Westfälische Wilhelms-Universität
Münster

Bachelorthesis

Evaluation of the md_hom Pattern Using the Example “Ensemble of Classifier Chains“

Author:
Luis Wetzel
l_wetz03@uni-muenster.de
Matr. Nr. 405159

Betreuer:
Ari Rasch
a.rasch@uni-muenster.de

handed in on 2. Mai 2018

Fachbereich 10
Mathematik und
Informatik

Table of Contents

1. Introduction

Modern computer processors support parallel processing which can drastically increase program performance compared to traditional purely sequential computing. However effectively utilizing parallel architectures creates multiple challenges. md_hom is a pattern which provides a solution for certain recurring parallel programing problems. This thesis explores how md_hom can be used to create a competitive implementation of Ensemble of Classifier Chains. Ensemble of Classifier Chains is a machine learning algorithm which has been applied to a variety of domains [1]. This thesis will present a new implementation for Ensemble of Classifier Chains using the md_hom pattern, present how to further optimize the implementation using auto-tuning and compare the performance of the new implementation to the existing parallel implementation for Ensemble of Classifier Chains *eccCL* [2]. First Ensembles of Classifier Chains and other basic concepts will be introduced, afterwards the new implementation will be presented and the algorithm is evaluated and compared to *eccCL*.

2. Basic Concepts

2.1. Ensemble of Classifier Chains

Ensemble of Classifier Chains (ECC) is a family of machine-learning techniques used for multi-label classification. This section introduces the basic concepts underlying ECC following the original paper [3].

2.1.1. Supervised Classification

ECC is a supervised classification algorithm. In supervised classification input data is classified using a two step process. First a dataset with already known classification, the training set, is used with a training algorithm to create a model (or classifier). Then the second step uses another dataset for which the classification is not yet known. A classification algorithm uses the trained model to predict the classification of the data in the set. The quality of the classifier is usually evaluated by using a dataset with known classification, which does not intersect with the training set, as test set. The resulting predictions can be compared to the true classification (true set) using a variety of ways to produce a measure for prediction performance.

2.1.2. Multi-label Classification

The goal of multi-label classification is to associate data with a set of labels. The classification algorithm takes a set of instances D, the dataset, as input. Each instace $x_i \in D$ is a n-dimensional vector $(x_{i,1}, x_{i,2}, \ldots, x_{i,n})$ where each entry $x_{i,j}$ is called an attribute. In single-label classification every instance x_i would be associated with a single label from a finite set of labels L. In multi-label classification the algorithm outputs a set of labels $S_i \subseteq L$ for every instance x_i in D.
One common approach to multi-label classification is through algorithm adaption where an existing single-label classification method is adapted to work with mutli-label data. Another common approach is problem transformation, where the multi-label problem is transformed into multiple single-label problems which can then be solved using existing single-label classification methods. A problem transformation method related to ECC is known as binary relevance method. In binary relevance the problem of finding a set of labels $S_i \subseteq L$ for each instance x_i is split up into one binary decision for every label $l \in L$. For each label $l \in L$ a binary classifier decides wether or not l is in S_i. Since every label is treated fully independently of the others binary relevance cannot model correlations between labels, which is a common criticism of the method.

2.1.3. Classifier Chains

Classifier Chains (CC) is a variation of the binary relevance method which uses one binary classifier B_k for each label $l_k \in L$. Each classifier B_k predicts wether or not the associated label is included in the predicted classification of an instance x_i. The prediction is represented as either a 1 if the predicted label is part of the classification or a 0 if it isn't, therefore the full classification is represented as a vector $(l_1, l_2, \dots, l_{|L|}) \in \{0, 1\}^{|L|}$. The key difference to binary relevance is that the predictions for the individual labels are performed sequentially, so that each classifier B_k can predict the corresponding label l_k not only based on an instance x_i but also on the predictions for all previous labels $l_{k'}$ where $k' < k$. This allows CC to involve label correlations in the prediction with the downside that the inherently sequential process cannot be parallelized.
The training is done using a training set D where each element $x_i \in D$ is a vector $(x_{i,1}, \dots, x_{i,n}, l_{i,1}, \dots, l_{i,|L|})$ where $x_{i,j}$ are the instance attributes and $l_{i,k}$ the known classification of the instance represented as a 0 or 1 for each label. To train the individual classifiers B_k each training instance $x_i \in D$ is first transformed into a vector $(x_{i,1}, \dots, x_{i,n}, l_{i,1}, \dots, l_{i,k-1})$ which only contains the classification of the labels prior in the sequence. The transformed instances are then used to train the classifier. The classification is done by taking an instance x_i and predicting the first label l_1 using B_1. Then the label is appended to x_i to create a vector $(x_{i,1}, \dots, x_{i,n}, l_{i,1})$, which is then used with B_2 to predict l_2, which is then again appended to the vector and passed on to B_3. This is done sequentially for every classifier until all labels are predicted, the process is visualized in figure 2.1.

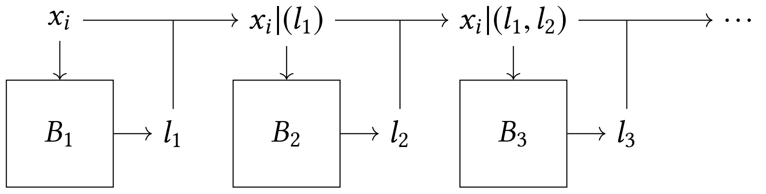


Figure 2.1.: Classification of an instance x_i using a CC classifier

2.1.4. Ensemble of Classifier Chains

Ensemble of Classifier Chains (ECC) is a multi-label classifier utilizing m CC classifiers $C_1, C_2, \dots, C_m$. ECC is trained using a training set D by training each C_k with:

1. a random subset of D
2. a random order in which the labels in L are predicted by the binary classifiers of C_k

A given instance x_i is classified using ECC by classifying the instance with every classifier C_k, each producing a prediction vector $(l_{k,1}, l_{k,2}, \dots, l_{k,|L|}) \in \{0, 1\}^{|L|}$. These predictions are then summed up to produce a vector

$$y = (y_1, y_2, \ldots, y_{|L|}) \in \mathbb{N}^{|L|} \text{ where } y_j = \sum_{k=1}^{m} l_{k,j}$$

This represents a simple voting scheme, where each classifier votes wether or not the label should be included. y represents the total number of votes for each label. The last step is to transform y into the final prediction $S_i \subseteq L$. The original paper on ECC suggests normalizing y to produce a vector $\hat{y}$ and then applying a threshold t to each element so that $S_i = \{l_j \mid l_j \in L \land \hat{y}_j > t\}$.
The advantage of ECC over CC is that the randomization involved in creating each classifier produces a different model which leads to more stable predictions, especially since a single CC classifier can only involve the prediction of the labels earlier in the sequence when predicting a label. By randomizing the label order there is a chance that the final prediction for a given label depends on predictions for every other label and can therefore take label correlations into account.

2.1.5. Random Forests

The individual CC classifiers in the ECC classifier are themselves chained up binary classifiers, any binary classifier can be used. This thesis will only consider Random Forest (RF) as base classifier. RF is an ensemble of t decision trees. A decision tree is a binary classifier. The training process of a RF [4] classifier consists of creating random subsets of the training set D for every decision tree T and training T with the subset. RF predicts a label for a given instance x_i by first making a prediction using every classifier T and then using a voting scheme, where the most common prediction is chosen.

