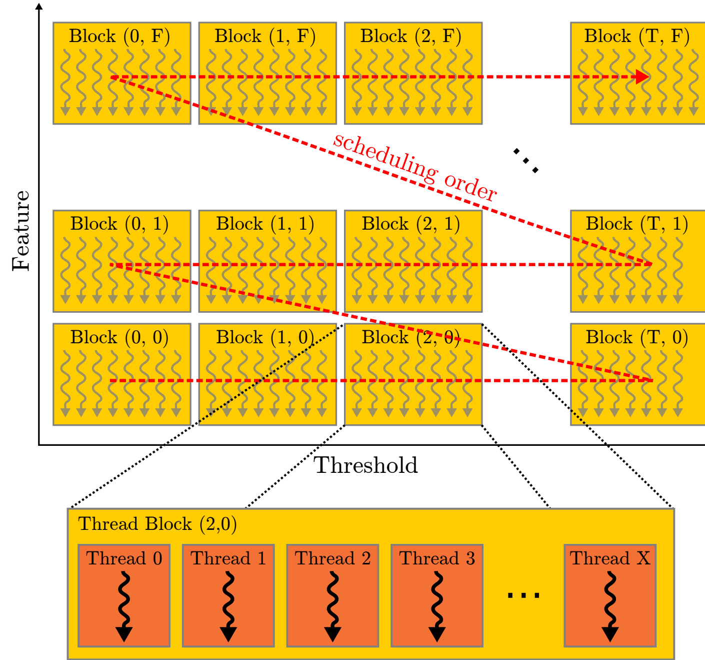

Figure 4.3.: Thread block layout of the histogram aggregation kernel for F features and T thresholds. One thread block per feature and per threshold. X threads in block aggregate histogram counters for D samples in parallel. Every thread iterates over at most $\lceil \frac{D}{X} \rceil$ samples.

without a significant performance penalty. This leads to a significant performance gain in the histogram aggregation phase since feature responses in global memory are fetched in a coalesced pattern with maximal read efficiency.

Summing up counters in shared memory amounts for the majority of execution time. We tested several variants that were inspired by Xu et al. [60]. We saw a significant performance drop when using atomic updates, which are necessary to handle concurrent counter updates. To avoid atomic operations entirely, every thread gets a distinct region in shared memory. For X threads and C classes, we need to allocate $X \cdot C \cdot 2$ counters. The downside of this method is an increased consumption of shared memory and the requirement of an additional reduction phase to reduce the $X \times C \times 2$ counters to a final sum matrix of size $C \times 2$ for every feature and every threshold. Our implementation of the final reduction is inspired by the sum kernel in *Thrust – Parallel Template Library* [23]. Figure 4.4 shows our first approach of histogram aggregation and sum reduction. Every thread increments a dedicated counter for each class in the first phase. Bank conflicts do not occur in this phase as we use thread block sizes that are a multiple of 16 such that counters of any two threads are mapped to different banks (cf. Section 3.5.2).

In a subsequent phase, we iterate over all C classes and reduce the counters of every thread in $O(\log X)$ steps, where X is the number of threads in a block. Every thread calculates the sum of two counters in a step. The structure of reductions, as depicted in Figure 4.4, is a binary tree in upside down order. Note that threads read and write counters such that no bank conflicts occur.

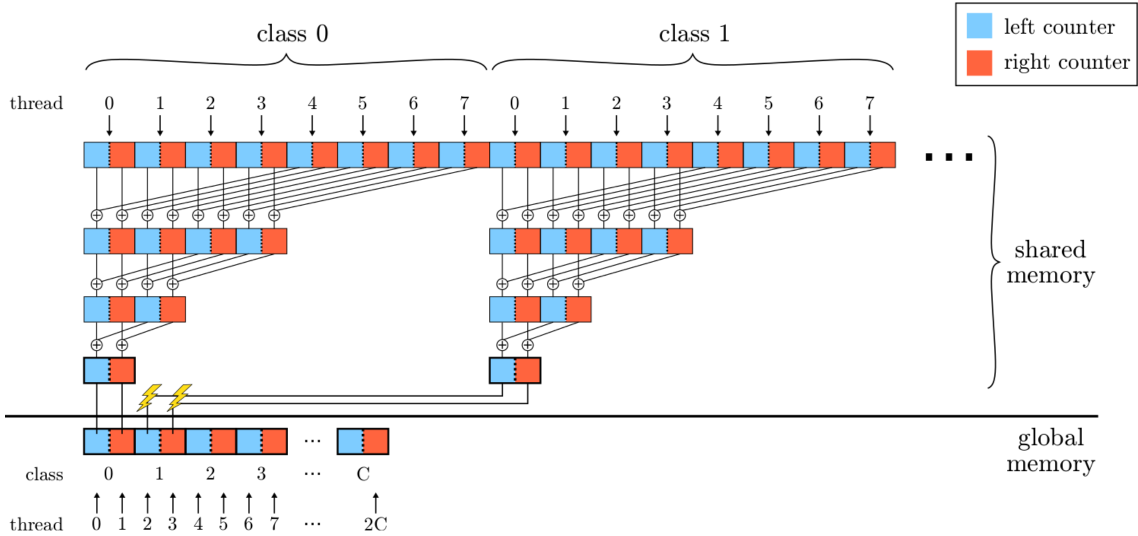

Figure 4.4.: Reduction of histogram counters. Every thread sums to a dedicated left and right counter (indicated by different colors) for each class (first row). Counters are reduced in a subsequent phase. Final counter pairs (thick border) are copied to global memory in a coalesced pattern. The lightning bolds indicate shared memory bank conflicts when copying final counters from shared to global memory.

We wait until the loop over all classes finished computing of the final left and right counter. This last step can be executed in parallel by $2 \cdot C$ threads that copy the left and right counter of C classes. The GPU thereby minimizes global memory write transactions since the threads write in a coalesced pattern. However, reading counter data from shared memory causes bank conflicts, as indicated by lightning bolts in Figure 4.4. The reason is the distance between each left and right counter pair which is a multiple of 16. All C threads access the same bank when reading the left counters. Accordingly, all C threads also access the same bank when reading the right counter.

Figure 4.5 shows an improved version that avoids bank conflicts. Final counters are written to shared memory at consecutive addresses by reusing memory locations that are no longer in use after the reduction of the previous class is finished. Profiling shows that no more bank conflicts occur with this variant, which results in a significant performance increase.

Measurements depicted in Table 4.5 show that the histogram aggregation kernel training time scales linearly with respect to the number of feature candidates and the number of thresholds. Contrastingly, Table 4.6 shows that the training time does *not* scale linearly with respect to number of samples. The reason is the overhead to reduce the histogram counters, which scales with the number of classes.

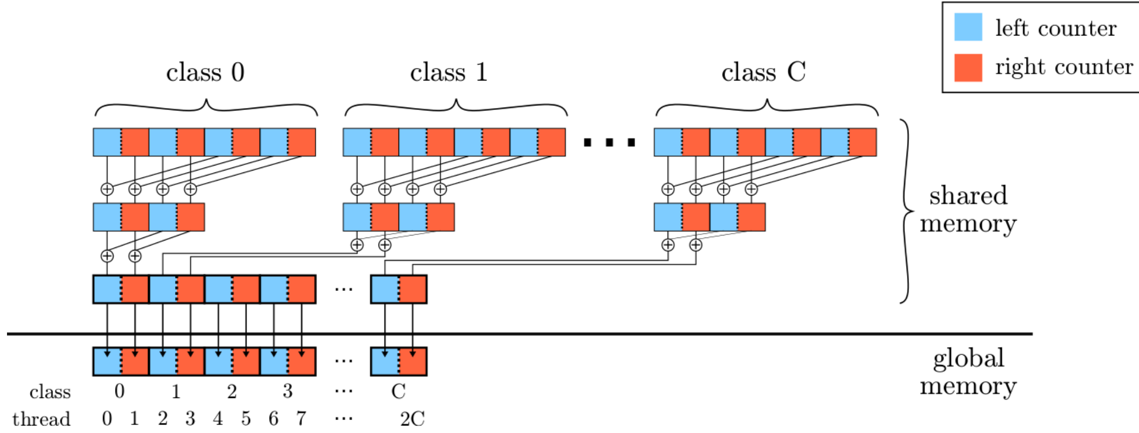

Figure 4.5.: Reduction of histogram counters without bank conflicts (cf. Figure 4.4). The last reduction step stores counters in shared memory, such that no bank conflicts occur when copying to global memory.

Skipping Classes without Samples

The binary reduction of counters (Figure 4.5) has a constant runtime overhead per class. We can entirely skip reduction of counters for classes without samples, as all counters are already zero in this case and would not change during reduction. Every thread maintains a thread-local bit set to store the list of classes with a positive number of samples. During the loop over all classes in the reduction phase, we use the

```
int __syncthreads_or(int predicate);
```

function, which is available since compute capability 2.x in the CUDA framework, to communicate this information across threads. The function returns a non-zero value if and only if `predicate` is a non-zero value for any of the threads. We set `predicate` to 1 if a thread counts one or more samples for a given class, or 0 otherwise. We skip counter reduction if `__syncthread_or()` returns 0, which means that no thread counted a sample.

Result of the histogram aggregation phase is the four-dimensional counter matrix of size $F \times T \times C \times 2$ for F features, T thresholds and C classes. The matrix is stored to global memory in row-major order.

Impurity Score Calculation

Computing impurity scores from the four-dimensional counter matrix is the last of the four training phases that we execute on GPU.

We implement the score kernel in a straightforward method. We use 128 threads per block, any of which computes the score for a different pair consisting of feature and threshold. Each thread loads $C \cdot 2$ counters from the four-dimensional counter matrix in global memory, calculates the impurity score and writes the resulting score back to global memory. Table 4.7 shows that the more complex normalized information gain score function implies a significant performance overhead in double precision mode when compared to the simple information gain score function.

Features	Thresholds	Training time [ms]
2 000	50	82.7
2 000	25	41.6
1 000	50	42.3
1 000	25	21.5
500	50	20.7
500	25	10.5
200	50	8.5
200	25	4.3

Table 4.5.: Histogram aggregation kernel execution times for different number of feature and threshold candidates. Measured on a GeForce GTX 480 with 5 classes and 20000 samples from 10 images. Training time scales almost linearly with respect to number of features and number of threshold candidates.

Features	Training time [ms]
30 000	119.6
20 000	82.7
10 000	50.6
1 000	12.6
100	8.1

Table 4.6.: Histogram aggregation kernel execution times for different number of samples. Measured on a GeForce GTX 480 with 5 classes, 2000 feature candidates, 50 thresholds and 10 images. Training time does *not* scale linearly with respect to number of samples. The reason is the constant time overhead for the reduction of histogram counters.

We tested different variants to make use of shared memory in order to read counter data in a coalesced pattern. We did not measure a significant performance benefit, while code complexity increased drastically. Since the impurity score calculation only accounts for a small fraction of the total training time on GPU, we did not spend more effort to evaluate possible optimizations.

We store the calculated scores in a $T \times F$ matrix for T thresholds and F features. The score matrix is finally transferred from device to host memory space which finishes the training phase on GPU.

Limitation of Global Memory

Global memory on GPUs is a limited resource. In this section we introduce the techniques that we use to scale our training implementation to large datasets that do not fit in device memory.

	Training time [ms]	
Impurity score function	Double precision	Single precision
Normalized information gain	1.23	0.57
Information gain	0.81	0.57
NoOp	0.38	0.38

Table 4.7.: Score kernel execution times for different impurity score functions (cf. Section 3.1.1). Measured on a GeForce GTX 480 with 2000 feature candidates, 5 labels and 50 thresholds. Execution time significantly decreases in double precision mode for the more complex normalized information gain score function. NoOp is a simple dummy function that just calculates a sum of the counters. We use it to estimate an upper bound for the memory access overhead of the score kernel.

Slicing of Samples Training arbitrarily large datasets with many samples can exceed storage capacity of global memory. The feature response matrix of size $D \times F$ (cf. Section 4.2.4) scales linearly in the number of samples D and the number of feature candidates F. We cannot keep the entire matrix in global memory if D or F is too large. For instance, training a dataset with 500 images, 2000 samples per image, 2000 feature candidates and double precision feature responses (64 bit) would require $500 \cdot 2000 \cdot 2000 \cdot 64\,\text{bit} \approx 15\,\text{GB}$ of global memory for the feature response matrix in the root node split evaluation.

To circumvent this limitation, we split samples into partitions and sequentially compute feature responses and aggregate histograms for every partition. The maximum possible partition size depends on the available global memory of the GPU.

Image Cache The memory consumption of a $640\,\text{px} \times 480\,\text{px}$ RGB-D image is $640 \cdot 480 \cdot 5 \cdot 32\,\text{bit} \approx 5.86\,\text{MB}$ for its three integrated color and two integrated depth channels. Hence, we might not be able to keep all images of a large dataset in global memory. We implement an image cache with Least Recently Used (LRU) strategy that keeps a fixed number of images in global memory. The slicing of samples takes care that a partition does not require more images in memory than fit into cache. The experimental results in Chapter 5 show that training with more images than fit into cache can cause a significant performance penalty. This is due to the fact that during training images are regularly evicted and later transferred back from host to device. As an alternative, our implementation can sample the dataset such that only a subset of images are used and no costly, regular image transfers occur during training.