2.1.6. Classifier Tree

The decision trees are a variation of the *Iterative Dichotomiser 3* (ID3) algorithm presented by Ross Quinlan [5]. The tree algorithm is taken from the *eccCL* implementation of ECC, which forces the trees to be complete binary trees of fixed height [**eccCl**], the original ID3 does not have this requirement.
The algorithm differentiates between internal nodes and leaf nodes. The internal nodes are a tuple $n = (n_a, n_v)$ with $n \in \{0, \ldots, A\} \times \mathbb{R}$ where A is the number of attributes in the training set. n_a is the index of an attribute of the instances, n_v is a threshold value for that attribute. The leaf nodes contain predictions $p \in \{-1, 0, 1\}$, where

1 represents that tree predicts the label to be set

-1 represents that tree predicts the label to not be set

0 a special case that is introduced because of the complete binary tree requirement

The training of the internal nodes is shown in Algorithm 1. The nodes are trained using a set $D' \subseteq D$,where D is the training set for the tree, which contains the attributes and the known label value of the instances. The algorithm chooses k_{max} candidate nodes by selecting a random attribute index a and and a random instance x and then selecting the best candidate (Line 4-12). The candidate nodes are evaluated using the information gain function G. G is a measure for how much better the data in D' would ordered when split into two subsets

$$D_{\leq} = \{x' \in D' | x'_a \leq x_a\}$$

and

$$D_{>} = \{x' \in D' | x'_a > x_a\}$$

x_a is the a-th attributs in x. The measure for order is the entropy E where the entropy of a training set with respect to the label is defined as:

$$E(S) = -\Big(P(l = 1) \cdot log_2(P(l = 1)) + P(l \neq 1) \cdot log_2(P(l \neq 1))\Big)$$

where $P(l = 1)$ is the empirical propability that the label is set on an instance $x \in S$, $P(l \neq 1)$ the opposite. A high entropy means low order. The information gain G is therefore defined as:

$$G(D, x, a) = E(D) - \left(\frac{|D_{>}|}{|D|}E(D_{>}) + \frac{|D_{\leq}|}{|D|}E(D_{\leq})\right)$$

The candidate node with the highest information gain is chosen as the final node. The input data set D' is then split into the two subsets $D_{\leq}$ and $D_{>}$ (Lines 15-21).

Algorithm 1 Building Nodes

```
 1: function BUILDNODE($D'$)
 2:     $g_{max} \leftarrow 0$
 3:     $n \leftarrow (0, 0)$                                    ▷ $n = (n_a, n_v)$
 4:     for $k \leftarrow 1, k_{max}$ do          ▷ Find node candidate with high information gain
 5:         $x \leftarrow$ Random instance from $D'$
 6:         $a \leftarrow$ Random attribute index
 7:         $g = G(D', x, a)$
 8:         if $g > g_{max}$ then
 9:             $g_{max} \leftarrow g$
10:             $n \leftarrow (a, x_a)$
11:         end if
12:     end for
13:     $D_{<} \leftarrow \emptyset$
14:     $D_{>} \leftarrow \emptyset$
15:     for $x \in D'$ do                                       ▷ Split training set
16:         if $x_{n_a} > n_v$ then
17:             $D_{>} \leftarrow D_{>} \cup \{x\}$
18:         else
19:             $D_{\leq} \leftarrow D_{\leq} \cup \{x\}$
20:         end if
21:     end for
22:     return $(n, D_{\leq}, D_{>})$   ▷ Return node and subsets, subsets used to build child nodes
23: end function
```

The algorithm first trains the root node using a subset of the training data of the forest. Then the left child of a trained node is trained with buildnode($D_{\leq}$) and the right with buildnode($D_{>}$), nodes are trained until a fixed depth is reached. If one subset is empty it is not possible to train the corresponding child node and it is set to a default value $n \leftarrow (0, 0)$, to satisfy the complete binary tree condition. When all internal nodes are built the leaf nodes are appended and initialized with 0. The training of the leaf nodes is shown in Algorithm 2. The leaves are trained by traversing the tree with every instance (Lines 1-16). The traversal of an instance x starts at the root node, then at every node $n = (n_a, n_v)$ the traversal continues with the left child if $x_{n_a} \leq n_v$ and with the right child otherwise. When a leaf node is reached the leaf node's value is incremented if the label that the tree should predict is set on x, the node value is decremented if it isn‘t set (Lines 11-15). Finally, after all traversals are done, the leaf nodes are set to 1 if it's value is positive, to -1 if it is negative and is otherwise left at 0 (Lines 17-23).

Algorithm 2 Training Leafs

```
for $x \in D$ do
    $n \leftarrow$ Root node
    while $n$ not leaf do
        if $x_{n_a} > n_v$ then
            $n \leftarrow$ right child of $n$
        else
            $n \leftarrow$ left child of $n$
        end if
    end while
        ▷ $n$ is now leaf node, count how many instances with and without the label set reach this node
    if $x_l = 1$ then
        $n \leftarrow n + 1$
    else
        $n \leftarrow n - 1$
    end if
end for
for Leaf node $n$ do
    if $n > 0$ then
        $n \leftarrow 1$
    else if $n < 0$ then
        $n \leftarrow -1$
    end if
end for
```

The classification of an instance works similarly to Algorithm 2, but when a leaf node is reached the leaf node's value is returned as the tree's prediction.

2.2. Parallelism and Auto-Tuning

2.2.1. Parallelism

Modern processors have the ability to execute multiple calculations in parallel. Effective utilisation of parallelism can significantly reduce the runtime of programs, but it also creates new challenges for programmers. It is often not obvious how to ensure correctness and use the hardware most effectively. Processors are often categorized into four groups [6]:

1. Single instruction, single data (SISD): supports no parallelism
2. Single instruction, multiple data (SIMD): can run the same programmwith different data in parallel

3. Multiple instruction, single data (MISD): can run multiple programs in parallel all operating on the same data

4. Multiple instruction, multiple data (MIMD): can run multiple programs in parallel each operating on different data

The parallelization techniques used in this thesis assume a SIMD architecture. SIMD processors are able to run the same program multiple times in parallel and let each instance of that program use different data.
A common example for a problem that can effectively utilize a SIMD architecture is vector addition. Two vectors $a, b \in \mathbb{R}^N$ are added to calculate $c \in \mathbb{R}^N$ where $c = a + b$. Algorithm 3 shows a sequential program for this calculation.

Algorithm 3 Sequential vector addition

```
1: for $i \leftarrow 1, N$ do
2:     $c_i \leftarrow a_i + b_i$
3: end for
```

Since every element of c can be calculated independently of the other the program can be transformed into a program that is executed N times in parallel as shown in Algorithm 4

Algorithm 4 Parallel vector addition

```
1: $i \leftarrow I$          ▷ I between 1 and N, set differently for each instance of the program
2: $c_i \leftarrow a_i + b_i$
```

2.2.2. Auto-Tuning

The program shown in Algorithm 4 has to be executed N times, once for every element in c. On real-world processors this might not be the solution with optimal performance. Various factors like memory access patterns, caching strategies, number of available compute units and others might make more sequential calculation with less parallelism a more efficient in practice. The complexity of modern hardware makes it very difficult to know how much parallelism is ideal on any given platform. One way to overcome this problem is by using *auto-tuning*. In auto-tuning a program has to be programmed in a way that the degree of parallelism depends on a, often compile-time, parameter. This process is called parameterization. Algorithm 5 shows a parameterized version of the vector addition program.