Hiding Memory Latency

We need to regularly transfer data from host to device and vice versa during training. Before the evaluation of a node split, we need to transfer the current feature candidate parameters. During the evaluation we need to transfer sample data for every partition of the samples. If the image cache cannot hold all images in memory, we also need to transfer images from host to device. At the end, the final scores need to be transferred from device to host memory.

Recent GPUs are able to partially hide latency caused by such host to device or device to host memory transfers.

Streams The GPU can execute memory transfers and kernels concurrently when they belong to different streams (cf. Section 3.5.2). We use two streams to perform some memory transfers and kernels in parallel. We execute the training of multiple trees in separate host threads. For instance, feature parameters for one tree can be transferred from host to device while histogram aggregation is running for a different tree.

Memory Pooling During profiling we saw that calls to `cudaMalloc()`, `cudaFree()`, `cudaHostMalloc()` and `cudaHostFree()` cause the entire GPU to pause processing. To avoid frequent memory allocations, we use memory pooling in our n-dimensional array implementation. Memory is reused that is already allocated but no longer in use. Due to the structure of random decision trees, evaluation of the root node split criterion is guaranteed to require the largest amount of memory. Child nodes always contain less or equal samples than the root node, which means that all data structures have at most the size as the structures used for calculating the root node split. Given this fact, we are able to train a tree without a single reallocation of memory.

After the training of all decision trees in the random forest is finished, we can store the tree and use it for prediction.

4.3. Accelerating Random Forest Prediction

The following section introduces our implementation and the acceleration of random forest prediction on CPU and GPU.

The first step of prediction is to load all decision trees of a random forest. We assume that we can keep the entire decision tree data structures in main memory throughout prediction. Prediction is performed in a loop over all pixels of the input image. Each decision tree in the random forest is then used sequentially to retrieve the probability distribution over all classes for the given pixel, as outlined in Section 3.1.2. The probability distributions which we retrieved from every tree in the forest are averaged and then returned to the user. To measure segmentation accuracy, we take the class labeling with highest probability and compare it with the label in the ground truth. We count the mismatches of predicted label and the label in the ground truth for the input pixel. The corresponding loss function is called misclassification rate or 0-1 loss [15].

Histogram Bias Stückler et al. [50] found that unbalanced class pixel occurrences in the image, such as frequent background pixels in the AIS Bonn Large-Objects dataset, can degrade segmentation accuracy significantly. The reason is the histogram adjustment by prior class distribution of Equation (3.7). A single pixel from a frequent class, that reaches a leaf during training, can outweigh multiple pixels from a less frequent class. In order to counter that effect, we subtract a fraction of all pixels from the class histogram of each

leaf node l_k by a user-specified parameter ρ, called "histogram bias", by

$$p_\rho(c \mid l_k) := \begin{cases} \dfrac{p'_\rho(c \mid l_k)}{\sum_{c' \in \mathcal{C}} p'_\rho(c' \mid l_k)} & \text{if } p'_\rho(c \mid l_k) > 0 \\ 0 & \text{otherwise,} \end{cases} \tag{4.10}$$

where

$$p'_\rho(c \mid l_k) := \max\left(0, p(c \mid l_k) - \rho \sum_{c' \in \mathcal{C}} p(c' \mid l_k)\right) \text{ with } 0 \leq \rho \leq 1. \tag{4.11}$$

4.3.1. Random Forest Prediction on CPU

Fast prediction is important for real-time applications as presented by Lepetit et al. [28] and Shotton et al. [43]. To accelerate prediction on CPU, we use multiple threads to process each image. After images are loaded from disk, we convert the image colors to CIE Lab space and calculate the integral image in a pre-processing step. Calculating image integrals is expensive with respect to processing time [52]. We accelerate it by calculating the integral for each of the five image channels in parallel with separate threads.

The main loop in the prediction iterates over every pixel in the image and traverses each decision tree in the forest until it reaches a leaf. We parallelized this loop by slicing the image into n partitions that are processed by n threads in parallel. This uses all of the available computing power of the CPU. Prediction time depends on the complexity of the feature but scales linearly with the number of trees, depth of the trees and the number of pixels in the input image. For typical parameters of three trees with a depth of 15 levels, we measure about 0.3 s to perform a dense prediction of a 640 px × 480 px image on a quadcore Intel Core i7–920. This is not sufficient for real-time applications. Toby Sharp [52] was able to accelerate random forest prediction by a factor of about 100 on a GPU. This result inspired us to follow the same path and accelerate our implementation on GPU.

4.3.2. Random Forest Prediction on GPU

In the previous section we mentioned that individual pixels of an image can be processed independently for prediction. Similar to the slicing of the image in the CPU implementation, we slice the image in patches such that pixels in each patch are processed in parallel by a block of threads on the GPU. In contrast to a CPU, the threads in a block on GPU share the same texture and L2 cache. This is why we design the code such that a block of threads processes pixels that are spatially close to each other.

Table 4.8 shows that we achieve best performance by using 128 threads to classify an image block-wise with patches of size 8 px × 16 px. Spatial locality and cache hit rate correlate and thus are both maximized for queries in the feature response calculation (cf. Section 4.2.4).

The work of Toby Sharp [52] inspired us to traverse the decision tree in a branch-less loop. We also followed the idea to map the tree data structure to a two-dimensional texture in GPU memory. However, we use the CUDA framework instead of Direct3D which lets us avoid the use of shaders. As afore mentioned in Section 3.2.3, the Microsoft

Patch size	Time	Texture cache	L2 cache	Occupancy
[px × px]	[ms]	Hit rate [%]		[%]
16 × 16	4.18	82.5	89.1	58.6
8 × 8	5.03	81.5	87.9	31.4
32 × 8	4.47	79.9	87.9	60.6
16 × 8	4.08	80.5	87.3	60.3
8 × 16	**3.95**	80.6	90.1	58.4
8 × 32	4.04	82.2	91.9	56.3

Table 4.8.: Random forest prediction runtime for a 640 px × 480 px RGB-D image with different patch sizes. Execution times are per-tree averages. We use a random forest trained on the AIS Bonn Large-Objects dataset with a maximal tree depth of 15 levels. Processing the images with patches of size 8 px × 16 px shows best performance because of a high occupancy, texture and L2 cache hit rates.

Kinect camera does not guarantee to deliver a valid distance measure for every pixel in the depth channel. Hence, in contrast to Toby Sharp, we cannot generalize to a single type of features as we need a different implementation for depth and color features. Shotton et al. [43], who use the implementation of Toby Sharp, do not encounter that problem because they only use depth information and query single pixels instead of region sums.

We reuse the code of random forest training and map an image to a two-dimensional layered texture in global memory. We store tree data as two-dimensional structure in global memory and map it also to a two-dimensional layered texture, where we use a dedicated layer for each tree. Nodes are serialized in breadth-first order. Deep trees with thousands of nodes can exceed the size limits of a layered texture which is 8192 and 16384 nodes for compute capability 1.x and 2.x/3.x, respectively. In this case we map the nodes to several layers in the texture. Figure 4.6 shows an example of a tree with six nodes that is mapped to a layer in the two-dimensional texture. Only the left child node ID is stored since the right child node always directly follows the left child, due to breadth-first order. A negative left child node ID indicates a terminating leaf node. Leaf nodes do not store parameters but contain the per-class probability distribution (histogram) as determined during training.

Every thread processes trees of the random forest sequentially and traverses the decision tree from root to leaf in a loop as outlined in Algorithm 6. All threads access a tree level at the same time which increases spatial locality and cache hit rate. We avoid branches in the loop by expressing the node index update (Line 6 of Algorithm 6) as

$$\text{node idx} \leftarrow \text{left node idx} + v, \tag{4.12}$$

where v is 0 if the feature response is less or equal than the threshold (left child) or 1 otherwise (right child).

Integration of the image channels is performed on CPU as we do not anticipate a significant speed increase on GPU (cf. [52, Section 5.1]).

Our implementation scales linearly in the number of trees in the forest, maximum depth of the decision trees and the number of pixels in the image. Feature complexity and

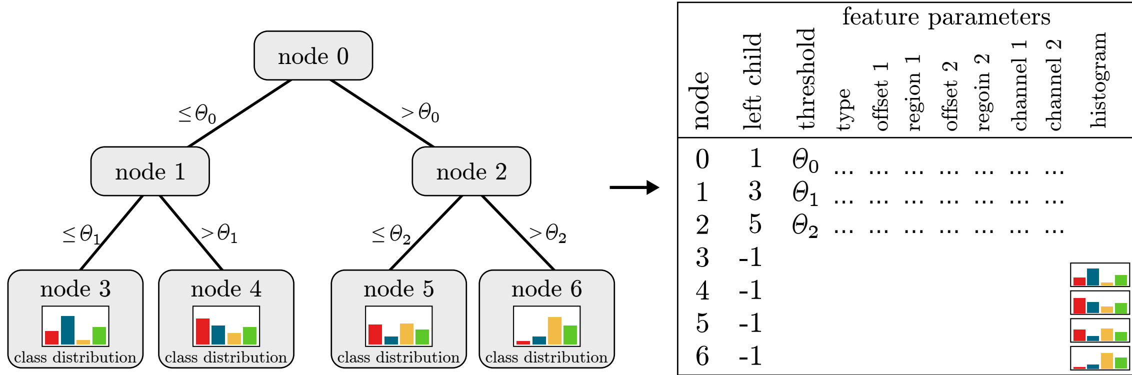

Figure 4.6.: Mapping of a binary tree with six nodes (left) to a layer in the 2-dimensional texture on GPU (right). Every node is mapped to a row in breadth-first order. Decision nodes contain left child node ID, threshold and feature parameters. Leaf nodes are indicated with a negative child ID and contain the class distribution as determined during training. Empty cells indicate unused values, such as missing feature parameters in leaf nodes.

Algorithm 6 Random forest prediction on GPU

```
1: for all levels in tree do
2:     if leaf node reached then
3:         return leaf node histogram
4:     Load feature parameters from current node
5:     Calculate feature response
6:     Update current node index depending on feature response and threshold
```

choice of parameters have significant impact on overall runtime performance. Larger region offsets, for instance, lower the cache hit rate which causes additional global memory fetches.

Combined class probabilities of all leaf nodes are returned in a $C \times W \times H$ array for C classes and an image of size $W \times H$.

4.4. Accelerating Random Ferns

The work of Rodrigues et al. [38] inspired us to evaluate random ferns, a random forest variant, that we introduced in Section 3.1.3. Rodrigues et al. use random ferns for a 6D pose estimation of shiny objects on RGB images. They chose to use random ferns because of two reasons. Firstly, the split criteria (questions) of a random forest with l levels have an exponential memory consumption with a complexity of $O(2^l)$. Random ferns, in contrast, only contain l questions. Secondly, the split criteria of a random fern have the form of a list where each element does not depend on its predecessor. This is an advantageous property for parallelization of random fern prediction as the entire list of questions is known in advance and results can be computed in parallel.

The following sections present our method to implement random fern training and random fern prediction. We do not implement random fern prediction on CPU as experimental results of random forest prediction indicate the GPU being predominant with a speed-up factor of over a magnitude.

4.4.1. Random Fern Training

Random ferns are specialized random forests where the nodes in a level share the same split criterion.

In contrast to the original random fern training method [36], we base our random fern training on our random forest implementation and change the best split criterion selection such that decision trees with the shape and restrictions of random ferns are constructed (cf. Section 3.1.3). Our method requires the training to proceed in breadth-first order which we fortunately already implement for random forest training as mentioned in Section 4.2.1. To select the same split criterion for all nodes in a level, we only need to change that part of the code which retrieves the impurity scores and selects the best split criterion. Instead of selecting an individual split criterion for each node, we combine the scores over all nodes in the level and select the split criterion that maximizes the common good.

In our first approach we selected the split criterion s' that maximizes the average over all impurity scores as defined by

$$s' = \arg\max_{s \in \mathcal{S}} \frac{1}{|\mathcal{N}|} \sum_{n \in \mathcal{N}} I_s(n), \tag{4.13}$$

where $\mathcal{N}$ is the set of nodes in the current level, $\mathcal{S}$ is the set of evaluated split criteria and $I_s(n)$ is the impurity score of node n and split criterion s. However, we observe a poor segmentation accuracy of random ferns that are trained with this method.

Instead, we use s^* which is the average score weighted by the number of samples in a

node as defined by

$$s^* = \arg\max_{s \in \mathcal{S}} \frac{\sum_{n \in \mathcal{N}} Q_n \cdot I_s(n)}{\sum_{n \in \mathcal{N}} Q_n} = \arg\max_{s \in \mathcal{S}} \sum_{n \in \mathcal{N}} Q_n \cdot I_s(n), \tag{4.14}$$

where Q_n is the number of training samples assigned to node n. All other parts of random fern training remain unchanged and are identical to the training of random forests.

4.4.2. Random Fern Prediction on GPU

Since a random fern is a random forest, we can reuse our random forest implementation for random fern prediction. However, we thereby do not leverage the properties of a random fern to accelerate the prediction.

In Section 4.3.2 we presented the strategy to map the random forest data structure to a texture in GPU memory. This approach is not efficient if it is applied as-is for random ferns. The same split criterion is duplicated for all nodes in a level. Thus, we do not achieve higher texture cache hit rates when loading the parameters. In our random fern implementation we change the random forest texture mapping (see Figure 4.6) to use two independent textures instead. The first texture only stores the left child node offsets for decision nodes while the second texture stores the histograms of all leaf nodes. The feature parameters for the split criteria are stored in an one-dimensional array since we only need to store one test per level. Since the array has only a size in memory of $O(l)$ for l levels in the fern, we place this array in constant memory on GPU. Constant memory has the advantageous property that values are broadcast to all threads which are accessing the same location in memory (cf. Section 3.5.2). This happens while the threads for each pixel in the patch traverse the tree from root to leaf.

5. Experimental Results

In Chapter 4 we introduced our methods to accelerate random forest training and prediction on CPU and GPU. In this section we discuss the results of experiments that we conduct on two different RGB-D datasets. We describe the two datasets in Section 5.1. In Section 5.2 we present the parameters that we use for random forest training and prediction on the two datasets. Section 5.3 evaluates the effect of our strategy to generate a new set of feature candidates only once per level during training. In Sections 5.4 and 5.5 we show experimental results concerning the runtime of random forest training and prediction, respectively. We will see that we achieve a significant training and prediction speed-up by accelerating the random forests on GPU. Section 5.6 presents the results of experiments that measure segmentation accuracy with random forests and variants that we trained on the two datasets. Section 5.7 closes with the results on the acceleration of prediction using random ferns.

5.1. Datasets

We conduct our experiments on two RGB-D datasets that we describe in the following sections.

5.1.1. AIS Bonn Large-Objects dataset

The AIS Bonn Large-Objects dataset was presented by Stückler et al. [50]. It contains 1533 RGB-D images for training and 500 RGB-D images for testing from 40 scenes. The images were taken with a Microsoft Kinect camera in VGA resolution (640 px × 480 px). Each image contains a densely labeled ground-truth with four classes. Training images are sliced into three partitions with 514, 507 and 512 images, respectively. Stückler et al. [50] train one tree on each partition.

The original dataset contains RGB-D images with a raw depth channel, i.e. unprocessed depth data with missing depth values as delivered by the Microsoft Kinect depth sensor. We adopt the scheme of the NYU Depth v2 dataset [45] to derive a variant of the AIS Bonn Large-Objects dataset where we reconstruct missing depth values from the neighborhood. The depth filling method is based on Levin et al. [29] and was originally used to colorize grayscale images. Missing values are estimated by depth values from the neighborhood. Neighboring pixels with a similar luminosity (i.e. the gray value) in the corresponding color channels have a higher influence. The runtime of this depth filling implementation is dominated by solving a system of linear equations (cf. Levin et al. [29]) and takes about 10 s per 640 px × 480 px image on an Intel Core i7–920.

We also use a second depth filling method as proposed by Jörg Stückler which we call “simple depth filling”. Simple depth filling runs sufficiently fast to be used for prediction in real-time applications. The computational complexity is low as the method simply iterates over an image four times and uses neighboring pixels

1. to the right,
2. to the left,
3. to the top,
4. to the bottom,

to reconstruct a missing depth value. The method guarantees that the depth channel of an image has no missing values afterwards, if and only if the original depth channel contains at least one valid depth. Filling an image in VGA resolution with a size of 640 px × 480 px takes less than 2 ms per image on an Intel Core i7–920 using a single thread.

Figure 5.1 shows three example images of the AIS Bonn Large-Objects dataset including a visualization of the raw depth channel with missing values and the depth filling by colorization according to Levin et al. [29]. Figure 5.2 demonstrates the difference between the more sophisticated depth filling by colorization and the simple depth filling by using an example of the AIS Bonn Large-Objects dataset. The example shows that simple depth filling quality is low if the raw depth channel contains large blocks of missing depth values.

The ground truth labelings of the dataset contain large background areas, depicted in black. Stückler et al. [50] propose to exclude background pixels when measuring the segmentation accuracy. When not specified otherwise, we use the pixel accuracy that excludes background pixels to report segmentation accuracy on the AIS Bonn Large-Objects dataset (cf. Section 3.3). In the average per-class accuracy, the background class is explicitly included if not stated otherwise.

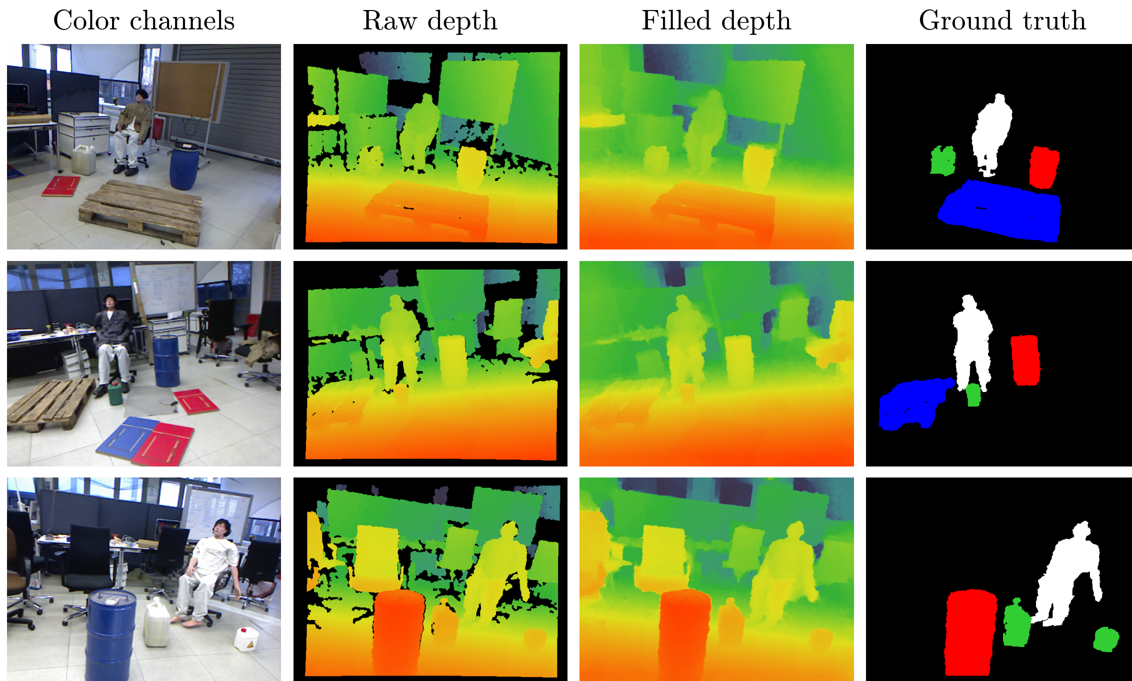

Figure 5.1.: Three RGB-D example images of the AIS Bonn Large-Objects dataset. First column: Color image channel. Second column: Visualization of the raw depth channel. Black is used to indicate missing depth information (NaN). Third column: Visualization of the depth channel filled by colorization. Fourth column: Manually created ground truth with the classes "palette" (●), "barrel" (●), "canister" (●) and "human" (○). Black indicates background.

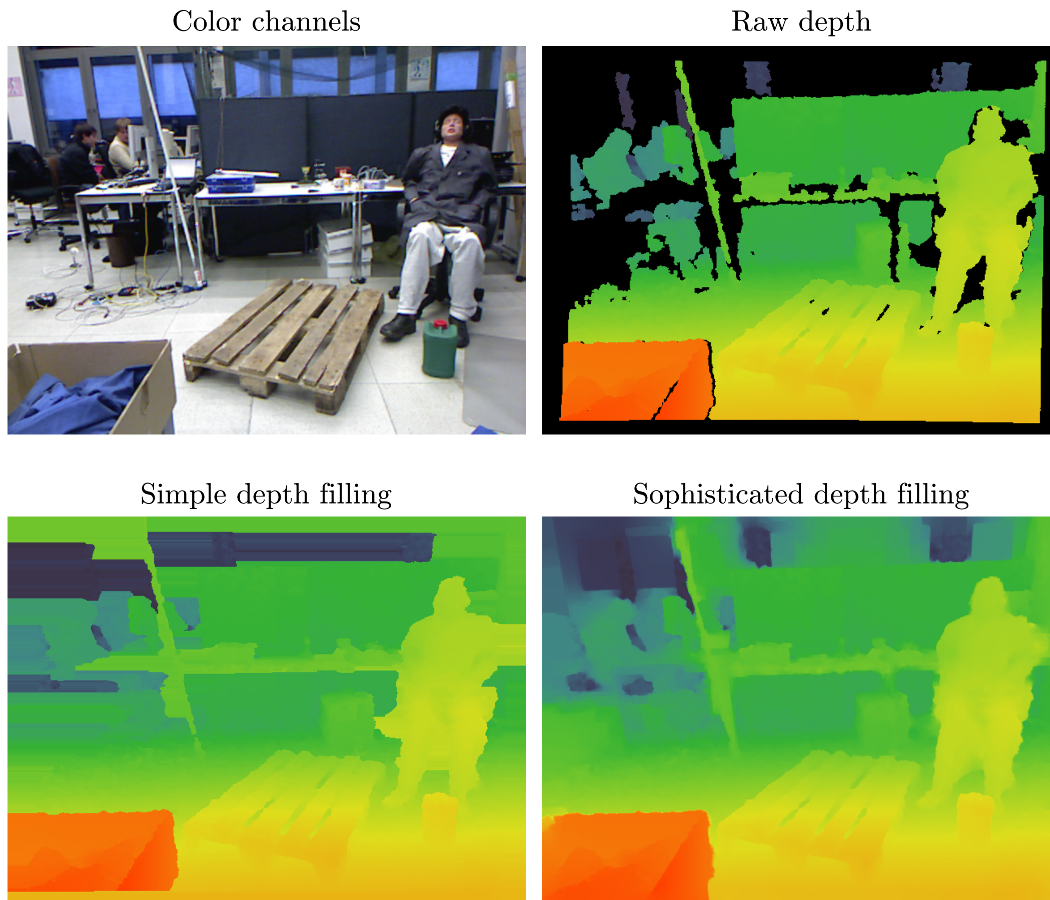

Figure 5.2.: Comparison of the two depth filling variants for an example image of the AIS Bonn Large-Objects dataset. Top left: Color channels of the RGB-D image. Top right: Visualization of the raw depth channel. Black indicates missing depth. Bottom left: Visualization of the depth channel that was filled with a simple method using neighboring pixels. Simple depth filling contains artefacts in the top and bottom left where depth information is missing in larger areas in the raw depth channel. Bottom right: Visualization of the depth channel that was filled with the more sophisticated method proposed by Silberman et al. [45].

5.1.2. NYU Depth v2 dataset

We use the NYU Depth v2 dataset, introduced by Silberman et al. [45], as the second dataset for our experiments. It contains 1449 densely labeled pairs of aligned RGB-D images from 464 indoor scenes. Silberman et al. define four semantic classes “Ground”, “Furniture”, “Props” and “Structure” that we use as class labels in our experiments. The dataset has been split into disjunct training and test sets with 795 and 654 images, respectively [46]. Figure 5.3 shows three example RGB-D images of the dataset. The dataset includes an additional set of depth images where missing values of the raw depth images have been filled using the colorization scheme proposed by Levin et al. [29]. As afore mentioned, we adopt this method for the AIS Bonn Large-Objects dataset to conduct experiments with the same depth filling scheme on both datasets.

When not specified otherwise, we use the accuracy measure that excludes “void” pixels to report segmentation accuracy on the NYU Depth v2 dataset (cf. Section 3.3).

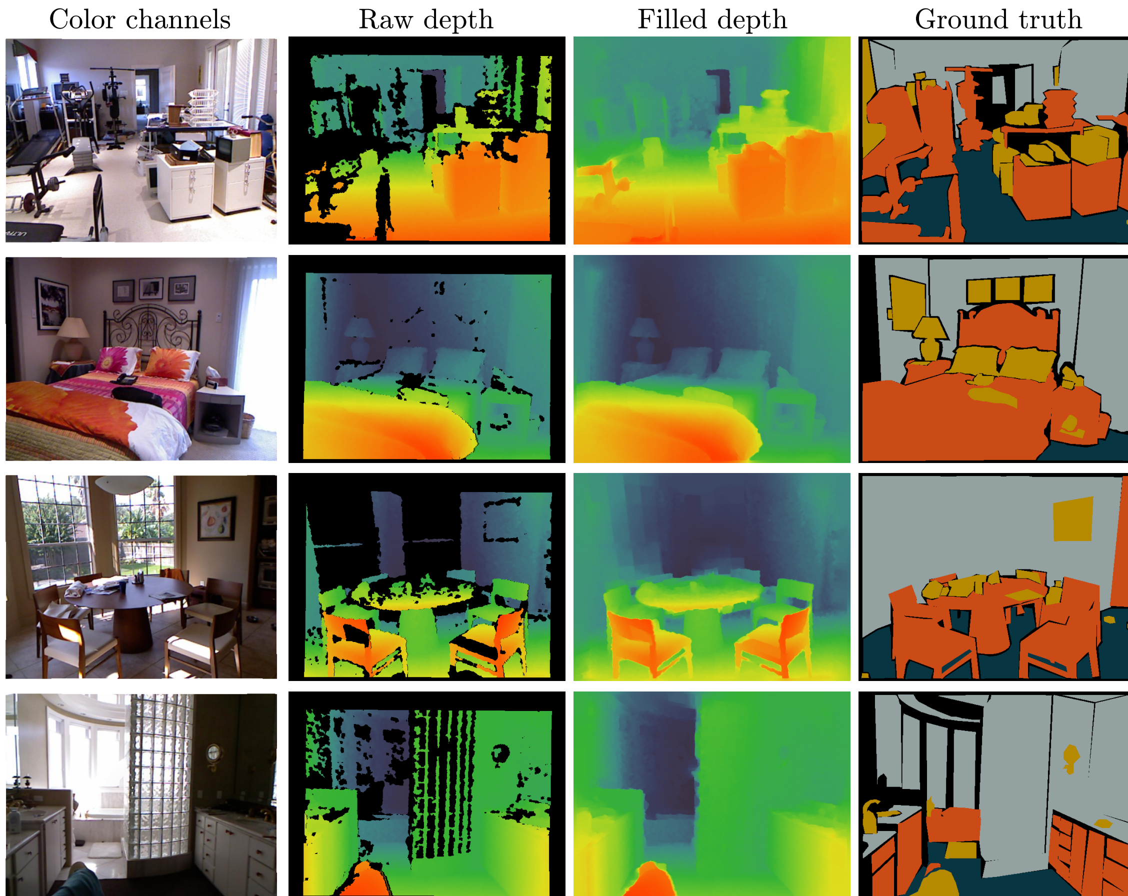

Figure 5.3.: Four RGB-D example images of the NYU Depth v2 dataset. First column: Color image channel. Second column: Visualization of the raw depth channel. Black is used to indicate missing depth information (NaN). Third column: Visualization of the filled depth channel. Fourth column: Manually created ground truth with the classes "floor" (●), "prop" (●), "furniture" (●) and "structure" (●). Black pixels indicate "void" where the class is unknown.