Algorithm 5 Parameterized Vector addition

```
1: $p \leftarrow I$        ▷ I between 1 and $P$, set differently for each instance of the program
2: for $j \leftarrow 1, \frac{N}{P}$ do
3:     $i \leftarrow (p-1) * \frac{N}{P} + j$
4:     $c_i \leftarrow a_i + b_i$
5: end for
```

The degree of parallelization depends on the parameter P , where P has to be a divisor of N. The algorithm becomes fully sequential, like Algorithm 3, for $P = 1$ and fully parallel like Algorithm 4 for $P = N$. In general a program with a vector of parameters $p \in P$ where P is the parameter space can be used to create a measurement function $t : P \longrightarrow \mathbb{R}$, where t returns a runtime measurement for the program when run with a set of paramters, or configuration. The goal is to find the configuration that minimizes t or at least makes t sufficiently small. Auto-tuning software is used to search the parameter space for good configurations, this can be done using a variety of search techniques.

2.3. md_hom Skeleton

Different programs often have similarities in their structure. It is therefore useful to find an abstract generalization of this structure in order to study the properties of these programs. These generalizations may also be used to find parallelization templates (skeletons), that can be used to parallelize any program that can be expressed using the generalization. This thesis explores the md_hom skeleton introduced by Ari Rasch and Sergei Gorlatch [7].

2.3.1. Multi-Dimensional Arrays

The md_hom skeleton is based on the notion of multi-dimensional arrays (MDA) and concatenation operations on MDAs. An MDA of type T, where T is a set of values, is a function

$$a\colon \prod_{i=1}^{d} \{1, \ldots, N_i\} \to T \ ,$$

where d is the number of dimensions and N_i the size in dimension i. The original paper suggests defining a C++ style notation for MDAs:

- $T[N_1] \ldots [N_d]$ as the set of all MDAs of type T with d dimensions and $(N_1, \ldots, N_d)$ as sizes

- given an MDA $a \in T[N_1] \ldots [N_d]$, we define $a[i_1]...[i_d] \in T$ as the element of a accessed throught the indices $i_1, \ldots, i_n$

MDA concatenation $+\!\!+_k$ is a binary function of two MDAs with same number of dimensions d and same sizes in every dimension except the k-th:

$$+\!\!+_k \colon T[N_1]\dots[\,P\,]\dots[N_d] \times T[N_1]\dots[\,Q\,]\dots[N_d] \to T[N_1]\dots[P+Q]\dots[N_d]$$

where

$$(a +\!\!+_k b)[i_1]\dots[i_d] = \begin{cases} a[i_1]\dots[\quad i_k \quad]\dots[i_d] & i_k \leq P \\ b[i_1]\dots[i_k - P]\dots[i_d] & \text{otherwise} \end{cases}$$

2.3.2. Multi-Dimensional Homomorphism

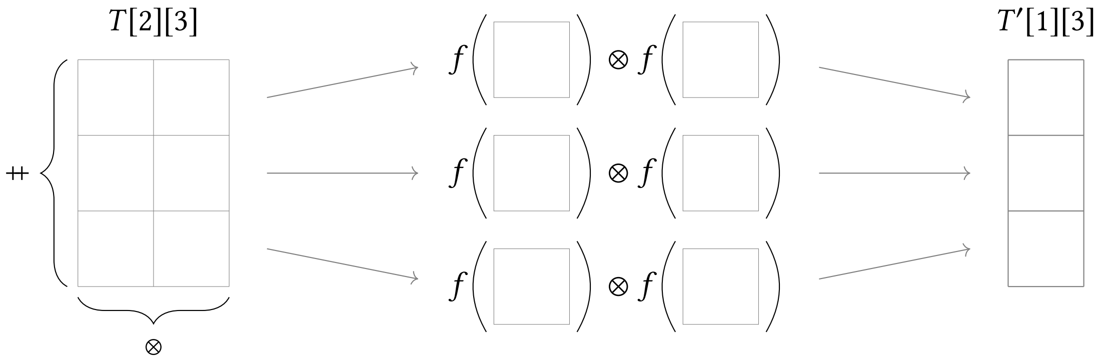
A function $h \colon T[N_1]\dots[N_d] \to T'$ on a d-dimensional MDA is called a multi-dimensional homomorphism (MDH) iff there exists an operator $\otimes_k \colon T' \times T' \to T'$ for every dimension $k \in \{1,\dots,d\}$ such that h is homomorphism for $\otimes_k$ and $+\!\!+_k$ in that dimension:

$$h(a +\!\!+_k b) = h(a) \otimes_k h(b)$$

This property, called homomorphic property, allows MDHs to be very flexible in the way they are evaluated as the input can be split in several dimensions in multiple ways at the same time down to every individual element of the input MDA.

2.3.3. md_hom Skeleton

The md_hom skeleton is a generic way to express MDHs and provides a program template that can be used to implement the MDH by utilising parallelism on SIMD architectures. Given a function $f \colon T \to T'$, called the scalar function, and d binary operators $\otimes_k \colon T' \times T' \to T'$ then *md_hom*$(f, (\otimes_1,\dots,\otimes_d))$ constructs an MDH $h \colon T[N_1]\dots[N_d] \to T'$ which recursively uses the homomorphic property until the input MDA is split into parts of size 1 in all dimensions. Evaluating a single part a is then defined as $h(a) = f(a[1]\dots[1])$. The individual parts can be calculated in parallel or sequentially and the results combined using the operators to get the final results. If the operators $\otimes_k$ do not satisfy the homomorphic property, *md_hom* is undefined. *md_hom* allows to express MDHs by just describing the operators and a function from T to T' without having to take MDAs into account directly. This allows *md_hom* to work with all MDAs of dimension d with no extra work.



$T[2][3]$
$++$
$\otimes$
$f(\square) \otimes f(\square)$
$f(\square) \otimes f(\square)$
$f(\square) \otimes f(\square)$
$T'[1][3]$

Figure 2.2.: Example of a $md\_hom(f, (\otimes, ++))$

Figure 2.2 illustrates an example of a 2 dimensional *md_hom* using a scalar function f, which uses a combination operator $\otimes$ in one and a concatenation operator $++$ in the other dimension. The input MDA is split into the individual elements and the scalar function evaluated for each, afterwards the results are combined. The different evaluations of f can be calculated in parallel.
The original md_hom paper[7] presents an implementation for this pattern which groups evaluations of the scalar function into a sevaral parallel jobs, each job evaluates the the assigned evaluations sequentially. The results in dimensions that are not concatenations are accumulated. The different jobs are run as a parallel program. The jobs are themselves grouped and the accumulated results of each job is reduced using parallel reduction within the group. Then in a final step a second parallel program is launched that uses parallel reduction again to reduce the results of of every group of the first program into a single result in all non-concatenation dimensions. The hierarchical grouping is done to be fit closely to the OpenCL programming model (see section 3.2.1). The size of the different groups in each dimension is parameterized and can therefore be used with auto-tuning.

3. Implementing Ensemble of Classifier Chains using the md_hom skeleton

In order to implement an ECC classifier using the md_hom pattern the ECC model has to be expressed using the md_hom definitions. This chapter will first present the scalar functions and operators that can be used to express the training and classification process of the ECC classifier using md_hom on an abstract level. Afterwards an ECC implementation using the OpenCL code skeleton given in the md_hom paper is presented. Finally some auto-tuning strategies are shown.

3.1. Expressing ECC with md_hom

3.1.1. Training

The model for the ECC classifier consists of C chains with L forests each and every forest has T classifier trees. Each classifier tree can be trained independently from the others using the label that is to be predicted as well as a set of instances $D' \subseteq D$ where D is the training data for the ECC classifier. The input data for the training algortihm can be represented as an MDA $(\{0, \ldots, L\} \times (\mathbb{R}^A \times \{0,1\}^L)^{|D'|})[C \cdot L \cdot T]$. Each element in the MDA can be transformed into a decision tree $t \in TREE$, where $TREE$ is the set of all decision trees, using a function $build\_tree\colon (\{0, \ldots, L\} \times (\mathbb{R}^A \times \{0,1\}^L)^{|D'|}) \longrightarrow TREE$. The entire process can thus be represented using as an md_hom when written as $build = md\_hom(build\_tree, +\!\!+)$. $build$ creates a MDA $TREE[C \cdot L \cdot T]$ which is the model for the ECC. It is a special case of an md_hom with only concatenation operators, where no combination of outputs of the scalar functions need to be calculated.