5.2. Parameters

Random forest training requires a set of parameters such as the number of trees to grow, the number of feature candidates to generate or the stopping criterion. The parameter values depend on the dataset and are a possible subject to an optimization method such as extensive parameter search.

To produce comparable results in our experiments with the AIS Bonn Large-Objects dataset, we adopt the parameters used by Stückler et al. [50].

For the NYU Depth v2 dataset we leverage our accelerated training implementation to train several random forests per day in order to search for an optimal set of training parameters with respect to segmentation accuracy. We use Hyperopt [4] for this optimization task, which implements an informed search over the hyper-parameter space to optimize a given loss function. We perform cross-validation with random splits on the training set and use a loss function $L(A_{\mathcal{F},\mathcal{D}})$ that depends on the measured average per-class accuracy $A_{\mathcal{F},\mathcal{D}}$ for random forest $\mathcal{F}$ on the dataset $\mathcal{D}$ and is defined by

$$L(A_{\mathcal{F},\mathcal{D}}) := 1 - A_{\mathcal{F},\mathcal{D}}. \tag{5.1}$$

We estimate the variance of the loss using the results of each fold in the cross-validation. We cancel the cross-validation earlier, after three or four folds, if it is unlikely that the current parameters will yield a lower loss than the currently lowest loss. To reject the hypothesis that the current trial can yield a lower loss, we use a Student's t-test with a significance level of $\alpha = 0.005$. Table 5.1 shows the parameter space that we use for the hyper-parameter search. Figure 5.4 depicts the loss function development during the parameter search with Hyperopt. We stop searching after 160 successful trials with a loss at about 0.34. A more extensive parameter search could be conducted in future work.

The resulting random forest training parameters for both datasets are shown in Table 5.2. If not explicitly stated otherwise, we conduct the experiments in the following section with these parameters.

Parameter	Lower bound	Upper bound
Number of trees	3	3
Samples per image	10	7500
Random feature candidates	10	7500
Threshold candidates	10	60
Minimum sample count	1	1000
Maximum tree depth [levels]	5	20
Box radius [px m]	1	127
Region size [px m]	1	127

Table 5.1.: Search space used for parameter optimization on the NYU Depth v2 dataset with Hyperopt [4].

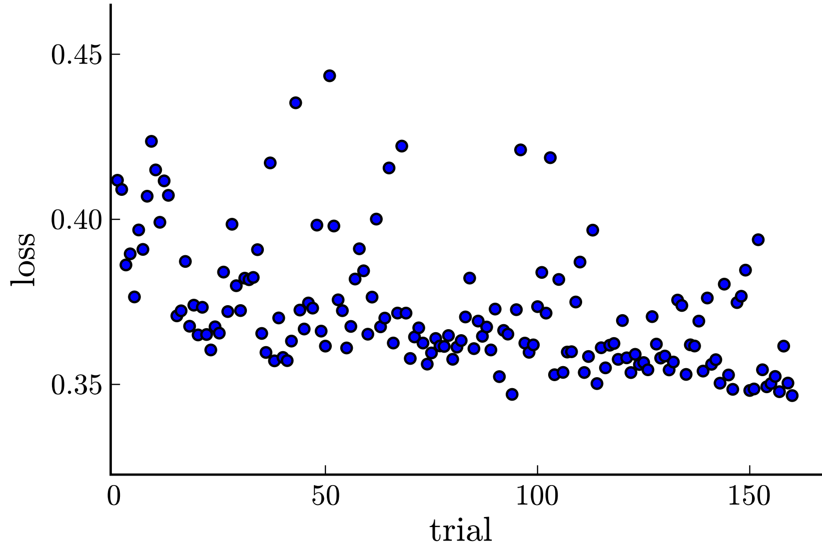

Figure 5.4.: Development of the loss function during hyper-parameter search with Hyperopt. We stop the parameter search after 160 trials.

Parameter	AIS	NYU
Number of trees	3	3
Samples per image	2000	4537
Random feature candidates	2000	5729
Threshold candidates	50	20
Minimum sample count	100	204
Maximum tree depth [levels]	15	18
Box radius [px m]	120	111
Region size [px m]	10	3
Histogram bias ρ	0.2	0

Table 5.2.: Random forest training parameters used for our experiments. Parameters for the AIS Bonn Large-Objects dataset are chosen to be consistent with Stückler et al. [50]. Parameters for the NYU Depth v2 dataset are the result of our hyper-parameter search with Hyperopt [4].

5.3. Random Feature Candidate Generation per Tree Level

Before we start with the experimental results on random forest training, we discuss a particular feature of our implementation that we use to improve training runtime for random forests with deep trees and many nodes.

The number of nodes in a decision tree grows exponentially with the depth of the decision tree. Random forests use binary decision trees such that we can estimate the average number of nodes n in a level by $n = O(2^l)$, where l is the level in the tree.

The default training strategy for a random forest is to generate a set of random feature candidates and thresholds for every node in the decision tree. We observe that deep trees with more than ten levels contain thousands of nodes. The training time per-node in the tree exponentially decreases per tree level since the total number of training samples per level remains constant or only decreases marginally and are distributed over an exponentially increasing number of nodes. Feature candidate generation scales linearly in the number of features and thresholds (cf. Section 4.2.4). As those parameters do not change while growing a tree, feature generation has a constant per-node runtime which results in a relatively large total training overhead.

Given that the number of generated random features and thresholds is sufficiently large , we assume that segmentation accuracy is not influenced significantly if we do not generate a new set of features for every node. Instead, we generate a new set of feature and thresholds candidates only once per level while a decision tree is trained. We can easily implement per-level feature generation since we train the trees in breadth-first order (cf. Section 4.2.1).

The results in Table 5.3 confirm that segmentation accuracy does not differ significantly on the AIS Bonn Large-Objects dataset, even if we generate a new set of random feature candidates only once per tree level.

Dataset	Feature Generation	Pixel accuracy [%]	Class accuracy [%]
AIS	Per-Node	64.1 ± 1.1	70.3 ± 2.0
AIS	Per-Level	63.8 ± 0.7	70.4 ± 1.2
NYU	Per-Node	68.2	65.0
NYU	Per-Level	68.1	65.1

Table 5.3.: Segmentation accuracy with feature candidate generation per-level and per-node. Class accuracy for the AIS Bonn Large-Objects dataset is calculated without the background class. There is no significant difference of segmentation accuracy on the AIS Bonn Large-Objects dataset between the two variants. We also include the segmentation accuracies for the NYU Depth v2 dataset. However, the results on the NYU Depth v2 dataset need to be interpreted carefully as we did not carry out an extra hyper-parameter search with feature candidate generation per-node.

5.4. Random Forest Training on GPU

In this section we present the experimental results of random forests training time on the two datasets AIS Bonn Large-Objects dataset and NYU Depth v2 dataset. Section 4.2 introduced our method to accelerate random forest training on CPU and GPU. Direct comparison of our results with the results of Toby Sharp [52] is not possible for three reasons. Firstly, Toby Sharp uses different visual features that are computationally less expensive as they require fewer arithmetic operations and memory accesses and need *not* to distinguish between the two feature types for color and depth information. Secondly, Toby Sharp trains on every pixel of the images of the Microsoft Research recognition database (MSRC) [59] which only contains RGB images without depth. In contrast, Stückler et al. [50] and Shotton et al. [43] use only a fraction of pixels for training. Thus, we optimize the implementation for the training with a subset of pixels per image rather than training with all pixels. Lastly, the implementation of Toby Sharp is not available and the publication does not specify all parameters that are used for training, such as the number of threshold candidates.

To estimate the computational efficiency of our implementation on GPU, we compare the runtime with our manually optimized implementation on CPU. Section 4.2.3 gave an estimate for the theoretical computing limits of the CPU. We anticipate that the potential improvement of our CPU implementation is less than one order of magnitude.

We train random forests with three trees and the parameters as stated in Section 5.2 on CPU and GPU. We measure training time on the fastest CPU that is available in our laboratory, a hexacore Intel Core i7–X980 (Extreme Edition) clocked at 3.33 GHz. We use two different high-end GPUs to measure the acceleration of random forest training. The first GPU is a NVIDIA Tesla K20c with 2496 CUDA cores, clocked at 705 MHz, 5 GB of GDDR5 memory with a peak memory bandwidth of 208 GB/s. The second GPU is a NVIDIA GeForce GTX TITAN with 2688 CUDA cores, clocked at 837 MHz, 6 GB of GDDR5 memory with a peak bandwidth of 288 GB/s.

Table 5.4 shows the training time for the AIS Bonn Large-Objects dataset and the NYU Depth v2 dataset on CPU and on both GPUs. On the GTX TITAN our GPU

	i7–X980	Tesla K20c		GTX TITAN	
Dataset	Train [min]	Train [min]	Speed-up	Train [min]	Speed-up
AIS	84	4.8	18	3.8	22
NYU	665	54.9	12	24.1	28

Table 5.4.: Comparison of random forest training time on a hexacore Intel Core i7–X980 CPU, a NVIDIA Tesla K20c and a NVIDIA GeForce GTX TITAN GPU to train a random forest with three trees on the AIS Bonn Large-Objects dataset and NYU Depth v2 dataset. Times are without loading and sampling of images.

Accelerator	Introduction	Price [$]	TDP [W]
Core i7–X980	Mar 2010	999	130
Tesla K20c	Nov 2012	3 199	225
GTX TITAN	Feb 2013	999	250

Table 5.5.: Date of introduction, price at introduction and TDP for the Intel Core i7–X980 CPU, the NVIDIA Tesla K20c and the NVIDIA GeForce GTX TITAN GPU that were used to train a random forest.

implementation trains a random forest 22 times faster for the AIS Bonn Large-Objects dataset and 28 times faster for the NYU Depth v2 dataset.

Speed-up factors are the state-of-the-art measure to assess the quality of a GPU implementation in current scientific publications. The speed-up factor highly depends on the two hardware platforms in comparison and the quality of the CPU implementation. To make the comparison as fair as possible, we use the strongest CPU available in our laboratory and manually optimize and parallelize our CPU implementation to make efficient use of the CPU resources (cf. Section 4.2.2). A speed-up of factor 10, for instance, would not justify the effort to implement the algorithm on GPU if the acquisition costs or the power consumption of a GPU are 10 times as high. Such an implementation would be inefficient from an economical point of view. Thus, in Section 3.4 we introduced two additional measures that normalize the speed-up factor over acquisition costs and over power consumption as proposed by Van Essen et al. [53].

Table 5.5 depicts the date of introduction, price at introduction and the TDP for the CPU and the two GPUs that we use for our experiments on the acceleration of random forest training.

The price at introduction of the GeForce GTX TITAN equals the price at introduction of the Intel Core i7–X980. Thus, the speed-up factor normalized over acquisition costs remains the same for the GeForce GTX TITAN. The Tesla K20c had a price at introduction of $3199 such that the speed-up factor of 18 and 12 for the AIS Bonn Large-Objects dataset and NYU Depth v2 dataset, respectively, is normalized to $18 \cdot \frac{\$999}{\$3199} \approx 5.6$ and $12 \cdot \frac{\$999}{\$3199} \approx 3.7$, respectively.

Van Essen et al. use the TDP to normalize the speed-up over the power consumption. We apply the idea on our results. Table 5.6 depicts the comparison of power consumption of the CPU and the two GPUs. The TDP of the two GPUs is about twice as high as

	i7–X980	Tesla K20c		GTX TITAN	
Dataset	PC [Wh]	PC [Wh]	Factor	PC [Wh]	Factor
AIS	182	17.9	10	15.7	12
NYU	1441	205.8	7	100.4	14

Table 5.6.: Comparison of power consumption (PC) for random forest training on a hexa-core Intel Core i7–X980 CPU, a NVIDIA Tesla K20c and a NVIDIA GeForce GTX TITAN GPU. Power consumption is estimated as the factor of training time from Table 5.4 and the TDP as depicted in Table 5.5.

the TDP of the Intel Core i7–X980. Thus, the speed-up factors reported in Table 5.4 are normalized by about a factor of two. The NVIDIA system managment interface tool (`nvidia-smi`) reports the true power consumption of the Tesla K20c. We measure an average power draw of 109.7 W while training the AIS Bonn Large-Objects dataset and an average power draw of 63.8 W while training the NYU Depth v2 dataset. The power consumption of training with the NYU Depth v2 dataset is significantly lower since the GPU is more often idle when images are transferred from host to device. The measurement indicates that the TDP of 225 W is at least a factor of two higher than the actual power draw on the Tesla K20c. Unfortunately, the power draw metric is not available on the GTX TITAN and it is not possible to measure the power draw on current CPUs. Thus, we can only use this information to estimate the magnitude of uncertainty for the reported speed-up factors over power consumption in Table 5.6, which is at least a factor of two. Nevertheless, we conclude that our implementation on GPU significantly reduces time and costs of random forest training.