3.1.2. Classification

A tree can create a vote $v \in VOTE = \{-1, 0, 1\} \times \{0, 1\}$ for every instance. The first element of the output is 1 if the tree predicts the label to be part of the classification and -1 if it isn‘t. The second element of the output is 1 in both cases. It is possible that the classification ends with a completely untrained node, both entries of the output are 0 in this case. To fully classify a dataset every tree in every forest in every chain has to make a prediction for every instance in the dataset, this leads to a total of $I \cdot C \cdot F \cdot T$ predictions. However the forests in a chain have to make their prediction sequentially. The parallel ECC algorithm therefore works as described in Algorithm 6.

Algorithm 6 General ECC Algorithm

```
1: for $s \leftarrow 0, L-1$ do
2:     predict every instace with every tree in $s$-th forest of every chain
3:     reduce each trees prediction to a single prediction per chain per instance
4: end for
5: reduce each chains prediction to a single prediction per label per instace
```

The base operation of the algorithm is a single tree performing a prediction on a single instance, this is a function mapping a tuple of a tree and an instance, including previous predictions in the chain, to a vote.

$$\text{traverse}_s : TREE \times (\mathbb{R}^A \times \{0,1\}^s) \to VOTE$$

A is the number of attributes and s the current step in the chain. *traverse*$_s$ has to be evaluated for every possible combination of every instance and every tree in the forests at the current step in the chains. Therefore a total of $I \cdot C \cdot T$ operations have to be performed at every loop iteration in Algorithm 6. This results in the same number of votes, the votes for every instance for every chain have to be combined to the combined votes of the forest and then the voting scheme has to transform the collected votes to a prediction $l \in \{0,1\}$.
This can be represented using the md_hom skeleton

$$step_s = md\_hom(\text{traverse}_s, (+\!\!+_I, +\!\!+_C, \otimes_{VOTE}))$$

where $\otimes_{VOTE} : VOTE \times VOTE \to VOTE$ is the operator for combining the votes $+\!\!+_I, +\!\!+_C$ are the concatenation operators for the votes of the different chains and instances. The md_hom takes a MDA $(TREE \times \mathbb{R}^{A+s})[I][C][T]$ as input and produces a MDA $VOTE[I][C]$. The input is created by pairing each tree of the current forest in every chain with all instances and then appending the labels previously predicted in the chain to the instances. A view function is used to present the data in that way:

$$VIEW_s : (TREE^{C \times L \times T} \times \mathbb{R}^{I \times A}, \{0,1\}^{I \times C \times s}) \to (TREE \times \mathbb{R}^{A+s})^{I \times C \times T}$$

. The output MDA of $step_s$ of type $VOTE[I][C]$ represents the combined votes of all forests for every instance, they are transformed to the finaly binary prediction using a function *voting_scheme*. After all predictions for the current step are calculated they get appended to a buffer storing all predictions made so far. Algorithm 7 shows the the updated step process shown in Algorithm 6 using $VIEW_s$ and $step_s$.

Algorithm 7 Step using md_hom

```
 1: $Labels \leftarrow null$
 2: $M \leftarrow$ all trees of ECC
 3: $D \leftarrow$ input dataset
 4: for $s \leftarrow 0, L-1$ do
 5:     $IT \leftarrow VIEW_s(M, D, Labels)$
 6:     $Votes \leftarrow step_s(IT)$
 7:     $Labels \mathbin{+\!\!+} voting\_scheme(Votes)$
 8: end for
 9: Reduce predictions to one per instance per label
```

The full chain iteration step can be written as $voting\_scheme \circ step_s \circ VIEW_s$. After all iterations are done the label buffer $Labels \in \{0,1\}^{I \times L \times C}$ has a prediction for every instance for every chain for every label.
The final step is to combine the votes from the different chains to one vote for every instance for every label and transforming it into a confidence value $p \in [0,1]$. The confidence value is used instead of a binary output, because techniques of evaluating prediction performance, like log loss, work with confidences. The final step can be expressed using the md_hom pattern:

$$\text{final} = md\_hom(id, (\mathbin{+\!\!+}_I, \mathbin{+\!\!+}_L, +))$$

where id is the identity function, $\mathbin{+\!\!+}_I, \mathbin{+\!\!+}_L$ are the concatenation operators for the instances and labels, $+$ is the familiar addition in $\mathbb{N}$. $final$ takes an MDA $\{0,1\}[I][L][C]$ and produces an MDA $\{0,\dots,C\}[I][L]$. In the end every element in the output has to be normalized from the range $[0,C]$ to $[0,1]$. Algorithm 8 describes the full algorithm.

Algorithm 8 Full ECC algorithm using md_hom

```
 1: $Labels \leftarrow null$
 2: $M \leftarrow$ all trees of ECC
 3: $D \leftarrow$ input dataset
 4: for $s \leftarrow 0, L-1$ do
 5:     $IT \leftarrow VIEW_s(M, D, Labels)$
 6:     $Votes \leftarrow step_s(IT)$
 7:     $Labels \mathbin{+\!\!+} voting\_scheme(Votes)$
 8: end for
 9: $Votes \leftarrow final(Labels)$
10: $Result \leftarrow normalize(Votes)$
```

3.2. Implementation

3.2.1. OpenCL

The implementation is based on OpenCL. OpenCL is both an API and a programming language that is supported by a wide variety platforms [8]. The OpenCL programming language is designed to write programs for SIMD architectures and has multiple features that reflect the architecture of modern SIMD processors. The OpenCL API can be used from traditional programming languages like Java or C++. The application which uses OpenCL API, called host, may use the API to communicate with an OpenCL device which can run programs written in the OpenCL programming language. The following lists a number OpenCL concepts key to the md_hom implementation:

- **Kernel**: An OpenCL program which can be executed by an OpenCL capable device
- **Workitem**: One instance of a Kernel. Usually many run in parallel.
- **Workgroup**: Workitems are organized in groups, the groups are called Workgroups
- **Private memory**: Memory area that every workitem has. Cannot be accessed from outside the Workitem it belongs to
- **Local memory**: Memory area that every Workgroup has. Can be accessed by every Workitem in the Workgroup
- **Global memory**: Memory area that can be accessed by every Workitem in every Workgroup
- **NDRangeKernel**: Usually many Workitems are started per Kernel. The Workitems are organized like a multi-dimensional array every Workitem is initialized with its respective index in the array. The Workgroups are chunks of the array, all Workgroups have the same size.
- **Local size**: Number of Workitems per Workgroup in every dimension
- **Global size**: Total number of Workitems in every dimension
- **Buffer**: A block of global memory that can accessed by the host and the workitems. Can, for example, be written by the host to provide data for the calculations or read back to retrieve results
- **Barrier**: Instruction in the OpenCL language that makes Workitems wait until all other Workitems in the Workgroup reach the Barrier instruction