5.4.1. Breakdown of Random Forest Training Time

Figure 5.5 depicts the breakdown of time spent for training per level in the decision trees on the AIS Bonn Large-Objects dataset with filled depth and the NYU Depth v2 dataset. The vast majority of training is spent for the evaluation of the best split criterion as depicted with blue circles. The plot shows the time for the feature response calculation, histogram aggregation, impurity score calculation and the transfer of images between host and device.

There is a small gap between the cumulated time of the three training phases and the total split evaluation time (blue circles). This gap represents the overhead that is necessary to maintain data structures on the CPU and the time to generate feature candidates once per level.

In the following sections we evaluate the results with respect to the various training phases.

Feature Response Calculation

The plot shows constant feature response calculation time per level (green area). Figure 5.6 illustrates a more detailed analysis of the feature response calculation phase. We see that feature response calculation on the GPU scales almost linearly in the number of samples,

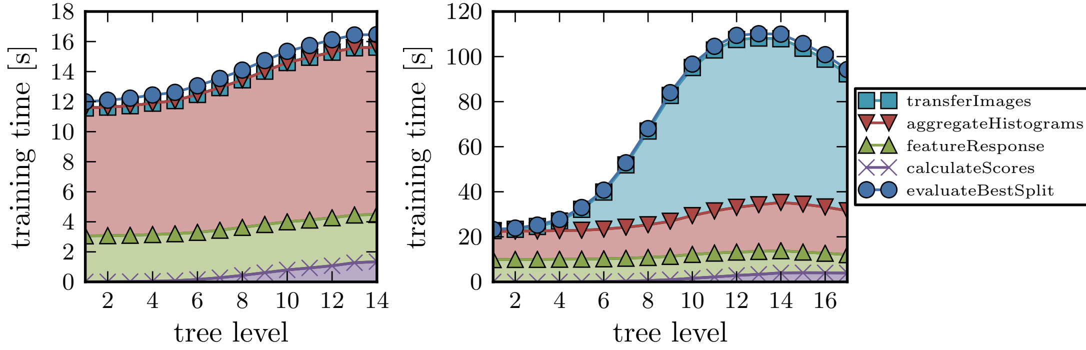

Figure 5.5.: Breakdown of random forest training time per tree level for the various training phases. Measured on a NVIDIA GeForce GTX Titan. Left: Random forest trained on AIS Bonn Large-Objects dataset with filled depth. Right: Random forest trained on NYU Depth v2 dataset.

which is the expected behavior of an ideal implementation. Note that the plot on the right side shows a cutoff below 200 samples. The reason is that we set a minimum of 204 samples per node as stopping criterion. Still, there are a few measurements with less than 200 samples. The reason is that we process the feature response calculation in slices (cf. Section 4.2.4). In seldom cases, the number of samples can be smaller than the configured minimum, if it happens to be the remainder of the slices.

Histogram Aggregation

Figure 5.5 indicates that histogram aggregation (red area) scales linearly on the levels lower than five. However, the total histogram aggregation time per level increases on levels above five. The analysis on training time with respect to the number of samples as depicted in Figure 5.7 confirms that histogram aggregation scales almost linearly with many samples. However, a constant runtime overhead is clearly visible for less than 10^3 samples. Note that both axes in the figure are in logarithmic scale. The constant overhead is caused by the sum reduction which is the final step of the histogram aggregation phase (cf. Section 4.2.4). Levels above five have an increasing amount of nodes which are small and contain less than 1000 samples. Since histogram aggregation gets less efficient for such nodes, the total histogram aggregation time per level increases.

Impurity Score Calculation

We use the normalized information gain in double precision as impurity score function. Runtime does not depend on the number of samples but on the number of feature and threshold candidates. Since these numbers are constant throughout training, we measure an average per-node time for impurity score calculation of 1.57 ± 0.25 ms on the AIS Bonn Large-Objects dataset (cf. Section 4.2.4). The total time per level increases for deep trees as the number of nodes per level increases exponentially.

Image Transfers

Figure 5.5 depicts the accumulated time per level in the tree, that is spent to transfer the RGB-D images from host to device, depicted in light blue. It explains the reason for the

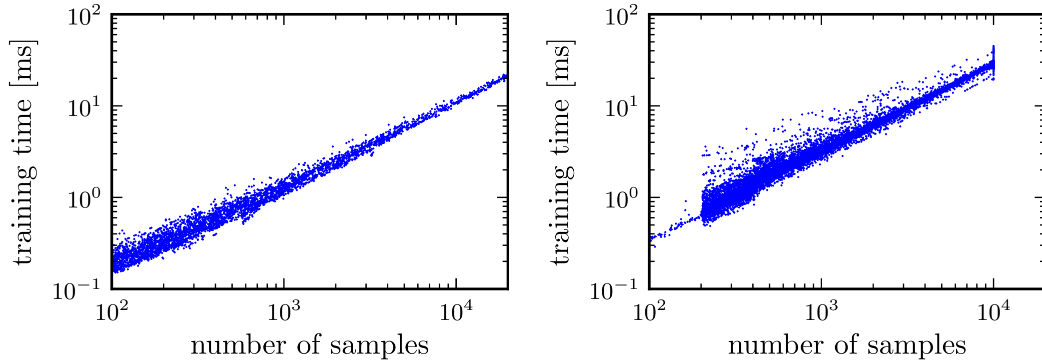

Figure 5.6.: Feature response calculation time with respect to the number of samples. Measured on a NVIDIA GeForce GTX Titan. Left: Random forest trained on AIS Bonn Large-Objects dataset with filled depth. Right: Random forest trained on NYU Depth v2 dataset. Feature response calculation scales almost linearly on both datasets.

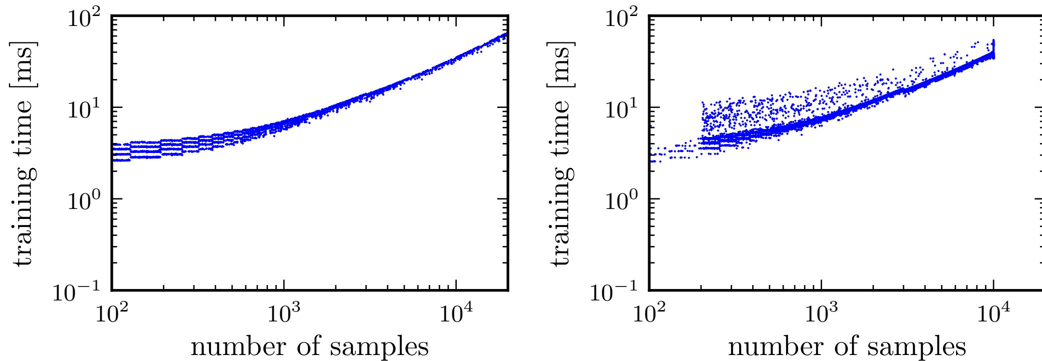

Figure 5.7.: Histogram aggregation time with respect to the number of samples. Measured on a NVIDIA GeForce GTX Titan. Left: Random forest trained on AIS Bonn Large-Objects dataset with filled depth. Right: Random forest trained on NYU Depth v2 dataset. Histogram aggregation time scales almost linearly for more than 10^3 samples. We observe a constant overhead for sum reduction (cf. Section 4.2.4) that causes non-linear scaling in the range between 10^2 and 10^3 samples.

large runtime difference between training the AIS Bonn Large-Objects dataset and the NYU Depth v2 dataset. We notice that the total time to transfer images is about zero throughout training the AIS Bonn Large-Objects dataset. This is because the two tested GPUs have a sufficient amount of memory to hold all training images. The right plot shows the training of the NYU Depth v2 dataset. The relative overhead of image transfers starts to increase at tree level five and dominates training on deeper levels. The reason is that the 6 GB of main memory on the GeForce GTX TITAN is not sufficient to hold all 795 images of the NYU Depth v2 dataset in memory. Instead, we set the image cache size to 670 images. The feature response calculation phase of the best split evaluation requires to have all images in device memory that belong to the training samples in the particular node. Unfortunately, we observe that the training samples distribute over a large fraction of images. Thus, we need to transfer a large fraction of images for the split evaluation for *every* node in the tree. Since the number of nodes increases with the tree level, the total time of image transfers increases with the number of levels and makes training less efficient, especially for deep trees. The Tesla K20c GPU is equipped with 5 GB of memory and thus can only hold a smaller amount of images when training the NYU Depth v2 dataset. This explains that training time is more than twice as long as on the GeForce GTX TITAN.

We use different CUDA streams (cf. Section 3.5.2) to reduce image cache transfer latency. This is possible since the GPU only accesses images during the feature response calculation. While the histogram aggregation and impurity score calculation is running on the GPU, the images for the next feature response calculation can be already transferred to the GPU. Hiding the latency is only partially possible on levels above five since the time spent for histogram aggregation is significantly lower than the time to transfer images as depicted in Figure 5.5. Nevertheless, our experiments show that image transfer latency can entirely be hidden on the upper levels if the cache size is sufficiently large.

Random forest training on the CPU is affected even more by the large size of the NYU Depth v2 dataset. We measure a cache miss rate of 30.1 % and 13.8 % on the NYU Depth v2 dataset and AIS Bonn Large-Objects dataset, respectively, when training a single tree with two levels on the CPU. The higher cache miss rate on the NYU Depth v2 dataset causes the CPU to transfer data from main memory to L2 cache and L1 cache more often which slows down the training.

5.4.2. Discussion of the Results

We presented an implementation that trains random forests efficiently on GPU. We achieve a speed-up factor of up to 22 on the AIS Bonn Large-Objects dataset and 28 on the NYU Depth v2 dataset. The measurements show that even if we normalize the speed-up factor over power consumption or acquisition costs, the random forest training implementation on GPU clearly saves time and costs.

Feature response calculation and histogram aggregation amount for the majority of training time with 44.1 s (22.4 %) and 136.7 s (69.4 %), respectively, in case of the AIS Bonn Large-Objects dataset. It scales linearly in the number of samples while histogram aggregation scales linearly with many samples but gets less efficient with fewer samples. Hence, our training implementation efficiently scales if more training instances (pixels) are sampled from the dataset or the user configures to evaluate more random feature

candidates.

Toby Sharp measures only about 2 % for feature response calculation while histogram aggregation amounts for 96 % of total training time [52, Figure 9]. We assume three causes for these large differences. Firstly, Toby Sharp implements features that have a lower computational complexity than ours (cf. Section 3.2). Secondly, he samples every pixel of the training images which presumably yields a higher cache hit rate. Lastly, he implements the histogram aggregation with DIRECT3D using a vertex shader but states that he anticipates a significant benefit by using CUDA instead.

5.5. Random Forest Prediction on GPU

In this section we present and discuss the experimental results of our implementation for random forest prediction as introduced in Section 4.3. Prediction in the scope of this master's thesis means a dense pixel-wise classification of an image.

Table 5.7 compares the prediction time for a 640 px × 480 px image on a Intel Core i7–X980 CPU to the prediction time on a NVIDIA GeForce GTX 675M mobile GPU and Tesla K20c GPU. For the measurements, we separate image loading and pre-processing, such as image integral calculation, from the actual pixel-wise classification. The timings do not include loading and pre-processing of the images since loading and integral calculation is only implemented on CPU. We use five threads to integrate the five channels of an RGB-D image in parallel which takes 4.8 ms on an Intel Core i7–920 and 3.9 ms on an Intel Core i7–X980.

The prediction runtime depends on a large variety of parameters such as the number of trees, depth of the decision trees, size of the images, complexity and choice of the implemented features, number of classes, optimizations of the compiler and the specific hardware. For the AIS Bonn Large-Objects dataset and the NYU Depth v2 dataset, we measure a CPU-GPU speed-up factor of up to 33, depending on the GPU.

5.5.1. Discussion of the Results

We presented an implementation for random forest prediction that performs dense pixel-wise classification of an image in VGA resolution in real-time speed on a GeForce GTX 675M, which is a mobile GPU available for notebooks.

Toby Sharp measures a speed-up factor of about 100 on a NVIDIA GeForce GTX 280

	i7–X980	GeForce GTX 675M		Tesla K20c	
Dataset	Pred. [ms]	Pred. [ms]	Speed-up	Pred. [ms]	Speed-up
AIS	343	35.1	10	10.3	33
NYU	393	44.4	9	13.9	28

Table 5.7.: Comparison of random forest prediction time on an hexacore Intel Core i7–X980 CPU, a NVIDIA GeForce GTX 675M mobile GPU and a Tesla K20c GPU. We use random forests with three trees trained on the AIS Bonn Large-Objects dataset and NYU Depth v2 dataset.

compared to an Intel Core 2 Duo [52, Figure 10]. He measures classification time of about 23.1 ms for a 640 px × 480 px image with eight trees on the GPU. In contrast to our setup, Toby Sharp evaluates on a dataset with RGB images that do not contain a depth channel. The trees are only trained to a depth of eight levels (256 nodes). Furthermore, the feature implementation has a significantly lower computational complexity which explains the difference to our results.

5.6. Segmentation Accuracy

Despite that the acceleration of random forests is the main focus of this master's thesis, we also measure segmentation accuracy on the two datasets. Our fast implementation allows us to train and predict with both datasets and many variants. We present the measured segmentation accuracies in this section.

5.6.1. Information Gain versus Normalized Information Gain

In Section 3.1.1 we introduced the two impurity score functions information gain and normalized information gain where we did not measure which function leads to a higher segmentation accuracy.

We train five random forests with different random seeds to measure the average segmentation accuracy and the according standard deviation for both impurity score functions. Table 5.8 shows the resulting comparison of a random forests trained with information gain score and a random forests trained with normalized information gain score. Training with normalized information gain score does only marginally run slower than training with information gain score. However, random forests trained with normalized information gain show a significant increase of average pixel accuracy. We also observe a different shape of the random forest when trained with the normalized information gain instead of the information gain. The random forest trained on the AIS Bonn Large-Objects dataset with normalized information gain contains 4361 decision nodes and 2802 leaf nodes. The random forest trained with information gain is significantly larger and contains 5831 decision nodes and 3670 leaf nodes.

Score Function	Training [s]	Pixel accuracy [%]	Class accuracy [%]
Information gain	195.0	61.9 ± 0.3	71.4 ± 1.4
Normalized information gain	196.1	64.2 ± 0.5	72.5 ± 0.8

Table 5.8.: Comparison of random forest training time and segmentation accuracy with respect to the score function used for training. Random forest trained on the AIS Bonn Large-Objects dataset with raw depth. Note that we do not include results on the NYU Depth v2 dataset as it would require to re-evaluate the hyper parameters.

5.6.2. Depth Filling

In Section 5.1 we presented two different depth filling methods. We measure the influence of these methods on the segmentation accuracy by training five random forests with different seeds for each of the three depth filling variants and for both datasets. Segmentation accuracy is measured for every random forest in combination with the three depth filling variants applied on the test set. Thus, there are nine combinations of depth filling on the training and test set. The results are depicted in Table 5.9 and Table 5.10 for the AIS Bonn Large-Objects dataset and the NYU Depth v2 dataset respectively. On the AIS Bonn Large-Objects dataset we achieve the highest segmentation accuracy when the random forest is trained with simple depth filling and tested on images that were pre-processed with the more sophisticated depth filling. The results with the NYU Depth v2 dataset (Table 5.10) have more reliable evidence as the standard deviations are significantly lower than the results with the AIS Bonn Large-Objects dataset. We see that depth filling significantly improves the segmentation accuracy. The more sophisticated depth filling yields the best results but only if the random forest was also trained with images that were pre-processed with that depth filling variant.

We observe particularly low segmentation accuracies and high standard deviations if the random forests are trained with filled depth images but tested with raw depth images. The feature response for pixels in the image that have no valid depth is NaN since the offset and region size cannot be normalized in this case (cf. Section 4.1). Such pixels always traverse to the right-most leaf node in the random decision trees as the comparison with the threshold always yields false. Hence, it is crucial that pixels without valid depth exist in the training set so that the path in the tree to the right-most leaf node is constructed appropriately. Otherwise, pixels without valid depth are assigned to the class of the right-most leaf node which is as if a class label is assigned randomly. This explains the low segmentation accuracy and high standard deviation for the combination of training with depth filling and testing with raw depth.

Figure 5.8 shows an example for the prediction of an image in the NYU Depth v2 dataset. The example shows that misclassifications are more frequent if depth information is missing. We conclude that depth filling is an important pre-processing step that improves segmentation accuracy significantly.

Depth filling			
Training	Prediction	Pixel accuracy [%]	Class accuracy [%]
Raw	Raw	**64.2 ± 0.5**	72.5 ± 0.8
Raw	Simple	56.5 ± 0.3	72.0 ± 0.7
Raw	Sophisticated	60.2 ± 0.5	73.5 ± 0.8
Simple	Raw	61.3 ± 1.5	70.5 ± 0.8
Simple	Simple	62.4 ± 0.6	75.7 ± 0.8
Simple	Sophisticated	**64.2 ± 0.5**	**76.5 ± 0.8**
Sophisticated	Raw	59.3 ± 1.3	67.1 ± 0.8
Sophisticated	Simple	60.0 ± 0.6	73.0 ± 0.3
Sophisticated	Sophisticated	63.7 ± 0.5	74.5 ± 0.1

Table 5.9.: Comparison of segmentation accuracy for all combinations of training and prediction with the three depth filling variants on the AIS Bonn Large-Objects dataset.

Depth filling			
Training	Prediction	Pixel accuracy [%]	Class accuracy [%]
Raw	Raw	64.2 ± 1.2	60.3 ± 0.6
Raw	Simple	65.3 ± 0.3	62.6 ± 0.3
Raw	Sophisticated	66.0 ± 0.1	62.0 ± 0.2
Simple	Raw	58.7 ± 3.5	56.6 ± 2.0
Simple	Simple	67.7 ± 0.0	64.7 ± 0.0
Simple	Sophisticated	67.3 ± 0.0	63.7 ± 0.0
Sophisticated	Raw	63.6 ± 0.2	58.9 ± 0.3
Sophisticated	Simple	66.9 ± 0.1	64.1 ± 0.1
Sophisticated	Sophisticated	**68.1 ± 0.0**	**65.1 ± 0.1**

Table 5.10.: Comparison of segmentation accuracy for all combinations of training and prediction with the three depth filling variants on the NYU Depth v2 dataset.

Figure 5.8.: Example of random forest predictions with the raw depth and filled depth version of the NYU Depth v2 dataset. Top left: Color channels. Top right: Ground truth. Middle left: Visualization of the raw depth channel. Black color indicates missing depth. Middle right: Visualization of the filled depth channel. Bottom left: Prediction with raw depth (Pixel accuracy: 87.0 %). Bottom right: Prediction with filled depth (Pixel accuracy: 91.4 %). Black in the bottom row indicates "void" in the ground truth. Raw depth causes more frequent misclassifications in areas where depth information is missing.

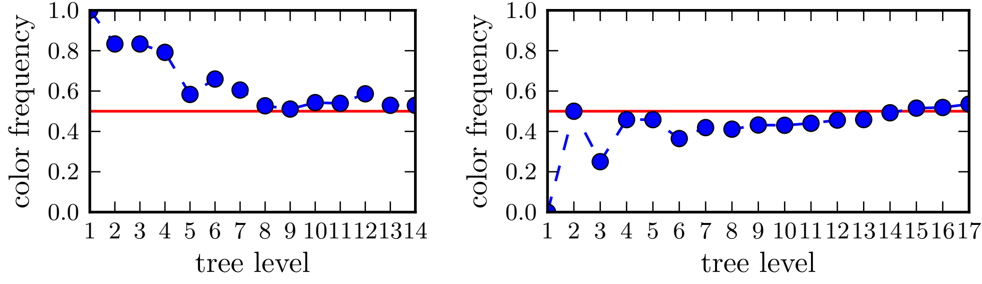

Figure 5.9.: Relative color feature frequency per tree level. Left: AIS Bonn Large-Objects dataset. Right: NYU Depth v2 dataset. The relative frequency of the depth feature f_d equals to $1 - f_c(l)$ where f_c is the relative color feature frequency in the tree level l. Color features are selected more frequently for the AIS Bonn Large-Objects dataset while depth features are selected more frequently for the NYU Depth v2 dataset.

Dataset	Color space	Pixel accuracy [%]	Class accuracy [%]
AIS	RGB	61.4 ± 1.1	71.8 ± 0.8
AIS	CIE Lab	63.7 ± 0.5	74.5 ± 0.1
NYU	RGB	67.7 ± 0.0	64.7 ± 0.0
NYU	CIE Lab	68.1 ± 0.0	65.1 ± 0.1

Table 5.11.: Comparison of segmentation accuracy for images in RGB and CIE Lab color space. We use random forests trained on the AIS Bonn Large-Objects dataset with filled depth and the NYU Depth v2 dataset.

5.6.3. RGB versus CIE Lab

In Section 3.2.2 we stated that the color image is transformed to CIE Lab color space in a pre-processing step after loading the image. We conduct an experiment to measure how segmentation accuracy compares between the CIE Lab and RGB versions of the two datasets. We train five random forests to measure average segmentation accuracies and standard deviations for each variant on both datasets. The results depicted in Table 5.11 show that the conversion to CIE Lab increases the segmentation accuracy for both datasets significantly. Segmentation accuracy measured with the AIS Bonn Large-Objects dataset is more affected than the NYU Depth v2 dataset. Figure 5.9 shows that color features are more frequent for the AIS Bonn Large-Objects dataset while the depth feature is more frequent for the NYU Depth v2 dataset. We assume this to be the reason why a change of the color representation has a larger effect on segmentation accuracy for the AIS Bonn Large-Objects dataset.

5.6.4. Segmentation Accuracy on the NYU Depth v2 dataset

Table 5.12 depicts the segmentation accuracy measured with the random forest that we trained with the optimized parameters as reported in Table 5.2. Our random forest outper-

Method	Pixel accuracy [%]	Class accuracy [%]
Silberman et al. [45]	59.6	58.6
Couprie et al. [6]	63.5	64.5
Our Random Forest	**68.1**	**65.1**

Table 5.12.: Comparison of segmentation accuracies with the NYU Depth v2 dataset. We compare the pixel accuracy and average per-class accuracy of the random forest with state-of-the-art methods of Silberman et al. [45] and Couprie et al. [6].

forms state-of-the-art methods of Silberman et al. [45] and Couprie et al. [6] with respect to average pixel accuracy as well as average per-class accuracy.

5.6.5. Discussion of the Results

The fast implementation for random forest training and prediction enables us to quickly train and test random forests with a variety of different pre-processing steps. In Section 5.6.1 we saw with the AIS Bonn Large-Objects dataset that the normalized information gain yields a higher segmentation accuracy than the information gain. Future work could further evaluate the effect of the two different impurity score functions on other datasets and in combination with the change of other training parameters. Section 5.6.1 showed that depth filling improves segmentation accuracy and is a pre-processing step that should be applied when training from RGB-D images that can contain missing depth information. The results of Section 5.6.3 showed that the transformation from RGB to CIE Lab color space significantly increase segmentation accuracy. Section 5.6.4 showed that our random forest with optimized parameters outperforms the segmentation accuracy of state-of-the-art methods. The extensive hyper-parameter search with Hyperopt [4] was feasible because of the accelerated random forest implementation.

5.7. Random Fern Prediction on GPU

In Section 4.4.2 we described our implementation to accelerate random fern prediction on GPU. It is similar to the prediction of random forests but we use two textures for the left child node indices and the leaf node histograms. The list of feature parameters is stored in constant memory on GPU.

In Table 5.13 we present the prediction time and the segmentation accuracy of random ferns in comparison to a random forest on the AIS Bonn Large-Objects dataset. We evaluate the prediction on a NVIDIA GeForce GTX 675M mobile GPU. The results show that prediction of random ferns runs about 25 % faster. However, segmentation accuracy of the random fern with 15 levels is clearly inferior to a random forest trained with the same parameters. We observe that the training of deeper ferns improves segmentation accuracy accuracy. However, since the prediction scales linearly with the number of trees and the depth of the tree, the prediction of a random fern with 22 levels runs slower than the prediction of a random forest with 15 levels while the random forest achieves the best segmentation accuracy.

Dataset	Method	Depth	Pred. [ms]	Pixel accuracy [%]	Class accuracy [%]
AIS	Ferns	15	**25.8**	53.0	61.9
AIS	Ferns	22	38.0	60.9	69.0
AIS	Forest	15	35.1	**62.9**	**70.4**
NYU	Ferns	18	**27.3**	66.0	61.1
NYU	Ferns	25	33.6	66.5	61.6
NYU	Forest	18	44.4	**68.1**	**65.1**

Table 5.13.: Comparison of random fern and random forests average prediction runtime and segmentation accuracy with the AIS Bonn Large-Objects dataset and the NYU Depth v2 dataset.

Similar results are observed for the NYU Depth v2 dataset. Prediction runtime of a random fern is about 40 % faster compared to a random forest with the same parameters. As for the AIS Bonn Large-Objects dataset, the segmentation accuracy is significantly lower, even if we train the fern with 25 levels.

5.7.1. Discussion of the Results

In this section we presented the experimental results of random fern prediction. The results indicate that our implementation classifies an image up to about 40 % faster with random ferns. However, our random ferns yield a significantly lower segmentation accuracy that we partially compensate by training deeper ferns. The original speed advantage is thereby vastly countervailed. We expect that the segmentation accuracy of random ferns can be increased by improving our method to train random ferns. Furthermore, random fern training parameters could be optimized with an extensive parameter search in future work by conducting a hyper-parameter search with Hyperopt [4]. Nevertheless, we do not anticipate speed-ups of more than 50 % since the results indicate a large fraction of prediction runtime to be spent in the feature response calculation of our visual features.

6. Conclusion

This master's thesis addresses the acceleration of random forests for object-class segmentation of RGB-D images. Popularity of RGB-D datasets is rising since cameras with depth sensor such as the Microsoft Kinect or the Asus Xtion PRO LIVE have become available at low price.