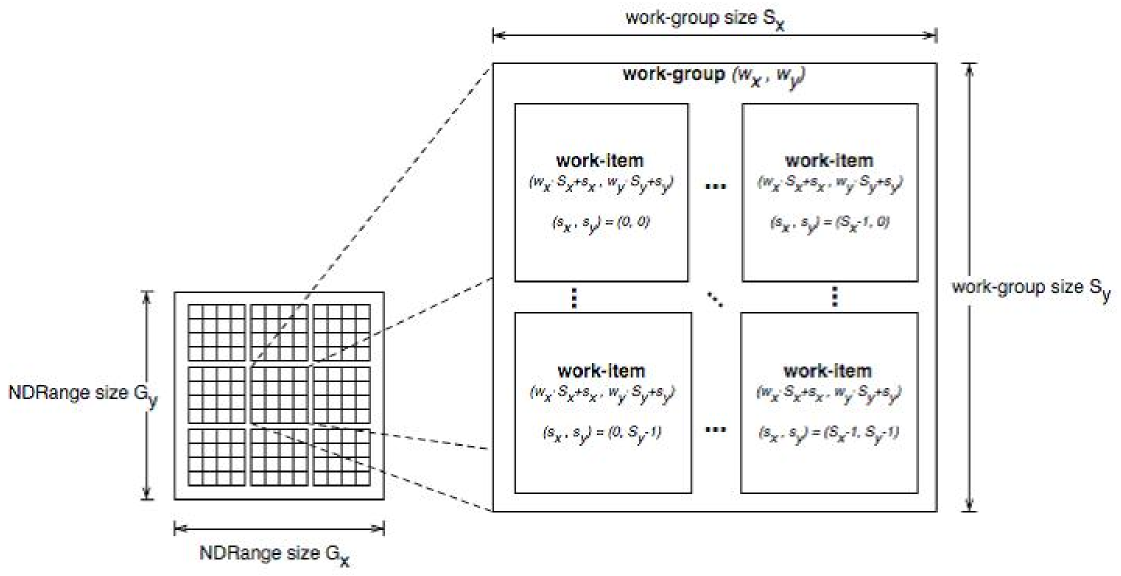

work-group size S_x
work-group (w_x, w_y)
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (0, 0)$
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (S_x-1, 0)$
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (0, S_y-1)$
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (S_x-1, S_y-1)$
work-group size S_y
NDRange size G_y
NDRange size G_x

Figure 3.1.: Workgroups and Workitems in OpenCL [8]

3.2.2. Basic Structure

The md_hom implementation described in section 2.3.3 uses two kernels, the first evaluates the scalar function grouped into workitems and worgroups, producing a result for every element in the input MDA, afterwards all non concatenation dimensions are reduced to a single result per workgroup. The second kernel reduces the per workgroup results to a single one. The $step_s$ and $final$ MDHs described in Algorithm 8 as well as the *build* MDH are implemented using the md_hom kernels. The md_hom kernels share a buffer which store per Workgroup results of the first kernel. The ECC model is stored in two buffers one for the attribute indices at every node and one for the threshold values in every node in the ECC. The basic OpenCL code for tree traversal and training is in large parts taken from *eccCL* [2]. The individual trees are flattened depth first (see figure 3.2). The random selection of indices and instances used in the tree building process is done using random seeds generated on the host which are used by a pseudo random number generator in the OpenCL code to generate pseudo random numbers.

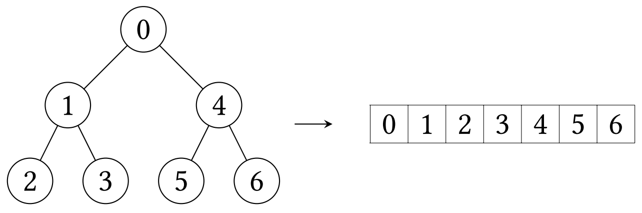


Figure 3.2.: Storing a tree in a linear buffer

3.2.3. Kernels

The OpenCL kernels follow the basic md_hom pattern. The listing 3.1 shows the OpenCL code for the prototype of the scalar function *traverse*$_s$ (Line 56) and the combine operator $\otimes_{VOTE}$ (Lines 20-36) as well as $VIEW_s$ (Lines 10-12). InputAtom (Line 50) is the tuple of a tree and an instance, which *traverse*$_s$ operates on. OutputAtom (Line 14) is the $VOTE$ structure that *traverse*$_s$ produces. The combine operator are implemented as add assign operators, because the md_hom skeleton only uses the operators that way. Since OpenCL requires to explicitly specify the memory type of each function parameter the operator has to be implemented for all used combinations of memory types. NUM_WI_... is the number of Workitems per Workgroup in the respective dimension and NUM_WG_... is the number of Workgroups in the respective dimension. They are the parameters that can be auto tuned. All data is stored in one dimensional buffers, the defines (Lines 1-4) are used to emulate the behaviour of a multidimensional array. The $VIEW$ macro (Lines 10-12) takes the buffers storing the dataset, the predictions and the attribute indices and threshold values of the tree nodes to create the InputAtom for the current evaluation of *traverse*$_s$, this is the operation described by $VIEW_s$.

Listing 3.1: *step*$_s$ md_hom parameters

```
1  #define INSTANCES_PER_ITEM (NUM_INSTANCES/(NUM_WI_INSTANCES_SC*NUM_WG_INSTANCES_SC))
2  #define CHAINS_PER_ITEM (NUM_CHAINS/(NUM_WI_CHAINS_SC*NUM_WG_CHAINS_SC))
3  #define TREES_PER_ITEM (NUM_TREES/(NUM_WI_TREES_SC*NUM_WG_TREES_SC))
4
5  #define LB_IDX(I, L, C) ((NUM_LABELS * I + L) * NUM_CHAINS + C)
6  #define LO_IDX(I, C, T) ((NUM_WI_CHAINS_SC * I + C) * NUM_WI_TREES_SC + T)
7  #define IN_IDX(I, C, WG) ((NUM_CHAINS * I + C) * NUM_WG_TREES_SC + WG)
8  #define TREE_IDX(C,T) ((C * NUM_TREES + T) * NODES_PER_TREE)
9
10 #define VIEW(I,C,T) (InputAtom){
11 (Instance){data + I * (NUM_ATTRIBUTES+NUM_LABELS), labelBuffer + LB_IDX(I,0,C)},
12 (Tree){attributeIndices + TREE_IDX(C,T), nodeValues + TREE_IDX(C,T)}}
13
14 typedef struct OutputAtom
15 {
16         double result;
17         int vote;
18 }OutputAtom;
19
20 inline void addAssignOutputAtomsPrv(OutputAtom* a, OutputAtom* b)
21 {
22         a->vote += b->vote;
23         a->result += b->result;
24 }
25
26 inline void addAssignOutputAtomsLoc(local OutputAtom* a, local OutputAtom* b)
27 {
28         a->vote += b->vote;
29         a->result += b->result;
30 }
31
32 inline void addAssignOutputAtomsGlo(global OutputAtom* a, global OutputAtom* b)
33 {
34         a->vote += b->vote;
35         a->result += b->result;
36 }
37
```

```
typedef struct Instance
{
        global double* attributes;
        global double* labels;
}Instance;

typedef struct Tree
{
        global int* nodeIndices;
        global double* nodeValues;
}Tree;

typedef struct InputAtom
{
        Instance inst;
        Tree tree;
}InputAtom;

OutputAtom traverse(InputAtom input)
{
        ...
}
```

Listing 3.2 shows the OpenCL code for the first kernel for the *step*$_s$ MDH following the md_hom skeleton. The outer loops (Lines 18 and 22) iterate over the dimension with concatenation operators. The inner loop (Line 28) evaluates *traverse*$_s$ for all trees of current forest and the current instance that the current Workitem is responsible for. Afterwards the results are combined to a single result for the workitem in a local buffer. Afterwards the barrier (Line 39) makes all Workitems wait until all per Workitem results in the current Workgroup are available. Then the per Workitem results are reduced to a single result per Workgroup using parallel reduction (Line 51 to 59). Since classic parallel reduction requires a number of elements that is a power of two the per Workitem results are first reduced to the next lower power of two if neccessary (Lines 41-50). The per Workigroup result is then written to an intermediary buffer by one of the Workitems (Lines 61-66) which is then reduced to the final result by the second kernel similarly to the reduction step in the first. The program then continues with the next chains and instances in the chunk the current Workitem is responsible for.

Listing 3.2: *step*$_s$ md_hom

```
1  kernel void stepCalc(
2                  global double* nodeValues,
3                  global int* attributeIndices,
4                  global double* data,
5                  global double* labelBuffer,
6                  local OutputAtom* localBuffer,
7                  global OutputAtom* intermediateBuffer
8          )
9  {
10         int i_wg_instance = get_group_id(0);
11         int i_wg_chain = get_group_id(1);
12         int i_wg_tree = get_group_id(2);
13 
14         int i_wi_instance = get_local_id(0);
15         int i_wi_chain = get_local_id(1);
16         int i_wi_tree = get_local_id(2);
17 
18         for (int i = 0; i < INSTANCES_PER_ITEM; ++i)
```

```
    {
        int instance = i_wi_instance + i_wg_instance * NUM_WI_INSTANCES_SC +
                 i * NUM_WG_INSTANCES_SC * NUM_WI_INSTANCES_SC;
        for (int c = 0; c < CHAINS_PER_ITEM; ++c)
        {
            int chain = i_wi_chain + i_wg_chain * NUM_WI_CHAINS_SC +
                     c * NUM_WG_CHAINS_SC * NUM_WI_CHAINS_SC;
            OutputAtom res_prv = (OutputAtom) { 0, 0 };

            for (int t = 0; t < TREES_PER_ITEM; ++t)
            {
                int tree = i_wi_tree + i_wg_tree * NUM_WI_TREES_SC +
                        t * NUM_WG_TREES_SC * NUM_WI_TREES_SC;
                OutputAtom res = traverse(VIEW(instance, chain, tree));
                addAssignOutputAtomsPrv(&res_prv, &res);
            }

            int localIndex = LO_IDX(i_wi_instance, i_wi_chain, i_wi_tree);
            localBuffer[localIndex] = res_prv;

            barrier(CLK_LOCAL_MEM_FENCE);

#if (NUM_WI_TREES_SC & (NUM_WI_TREES_SC - 1)) == 0
            int t = NUM_WI_TREES_SC / 2;
#else
            int t = pow(2, floor(log2((float)NUM_WI_TREES_SC)));
            if (i_wi_tree < t && i_wi_tree + t < NUM_WI_TREES_SC) {
                localBuffer[i_wi_tree] += localBuffer[i_wi_tree + t];
            }
            t /= 2;
            barrier(CLK_LOCAL_MEM_FENCE);
#endif
            for (; t > 0; t /= 2)
            {
                if (i_wi_tree < t)
                {
                    addAssignOutputAtomsLoc(
                            localBuffer + localIndex,
                            localBuffer + localIndex + t);
                }
                barrier(CLK_LOCAL_MEM_FENCE);
            }

            if (i_wi_tree == 0)
            {
                int intermediateIndex = IN_IDX(instance, chain,
                                     i_wg_tree);
                intermediateBuffer[intermediateIndex] =
                                localBuffer[localIndex];
            }
            barrier(CLK_LOCAL_MEM_FENCE);
        }
    }
}
```

The *build* MDH is implemented using only the first kernel of the md_hom pattern because it is a special case where all operators are concatenation operators. The second kernel combines per Workgroup result of the first kernel which is not neccesarry for concatenations. The $final$ MDH is implemented like $step_s$.

3.2.4. Host Code

Listing 3.3 shows the part of the ECC classification algorithm presented in Algorithm 8 that is implemented on the host. The $step_s$ MDH is implemented using two kernels *stepCalcKernel* and *stepReduceKernel* according to the md_hom pattern. A very useful feature of the algorithm is that for every step in the chains only the trees of those forests need to be resident in the memory of the OpenCL device. The total memory used by the ECC model is proportional to $C \cdot L \cdot T \cdot (2^{d+1} - 1)$ where C, L, T is the number of chains, labels and trees per forest and d is the height of the trees. By only using the part of the model that is required for the current step the device memory can be smaller and the memory usage is independent from the dataset, because the number of labels does not affect the per step model size. This makes it possible to have a tradeoff between prediction performance and memory usage where the same parameters can be used across multiple datasets and there is no need to limit prediction performance for datasets with many labels because of memory concerns. Every loop iteration the iteration counter is set in the kernel (Line 3) afterwards the model needed for the step is written to device memory (Lines 4-7) and finally the two kernels for $step_s$ are executed (Lines 8 and 9). After the algortihm stepped through the chains the two kernels of the $final$ MDH are exectued (Lines 11 and 12) to do the final reduction.

Listing 3.3: Host part of ECC algorithm

```
 1  for (size_t chainIndex = 0; chainIndex < numLabels; ++chainIndex)
 2  {
 3          stepReduceKernel->SetArg(4, chainIndex);
 4          stepNodeIndexBuffer.writeFrom(nodeIndices + chainIndex * stepModelSize,
 5                                        stepModelSize * sizeof(int));
 6          stepNodeValueBuffer.writeFrom(nodeValues + chainIndex * stepModelSize,
 7                                        stepModelSize * sizeof(double));
 8          stepCalcKernel->execute();
 9          stepReduceKernel->execute();
10  }
11  finalCalcKernel->execute();
12  finalReduceKernel->execute();
```

To avoid having the entire model in the device memory for large models the *build* mh_hom is split into several runs. Since the *build* MDH builds trees in parallel and only concatenates the result it can be calculated in multiple executions of the kernel. Listing 3.4 shows the process. The algorithm loops over the runs, the input subsets are stored as indices in the complete training data set. The subsets are stored in the *instancesBuffer*. The part of the instances used for the current run are written to device memory (Lines 9-11) together with random seeds which are used for the random node and attribute selection (see section 2.1.6). The predicted labels for each tree are kept in a separate buffer which is stored as a whole in device memory, because it is comparatively small. When the buffers are written the kernel for the current run is executed (Line 15). Then the results are read back to the corresponding part of the output buffers (Lines 17-22).

Listing 3.4: Host part of ECC algorithm

```
 1  for (int tree = 0; tree < numChains * numTrees * numLabels; tree += treesPerRun)
 2  {
        buildKernel->SetArg(0, static_cast<int>(gidMultiplier * treesPerRun));

        for (size_t seed = 0; seed < treesPerRun; ++seed)
        {
                seeds[seed] = Util::randomInt(INT_MAX);
        }
        seedsBuffer.writeFrom(seeds, seedsBuffer.getSize());

        instancesBuffer.writeFrom(indicesList.data() +
                gidMultiplier * treesPerRun * maxSplits,
                treesPerRun * maxSplits * sizeof(int));

        buildKernel->execute();

        tmpNodeIndexBuffer.readTo(nodeIndices +
                gidMultiplier*treesPerRun*nodesPerTree,
                treesPerRun*nodesPerTree * sizeof(int));
        tmpNodeValueBuffer.readTo(nodeValues +
                gidMultiplier*treesPerRun*nodesPerTree,
                treesPerRun*nodesPerTree * sizeof(double));

        ++gidMultiplier;
}
```