Training of random forests requires a lot of computing power if the dataset is large. Training is infeasible if computing power is not available, too costly or requires too much time. Thus, academic institutes without availability of large computing clusters have not been able to conduct extensive experiments with random forests on large RGB-D datasets.

In this master's thesis we present an implementation of random forests which accelerates training and prediction on CPU and GPU. We have implemented the visual features as proposed by Stückler et al. [50]. Feature responses for each pixel are calculated on-the-fly during training and prediction as the difference of two rectangular regions in the respective neighborhood of each pixel. Offset and size of the regions are normalized by the depth information in the RGB-D image.

The experimental results in Chapter 5 show that the time for random forest training is dominated by the evaluation of the best split criterion for the nodes in the decision trees. This evaluation is accelerated with the CUDA framework by using the massively parallel computing power of GPUs. The GPU implementation is designed to efficiently utilize GPU memory by using efficient access patterns, textures and shared memory.

Random forest prediction is accelerated by mapping the random forest data structure to a texture in GPU memory. This technique was inspired by Toby Sharp [52] and is used for real-time human body part detection from single depth images in the Microsoft Xbox gaming platform [43]. In contrast to their implementation, we use visual features with a significantly higher computational complexity that combine color and depth information and use image region sums instead of single pixel differences.

We evaluate the implementation on two RGB-D datasets, namely the AIS Bonn Large-Objects dataset [50] with 1533 images from 40 scenes and the NYU Depth v2 dataset [45] with 1449 images from 464 scenes. Random forest training runs up to 28 times faster on GPU compared to our optimized CPU implementation. Our experimental results also indicate that the power consumption is significantly lower on GPU which makes the acceleration on GPU attractive from an economical point of view.

We train a random forest on the AIS Bonn Large-Objects dataset in less than four minutes with a single GPU which previously took about one day on a CPU. Our fast training implementation has allowed us to perform an informed search with Hyperopt [4] in the space of training parameters in order to optimize segmentation accuracy on the NYU Depth v2 dataset. A random forest trained with accordingly optimized parameters outperforms the segmentation accuracy of state-of-the-art methods on the NYU Depth v2 dataset proposed by Silberman et al. [45] and Couprie et al. [6]. We achieve an average pixel accuracy of 68.1 % and an average per-class accuracy of 65.1 % while Couprie et al.

[6] reports 63.5 % and 64.5 %, respectively.

Random forest prediction runs in real-time speed on a single mobile GPU. Dense pixel-wise classification of an image in VGA resolution takes in average about 35 ms and 45 ms with the AIS Bonn Large-Objects dataset and NYU Depth v2 dataset, respectively.

We have also developed an implementation for training and prediction of random ferns. Random ferns are a specialization of random forests which use the same split criterion in each level of a decision tree. This is an advantageous property that we use to further speed-up prediction. The experimental results show a reduction of prediction time by up to 40 %. The disadvantage is a lower segmentation accuracy of random ferns in comparison to the accuracy achieved by random forests with the same training parameters. The lowered accuracy is partially compensated, for instance, by training deeper ferns. However, this also slows down prediction and countervails the original speed advantage of random fern prediction. One reason for the low prediction speed-up of random ferns is presumably the large fraction of runtime that is spent for feature response calculation which remains unchanged by using ferns instead of forests.

Our accelerated implementation forms a basis for scientific progress on computer vision applications that use random forests. We have shown that our implementation saves costs and time, a prerequisite for economically efficient research in the future such as the exploration of improved visual features.

Our random forest implementation CURFIL [56] is published on GitHub as open source software under the MIT license. Source code and documentation including examples can be downloaded from

`https://github.com/deeplearningais/curfil`.

To the best of my knowledge, this is the first random forest implementation of its kind that is publicly available.

6.1. Future Work

As future work, we expect that our implementation is used to evaluate random forests on other RGB and RGB-D image datasets. The fast implementation can be used to quickly conduct a hyper-parameter search even if only a few GPUs are available. Random forests with optimized parameters can achieve better results than the currently leading methods as we have seen in the case of the NYU Depth v2 dataset.

An interesting question is whether more sophisticated visual features or additional image channels help to increase the segmentation accuracy. Additional image channels could be, for instance, the result of filters that are applied on the original image such as an edge detection or a histogram of gradients.

Furthermore we expect that segmentation accuracy can be further improved by using the probabilistic output of the random forest prediction as input for a subsequently applied machine learning method.

Our implementation certainly enables efficient research that has not been feasible before.

`:wq!`

Glossary

ALU Arithemtic Logical Unit. 19

CART Classification And Regression Tree. 7

CIE Lab CIE Lab (CIELAB) is a color space specified by the International Commission on Illumination (French: Commission internationale de l'éclairage, hence CIE). The three coordinates represent lightness (L) and the color as a position (a, b) in a two-dimensional space. In contrast to RGB, CIELAB can represent all colors that a human eye can perceive and is specified device-independently. v, 14, 16, 49, 74, 75

CPU Central Processing Unit. 1–3, 5, 11, 18, 19, 22, 27–35, 38, 42, 48–50, 52, 54, 63–65, 68, 69, 77, 79

CUDA NVIDIA's general purpose parallel computing platform and programming model implemented by GPUs (formerly: Compute Unified Device Architecture). 3, 5, 18–22, 24, 25, 32–34, 42, 45, 49, 63, 68, 69, 77, 79

FPGA Field Programmable Gate Array. 3

GCC The GNU Compiler Collection. 31

GPU Graphics Processing Unit. 1–3, 5, 11, 18–20, 22–28, 32–39, 42, 44–50, 52–54, 63–65, 68–70, 75, 77–79

HLSL High Level Shader Language. 3

HOG Feature descriptors used in computer vision for the purpose of object detection. 3

L1 cache First-level cache of RAM on CPUs and GPUs. L1 caches are usually faster but smaller than L2 caches. 21, 22, 68

L2 cache Second-level cache of RAM on CPUs and GPUs. L2 caches reside logically between L1 and RAM or between L1 and L3 if an L3 cache is available. 21, 22, 38–41, 49, 50, 68, 87

LRU Least Recently Used. 47

MSRC Microsoft Research recognition database. 63

NaN Not a Number. 15, 28, 56, 59, 71

NVCC The CUDA C/C++ Compiler. 32, 34

PTX Parallel Thread eXecution. 19, 20, 22

RGB Additive color model in which the three channels red, green, and blue light are added together. The name comes from the initials of red, green, and blue. v, 3, 16, 17, 52, 63, 70, 74, 75, 78, 80

RGB-D RGB image with an additional channel that contains depth information per pixel. The depth represents the distance to the visible object. iii, vi, 2, 3, 13–15, 47, 50, 54, 56–59, 66, 69, 75, 77, 78

SDK Software Development Kit. 25

SIMD Single Instruction, Multiple Data. 20

SIMT Single Instruction, Multiple Threads. 20

TDIDT Top-Down Induction of Decision Trees. 6

TDP Thermal Design Power. 64, 65

VGA resolution A standardized resolution with a width and height of $640\,\text{px} \times 480\,\text{px}$. iii, 2, 54, 55, 69, 78

Weka Data Mining Software Package in Java [20]. 3, 4

Bibliography

[1] Ashwin M Aji, Mayank Daga, and Wu-chun Feng. “Bounding the effect of partition camping in GPU kernels”. In: *Proceedings of the International Conference on Computing Frontiers*. ACM. 2011, p. 27. ISBN: 978-1-4503-0698-0. DOI: 10.1145/2016604.2016637 (cit. on p. 24).

[2] *AMD Xenos (graphics chip)*. Visited on 2013-04-19. URL: http://en.wikipedia.org/wiki/Xenos_(graphics_chip) (cit. on p. 3).

[3] Y. Amit and D. Geman. “Shape quantization and recognition with randomized trees”. In: *Neural computation* 9.7 (1997), pp. 1545–1588. DOI: 10.1162/neco.1997.9.7.1545 (cit. on p. 5).

[4] J. Bergstra, R. Bardenet, Y. Bengio, B. Kégl, et al. “Algorithms for hyper-parameter optimization”. In: *Proceedings of the Conference on Neural Information Processing Systems (NIPS)*. 2011. URL: https://github.com/jaberg/hyperopt (cit. on pp. 60, 61, 75–77).

[5] L. Breiman. “Random forests”. In: *Machine learning* 45.1 (2001), pp. 5–32. DOI: 10.1023/A:1010933404324 (cit. on pp. 5, 7, 9).

[6] Camille Couprie, Clément Farabet, Laurent Najman, and Yann LeCun. “Indoor Semantic Segmentation using depth information”. In: *The Computing Resource Repository (CoRR) abs/1301.3572* (2013) (cit. on pp. 75, 77).

[7] A. Criminisi. “Decision forests: A unified framework for classification, regression, density estimation, manifold learning and semi-supervised learning”. In: *Foundations and Trends in Computer Graphics and Vision* 7.2-3 (2011), pp. 81–227. DOI: 10.1561/0600000035 (cit. on pp. 7, 12, 13).

[8] Franklin C. Crow. “Summed-area tables for texture mapping”. In: *SIGGRAPH Comput. Graph.* 18.3 (Jan. 1984), pp. 207–212. ISSN: 0097-8930. DOI: 10.1145/964965.808600 (cit. on p. 16).

[9] *CUDA C Programming Guide*. Visited on 2013-04-22. URL: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (cit. on pp. 25, 26, 33, 37).

[10] *CUDA Profiler User’s Guide*. Visited on 2013-05-07. URL: http://docs.nvidia.com/cuda/profiler-users-guide/index.html (cit. on pp. 24, 41).

[11] Ulrich Drepper. “What every programmer should know about memory”. In: *Red Hat, Inc* (2007) (cit. on pp. 22, 30).

[12] M. Everingham, L. Van Gool, C.K.I. Williams, J. Winn, and A. Zisserman. “The pascal visual object classes (voc) challenge”. In: *International journal of computer vision* 88.2 (2010), pp. 303–338. DOI: 10.1007/s11263-009-0275-4 (cit. on p. 17).

[13] Nadeem Firasta, Mark Buxton, Paula Jinbo, Kaveh Nasri, and Shihjong Kuo. “Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency”. In: *Intel white paper* (2008) (cit. on p. 32).

[14] Michael J Flynn. “Some computer organizations and their effectiveness”. In: *IEEE Transactions on Computers* 100.9 (1972), pp. 948–960. DOI: http://dx.doi.org/10.1109/TC.1972.5009071 (cit. on p. 20).

[15] J. Friedman, T. Hastie, and R. Tibshirani. *The elements of statistical learning*. Vol. 1. Springer Series in Statistics, 2001. DOI: 10.1007/978-0-387-84858-7 (cit. on pp. 7, 48).

[16] M.A. Geary, H.J. Lee, D. Signorelli, and J. Vaughan. “Implementing Extremely Randomized Trees in CUDA”. In: (2009) (cit. on p. 3).

[17] P. Geurts, D. Ernst, and L. Wehenkel. “Extremely randomized trees”. In: *Machine learning* 63.1 (2006), pp. 3–42. DOI: 10.1007/s10994-006-6226-1 (cit. on pp. 10, 12).

[18] Matt Godbolt. *GCC Explorer*. Visited on 2013-07-15. URL: http://gcc.godbolt.org/ (cit. on p. 31).

[19] *Google Performance Tools*. Visited on 2013-05-27. URL: http://goog-perftools.sourceforge.net/ (cit. on p. 29).

[20] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. “The WEKA data mining software: an update”. In: *SIGKDD Explor. Newsl.* 11.1 (Nov. 2009), pp. 10–18. ISSN: 1931-0145. DOI: 10.1145/1656274.1656278. URL: http://doi.acm.org/10.1145/1656274.1656278 (cit. on p. 80).

[21] Zellig S Harris. “Distributional structure.” In: *Word* (1954). DOI: http://dx.doi.org/10.1007/978-94-009-8467-7_1 (cit. on p. 9).

[22] Tin Kam Ho. “Random decision forests”. In: *Proceedings of the Third International Conference on Document Analysis and Recognition (ICDAR)*. IEEE. 1995, pp. 278–. ISBN: 0-8186-7128-9. URL: http://dl.acm.org/citation.cfm?id=844379.844681 (cit. on p. 5).

[23] Jared Hoberock and Nathan Bell. *Thrust – Parallel Template Library*. Visited on 2013-04-29. 2010–. URL: http://thrust.github.io/ (cit. on pp. 38, 39, 43).

[24] Intel. *A Guide to Vectorization with Intel C++ Compilers*. Visited on 2013-05-26. 2012. URL: http://software.intel.com/en-us/articles/a-guide-to-auto-vectorization-with-intel-c-compilers (cit. on p. 27).

[25] William Kahan. “Pracniques: further remarks on reducing truncation errors”. In: *Communications of the ACM* 8.1 (1965), p. 40. DOI: 10.1145/363707.363723 (cit. on p. 16).

[26] Julián Lamas-Rodríguez. *Why aren’t there bank conflicts in global memory for Cuda/OpenCL?* Visited on 2013-07-12. URL: http://stackoverflow.com/questions/3843032/why-arent-there-bank-conflicts-in-global-memory-for-cuda-opencl#comment15256084_3847660 (cit. on p. 25).

[27] V. Lepetit and P. Fua. “Keypoint recognition using randomized trees”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 28.9 (2006), pp. 1465–1479. DOI: 10.1109/TPAMI.2006.188 (cit. on p. 1).