3.2.5. Auto-Tuning

Parameter Space

The kernels of the md_hom pattern are parameterized by having parameters for the number of Workgroups and Workitems that the evaluations of the scalar functions are distributed between. The ECC training is a special case of a single dimensional MDH with a concatenation operator, which requires no second kernel and only has the number of Workgroups and Workitems in the one dimension as parameter, a total of 2. Both MDHs used in the ECC classification algorithm work with 3-dimensional input MDAs, $step_s$ works with an MDA of size $I \cdot C \cdot T$, $final$ with one of size $I \cdot C \cdot L$. The number of Workitems and Workgroups in 3 dimensions makes a total of 6 parameters for the first kernel. The second kernel reduces per Workgroup results of the first kernel for all dimensions that use a non concatenation operator and can only launch one Workgroup in that dimension. In both MDAs in the ECC algorithm 2 out of 3 operators are concatenations and the third dimension gets reduced. Therefore the second kernel has 5 Parameters for both MDHs, 2 for the number of Workgroups and Workitems in each concatenated dimension and 1 for the number of Workitems in the reduced dimension. The $step_s$ MDH is executed once per label, this makes a total of $L \cdot 11 + 11$ parameters for the classification. Using auto-tuning with very large parameter spaces is impractical as the time it takes to find good configurations can get very long, therefore several assumptions are made to reduce the dimensionality of the parameter space:

- **Assumption 1**: good parameters of the md_hom for $step_s$ are similar for every loop iteration, which makes it possible to find good parameters for one and use them for every iteration, this can be assumed because each iteration performs the same calculations only using different data

- **Assumption 2**: the runtime of the second kernel of each md_hom is small compared to the first and the same paramters for the concatenation dimensions will yield good enough results. The one reduction dimension in each MDH only adds only 1 extra parameter and can be tuned independently, this prooved to be a reasonable assumption during testing, where the second kernel usually only ran a fraction of the time of the first

- **Assumption 3**: $step_s$ and $final$ can be tuned independently because they are run completely sequentially

All assumptions together reduce the parameter space for each tuning run to a 7-dimensional plus a run with a 2-dimensional space for the build MDH.

Auto Tuning Framework

Auto-tuning is done using the Auto Tuning Framework (ATF) [9] which allows a generic method for auto-tuning applications. Implementing auto-tuning using ATF requires defining every parameter of the parameter space with possible values. Parameters can be grouped and additional constraints between the parameters within a group can be defined. At the core of the ATF tuning process is a function $f : P \longrightarrow \mathbb{R}$ where P is the parameter space, which ATF tries to minimize by evaluating $f(c)$ for different valid configurations $c \in P$.

Technique

Listing 3.5 shows the auto-tuning process using ATF using the $final$ md_hom as an example. First the parameters space is defined. The number of Workgroups in the instance dimension $NUM\_WG\_INSTANCES\_FC$ has to be in the range from 1 to I (Lines 6-10) and has to divide I evenly. The number of Workitems per Workgroup $NUM\_WI\_INSTANCES\_FC$ is in the same range but has to divide the chunk of instances a workgroup is responsible for evenly (Lines 21-26) and therefore needs to be in the same constraint group as $NUM\_WG\_INSTANCES\_FC$ (Line 40). The other dimensions are defined in the same way. The chains dimension is reduced in the second kernel and therefore has an extra parameter, that has to be between 1 and the number of Workgroups of the first kernel and has to divide it evenly (Line 32-36). After the parameter space is created it is used to create a tuner (Line 38). ATF offers multiple tuners, an exhaustive tuner, which tries every configuration in the parameter space, is chosen for small parameter spaces up to 25000 elements. If the parameter space is larger a tuner based on OpenTuner [10] is selected, which searches 25% of the parameter space. OpenTuner is a framework offering different simple search techniques as well as more complex meta techniques which use a combination of the simpler techniques to search the parameter space. Finally the tuner is used to optimize a function which executes the $final$ md_hom with the currently tested configuration the returns the runtime of the kernels.

Listing 3.5: Auto-tuning using ATF

```
auto tp_NUM_WG_CHAINS_FC =
        atf::tp("NUM_WG_CHAINS_FC", atf::interval(1, numChains),
        [&](auto tp_NUM_WG_CHAINS_FC) {
                return (numChains % tp_NUM_WG_CHAINS_FC) == 0;
         });
auto tp_NUM_WG_INSTANCES_FC =
        atf::tp("NUM_WG_INSTANCES_FC", atf::interval(1, numInstances),
        [&](auto tp_NUM_WG_INSTANCES_FC) {
                return (numInstances % tp_NUM_WG_INSTANCES_FC) == 0;
         });
auto tp_NUM_WG_LABELS_FC =
        atf::tp("NUM_WG_LABELS_FC", atf::interval(1, numLabels),
        [&](auto tp_NUM_WG_LABELS_FC) {
                return (numLabels % tp_NUM_WG_LABELS_FC) == 0;
        });
auto tp_NUM_WI_CHAINS_FC =
        atf::tp("NUM_WI_CHAINS_FC", atf::interval(1, numChains),
        [&](auto tp_NUM_WI_CHAINS_FC) {
                return ((numChains / tp_NUM_WG_CHAINS_FC) % tp_NUM_WI_CHAINS_FC) == 0;
        });
auto tp_NUM_WI_INSTANCES_FC =
        atf::tp("NUM_WI_INSTANCES_FC", atf::interval(1, numInstances),
        [&](auto tp_NUM_WI_INSTANCES_FC) {
                return ((numInstances / tp_NUM_WG_INSTANCES_FC)
                        % tp_NUM_WI_INSTANCES_FC) == 0;
        });
auto tp_NUM_WI_LABELS_FC =
        atf::tp("NUM_WI_LABELS_FC", atf::interval(1, numLabels),
        [&](auto tp_NUM_WI_LABELS_FC) {
                return ((numLabels / tp_NUM_WG_LABELS_FC) % tp_NUM_WI_LABELS_FC) == 0;
        });
auto tp_NUM_WI_CHAINS_FR =
        atf::tp("NUM_WI_CHAINS_FR", atf::interval(1, numChains),
        [&](auto tp_NUM_WI_CHAINS_FR) {
                return (tp_NUM_WG_CHAINS_FC % tp_NUM_WI_CHAINS_FR) == 0;
         });

auto tuner = make_tuner(
        G(tp_NUM_WG_CHAINS_FC, tp_NUM_WI_CHAINS_FC, tp_NUM_WI_CHAINS_FR),
        G(tp_NUM_WG_INSTANCES_FC, tp_NUM_WI_INSTANCES_FC),
        G(tp_NUM_WG_LABELS_FC, tp_NUM_WI_LABELS_FC));

auto best_config = tuner->operator()(
        std::bind(&ECCTuner::tuneClassifyFinalFunc, this, std::placeholders::_1));
```

4. Evaluation

4.1. eccCL

The results of the md_hom implementation are compared to *eccCL*, which is a OpenCL based ECC implementation introduced by Alexander Herbst [2]. It implements the build process by starting one Workitem per tree that has to be built and a variable number of Workgroups. For the classification process the entire ECC classifier is implemented inside a single kernel and one Workitem is launched per instance that has to be classified, which requires the entire ECC model to be present in device memory. During the research for this thesis an issue with the *eccCL* imlpementation was found which can be seen in listing 4.1. The implementation iterates over every tree in every forest in every chain (Loops in Lines 1,3 and 9). The votes for every tree are collected in the *results* buffer (Line 26) which is the final output buffer. After the votes for all trees of a forest are combined in the results buffer the input instances label is set so that the next forests in the chain can use it as input (Lines 34-41). The issue is that the results buffer only contains one element for every label in every instance, after the first chain predicted all labels the results buffer contains the values for those labels. The following chains now add their votes to the votes of all previous chains and set the inputs label according to that total sum. This means that all later chains depend on the predictions of the previous chains which goes against the idea of ECC. This is solved by introducing an extra variable which stores the vote sum of the current forest and the input is set based on that variable. After the input is set the accumulation variable is added to the result buffer. For fair comparison both versions of the kernel are considered in performance evaluations.