[28] V. Lepetit, P. Lagger, and P. Fua. “Randomized trees for real-time keypoint recognition”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Vol. 2. IEEE. 2005, pp. 775–781. DOI: 10.1109/CVPR.2005.288 (cit. on pp. 1, 3, 13, 49).

[29] Anat Levin, Dani Lischinski, and Yair Weiss. “Colorization using optimization”. In: *ACM Transactions on Graphics (TOG)*. Vol. 23. 3. ACM. 2004, pp. 689–694. DOI: 10.1145/1015706.1015780. URL: http://www.cs.huji.ac.il/~yweiss/Colorization/ (cit. on pp. 54, 55, 58).

[30] Paulius Micikevicius. *Local Memory and Register Spilling*. Visited on 2013-06-17. 2011. URL: http://on-demand.gputechconf.com/gtc-express/2011/presentations/register_spilling.pdf (cit. on p. 21).

[31] Sebastian Nowozin. Personal communication. Apr. 12, 2013 (cit. on p. 3).

[32] Sebastian Nowozin. *Tuwo Computer Vision Library*. Visited on 2012-12-10. URL: http://www.nowozin.net/sebastian/tuwo/ (cit. on pp. 28, 29, 33–35).

[33] Travis Oliphant et al. *Scientific Computing Tools For Python – NumPy*. Visited on 2013-05-24. 1995–. URL: http://www.numpy.org (cit. on p. 27).

[34] *OpenMP*. Visited on 2012-12-10. URL: http://openmp.org/ (cit. on p. 29).

[35] M. Ozuysal, M. Calonder, V. Lepetit, and P. Fua. “Fast keypoint recognition using random ferns”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 32.3 (2010), pp. 448–461. DOI: 10.1109/TPAMI.2009.23 (cit. on p. 12).

[36] M. Ozuysal, P. Fua, and V. Lepetit. “Fast keypoint recognition in ten lines of code”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE. 2007, pp. 1–8. DOI: 10.1109/CVPR.2007.383123 (cit. on pp. 12, 52).

[37] *Parallel Programming and Computing Platform | CUDA | NVIDIA*. Visited on 2012-12-10. URL: http://www.nvidia.com/object/cuda_home_new.html (cit. on pp. 19–22, 34, 38).

[38] J.J. Rodrigues, J.S. Kim, M. Furukawa, J. Xavier, P. Aguiar, and T. Kanade. “6D pose estimation of textureless shiny objects using random ferns for bin-picking”. In: (2012) (cit. on pp. 1, 52).

[39] Greg Ruetsch and Paulius Micikevicius. “Optimizing Matrix Transpose in CUDA”. In: *Nvidia CUDA SDK Application Note* (2009) (cit. on pp. 21, 23–25, 36).

[40] Thorsten Scheuermann and Justin Hensley. “Efficient Histogram Generation Using Scattering on GPUs”. In: *Proceedings of the symposium on Interactive 3D graphics and games*. ACM. 2007, pp. 33–37. DOI: 10.1145/1230100.1230105 (cit. on p. 42).

[41] F. Schroff, A. Criminisi, and A. Zisserman. “Object Class Segmentation using Random Forests”. In: (2008) (cit. on pp. 1, 3, 13).

[42] Hannes Schulz and Andreas Müller. *Matrix library for CUDA in C++ and Python*. Visited on 2013-05-24. 2011–. URL: https://github.com/deeplearningais/CUV (cit. on p. 27).

[43] J. Shotton, A. Fitzgibbon, M. Cook, T. Sharp, M. Finocchio, R. Moore, A. Kipman, and A. Blake. "Real-time human pose recognition in parts from single depth images". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE. 2011, pp. 1297–1304. DOI: 10.1109/CVPR.2011.5995316 (cit. on pp. 1–3, 11, 13, 29, 49, 50, 63, 77).

[44] J. Shotton, M. Johnson, and R. Cipolla. "Semantic texton forests for image categorization and segmentation". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE. 2008, pp. 1–8. DOI: 10.1109/CVPR.2008.4587503 (cit. on p. 1).

[45] N. Silberman, D. Hoiem, P. Kohli, and R. Fergus. "Indoor segmentation and support inference from RGBD images". In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2012. DOI: 10.1007/978-3-642-33715-4_54 (cit. on pp. 54, 57, 58, 75, 77).

[46] N. Silberman, D. Hoiem, P. Kohli, and R. Fergus. *NYU Depth v2 Train Test Split*. Visited on 2013-07-17. URL: http://horatio.cs.nyu.edu/mit/silberman/indoor_seg_sup/splits.mat (cit. on p. 58).

[47] Mark Silberstein, Assaf Schuster, Dan Geiger, Anjul Patney, and John D Owens. "Efficient Computation of Sum-products on GPUs Through Software-Managed Cache". In: *Proceedings of the International Conference on Supercomputing (ICS)*. ACM. 2008, pp. 309–318. ISBN: 978-1-6055-8158-3. DOI: http://dx.doi.org/10.1145/1375527.1375572 (cit. on p. 21).

[48] D. Slat and M.H. Lapajne. "Random Forests for CUDA GPUs". In: (2010) (cit. on p. 3).

[49] Pete Stevenson. *mem-latency – tool to measure memory latency*. Visited on 2013-07-12. URL: http://code.google.com/p/mem-latency/ (cit. on p. 22).

[50] J. Stückler, N. Biresev, and S. Behnke. "Semantic Mapping Using Object-Class Segmentation of RGB-D Images". In: *Proceedings of the IEEE/RSJ International Conference on Robots and Systems (IROS)*. 2012 (cit. on pp. 1–3, 10, 11, 13, 15, 18, 29, 34, 42, 48, 54, 55, 60, 61, 63, 77).

[51] *Thread Building Blocks*. Visited on 2012-12-10. URL: http://threadingbuildingblocks.org/ (cit. on pp. 29, 31).

[52] Toby Sharp. "Implementing decision trees and forests on a GPU". In: *Proceedings of the European Conference on Computer Vision (ECCV)*. Springer, 2008, pp. 595–608. DOI: 10.1007/978-3-540-88693-8_44 (cit. on pp. 1–3, 10, 13, 15, 18, 29, 42, 49, 50, 63, 69, 70, 77).

[53] B. Van Essen, C. Macaraeg, M. Gokhale, and R. Prenger. "Accelerating a Random Forest Classifier: Multi-Core, GP-GPU, or FPGA?" In: *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2012, pp. 232–239. DOI: 10.1109/FCCM.2012.47 (cit. on pp. 3, 4, 18, 64).

[54] P. Viola and M. Jones. "Rapid object detection using a boosted cascade of simple features". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. Vol. 1. IEEE. 2001, pp. I–511. DOI: 10.1109/CVPR.2001.990517 (cit. on pp. 3, 13, 16).

[55] Vasily Volkov. "Better Performance at Lower Occupancy". In: *Proceedings of the GPU Technology Conference (GTC)*. Vol. 10. 2010 (cit. on p. 26).

[56] Benedikt Waldvogel. *CURFIL – CUDA Random Forest for Image Labeling tasks*. 2013. URL: https://github.com/deeplearningais/curfil (cit. on p. 78).

[57] James Wang. *Whitepaper – NVIDIA's Next Generation CUDA™ Compute Architecture: Fermi™*. Visited on 2013-04-19. URL: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf (cit. on pp. 19, 20).

[58] Louis Wehenkel and Mania Pavella. "Decision trees and transient stability of electric power systems". In: *Automatica* 27.1 (1991), pp. 115–134 (cit. on p. 10).

[59] John Winn, Antonio Criminisi, and Thomas Minka. "Object categorization by learned universal visual dictionary". In: *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. Vol. 2. IEEE. 2005, pp. 1800–1807 (cit. on p. 63).

[60] Mo Xu, Ningyi Xu, Chunshui Zhao, and Feng-Hsiung Hsu. "Efficient Weighted Histogramming on GPUs with CUDA". In: (2012) (cit. on p. 43).

A. Appendix

The following plots and tables serve as supplementary results that have been mentioned in this master's thesis but were extracted to avoid distraction.

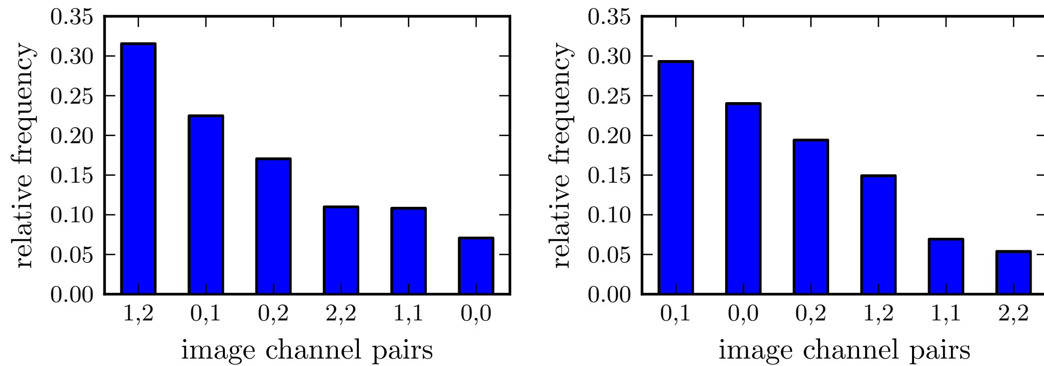

Figure A.1.: Relative feature frequency per (channel 1, channel 2) pair. Comparisons on the same image channels are less frequently selected in training than comparisons on two different image channels. Left: Random forest trained on AIS Bonn Large-Objects dataset with raw depth. Right: Random forest trained on NYU Depth v2 dataset. See Section 3.2.2.

Samples per thread block	Features per thread block	Training time [ms]	Texture cache Hit rate [%]	L2 cache Hit rate [%]
1	128	6.6	19.4	76.9
4	32	**6.2**	12.6	78.5
8	16	7.2	12.6	76.3
16	8	10.2	8.2	62.7
16	16	10.5	11.6	62.4
32	8	13.8	7.3	44.1
32	4	14.8	5.3	44.9

Table A.1.: Training time and texture cache hit rate of feature response calculation for **20 samples per image**, **100 images**, 2000 features, maximum offset of 120 px m and maximum region extent of 20 px m $\times$ 20 px m on a GeForce GTX 480. Feature response values are stored to a $D \times F$ matrix for D samples and F features in **row-major** order. Training time increases with more samples per thread block, because texture cache rate decreases. Performance peaks at 4 samples and 32 features per thread See Section 4.2.4.

Samples per block	Features per block	Training time [ms]	Texture cache hit rate [%]	L2 cache hit rate [%]	Global memory store efficiency [%]
1	128	61.9	21.8	97.3	25
4	32	51.7	27.0	96.6	100
8	16	49.2	30.3	97.1	100
16	8	47.8	31.2	97.7	100
16	16	**46.8**	37.1	97.7	100
32	8	48.8	34.1	97.6	100
32	4	50.6	28.0	97.7	100

Table A.2.: Training time, cache hit rate and global memory store efficiency of feature response calculation **2000 samples per image**, **10 images**, 2000 samples, maximum offset of 120 px m and maximum region extent of 20 px m $\times$ 20 px m on a GeForce GTX 480. Feature response values are stored to a $D \times F$ matrix for D samples and F features in **column-major** order. Training time decreases when texture cache hit rate increases. Performance peaks at 16 samples and 16 features per thread block. Using 128 features and only one sample per thread block is penalized due to inefficient global memory writes. See Section 4.2.4.

Samples per block	Features per block	Training time [ms]	Texture cache hit rate [%]	L2 cache hit rate [%]	Global memory store efficiency [%]
1	128	70.0	22.2	91.0	25
4	32	**66.7**	19.6	92.0	100
8	16	67.8	16.0	93.0	100
16	8	71.9	12.1	92.6	100
16	16	67.3	14.4	92.4	100
32	8	76.1	9.0	89.4	100
32	4	78.6	7.8	89.6	100

Table A.3.: Training time, cache hit rate and global memory store efficiency of feature response calculation **200 samples per image**, **100 images**, 2000 samples, maximum offset of 120 px m and maximum region extent of 20 px m × 20 px m on a GeForce GTX 480. Feature response values are stored to a $D \times F$ matrix for D samples and F features in **column-major** order. Training time decreases when texture cache hit rate increases. Performance peaks at 4 samples and 32 features per thread block. Using 128 features and only one sample per thread block is penalized due to inefficient global memory writes. See Section 4.2.4.

Samples per block	Features per block	Training time [ms]	Texture cache hit rate [%]	L2 cache hit rate [%]	Global memory store efficiency [%]
1	128	**7.8**	22.8	75.3	25
4	32	9.1	13.0	77.2	100
8	16	11.8	8.8	76.8	100
16	8	14.1	6.8	64.6	100
16	16	10.5	6.4	65.7	100
32	8	16.1	4.0	47.0	100
32	4	15.2	4.5	44.9	100

Table A.4.: Training time, cache hit rate and global memory store efficiency of feature response calculation **20 samples per image**, **100 images**, 2000 samples, maximum offset of 120 px m and maximum region extent of 20 px m × 20 px m on a GeForce GTX 480. Feature response values are stored to a $D \times F$ matrix for D samples and F features in **column-major** order. Training time decreases with increasing number of samples per thread block. The reason is a decreasing texture cache hit rate, since sampling is dense and two samples are not likely to lie close to each other. Performance peaks at one sample and 128 features per thread block, even though it is penalized due to inefficient global memory writes. See Section 4.2.4.