Listing 4.1: Original *eccCL* implementation

```
1      for(int ensembleIndex = 0; ensembleIndex < numEnsembles; ++ensembleIndex)
2      {
3          for(int chainIndex = 0; chainIndex < numChains; ++chainIndex)
4          {
5
6              int label = labelOrders[numChains * ensembleIndex + chainIndex];
7              int resultIndex = (gid * numChains) + label;
8
9              for(int treeIndex = 0; treeIndex < forestSize; ++treeIndex)
10             {
11                 double value = traverse(
12                     data,
13                     nodeValues,
14                     forestSize,
15                     maxLevel,
16                     nodesPerTree,
17                     attributeIndices,
18                     gid,
                    numValues ,
                    treeIndex ,
                    chainIndex ,
                    numChains ,
                    ensembleIndex
                    );

                results[resultIndex] += value;

                if(value != 0)
                {
                    ++votes[resultIndex];
                }
            }

            if(results[resultIndex] > 0)
            {
                data[numValues * gid + numAttributes + label] = 1;
            }
            else
            {
                data[numValues * gid + numAttributes + label] = 0;
            }

        }
    }
```

4.2. Measurement

Performance measurements for the md_hom algorithm were done by first auto-tuning all parameters and then using those to measure runtime. The runtime is split into time spent running the OpenCL kernels, time spent transfering data and additional overhead. The overhead is measured by calculating the total runtime on the host and substracting kernel and transfer times. The total time does not include setting up kernels and writing buffers that do not change during the algorithm, but does include reading result data back and writing buffers that are written several times. All measurements were done once using a NVIDIA Tesla k20m as OpenCL device as well as 32 Intel Broadwell cores. Several datasets from the Mulan sample datasets [11] were evaluated to get measurements for datasets of different sizes and different label counts. Mulan is a java based machine learning library. All measurements are made using the following parameters:

- 64 Chains per ECC
- 32 Trees per Forest
- a maximum of 250000000 tree nodes per run of the build kernel

For comparison the runtime of *eccCL* is measured using the same parameters. *eccCL* is run using custom host code. For the number of Workgroups during classification the first divider of the global size which is smaller or equal to 32 was chosen. The number of Workgroups for the training process is equal to the number of labels. Only kernel and transfer time are measured, since *eccCL* implements the entire algorithm inside a single kernel, whereas md_hom implements small parts of the algorithm on the host. *eccCL* requires having the entire model in device memory at once, which makes it impossible to classify some of the larger datasets, especially those with many labels. For those datasets the *eccCL* measurement is missing and only the md_hom measurements are shown. Additionally the datasets *mediamill* and *Corel5k* could not be tuned for the Intel platform in reasonable time, it is therefore only showing results for the NVIDIA platform.

4.3. Results

The measurements (see section A) show that the presented md_hom implementation can compete with *eccCL*. The transfer times are similar on the same platform, but the kernel performance is usually better, especially for classification. Results for the build process often show similar performance on the same platform for the md_hom implementation and *eccCL*, this could be because both implementations parallelize the build process in a similar way. This can be seen in figure 4.1, in the case of the *scene* dataset *eccCL* performs almost identical to the md_hom implementation on the Intel platform and is slightly slower on NVIDIA. Most datasets perform similarly. Performance gains over *eccCL* might be due to auto-tuning the parameters. It is also worth noting that the Intel platform performs better on the build process than the NVIDIA platform in all tested cases, which is clearly visible in the *scene* dataset.

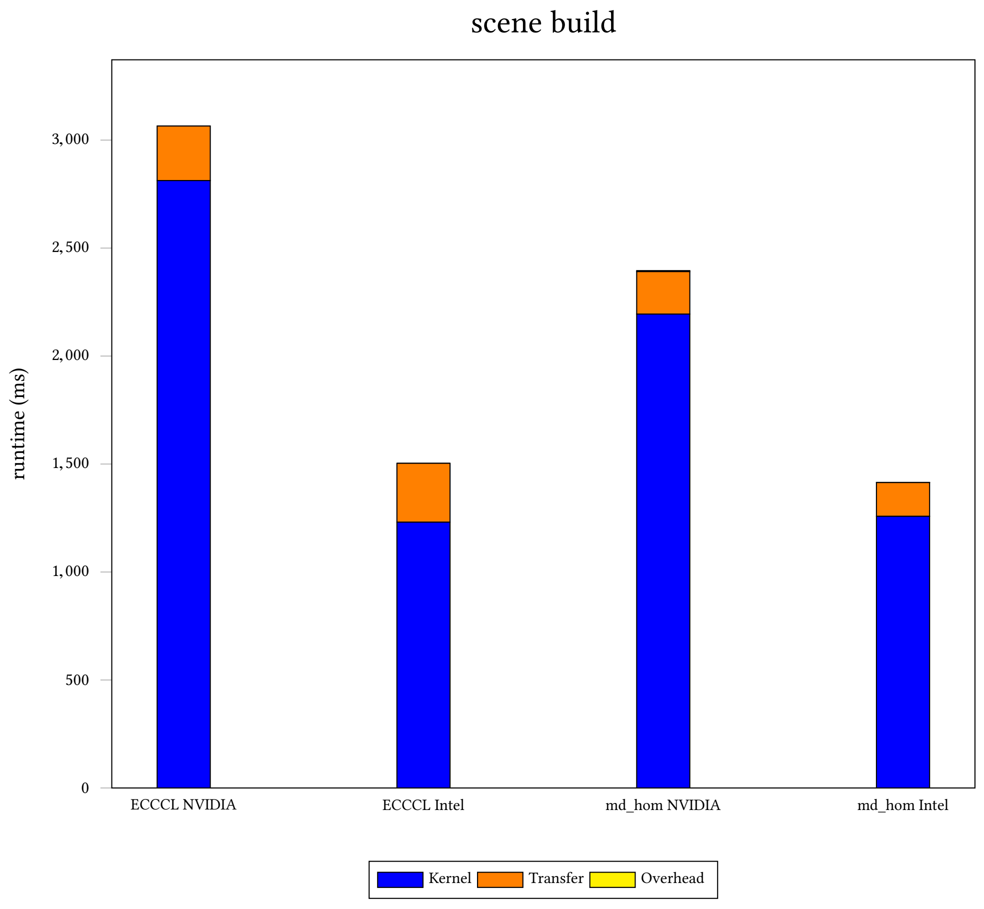


Figure 4.1.: Build performance with the scene dataset

The classification implementation using md_hom can, depending on the tuning parameters, use more Workitems then *eccCL* which only uses as many Workitems as there are instances to be classified. Parallelization is also used in different ways and to a flexible degree which, combined with auto-tuning has the potential to use the hardware more effectively. The md_hom implementation performs better than *eccCL* in classification in all cases, especially on the NVIDIA platform, where the md_hom implementation is 8 times faster in case of the *scene* and up to 55 times faster with the *flags* dataset. The classification performance using the *scene* dataset is show in figure ??. The transfer times are a significant part of the calculation time and often cause the Intel platform to perform better even though kernel times are often shorter on the NVIDIA platform. In general the fixed *eccCL* classification performs very similarly and often very slightly better than the original kernel.

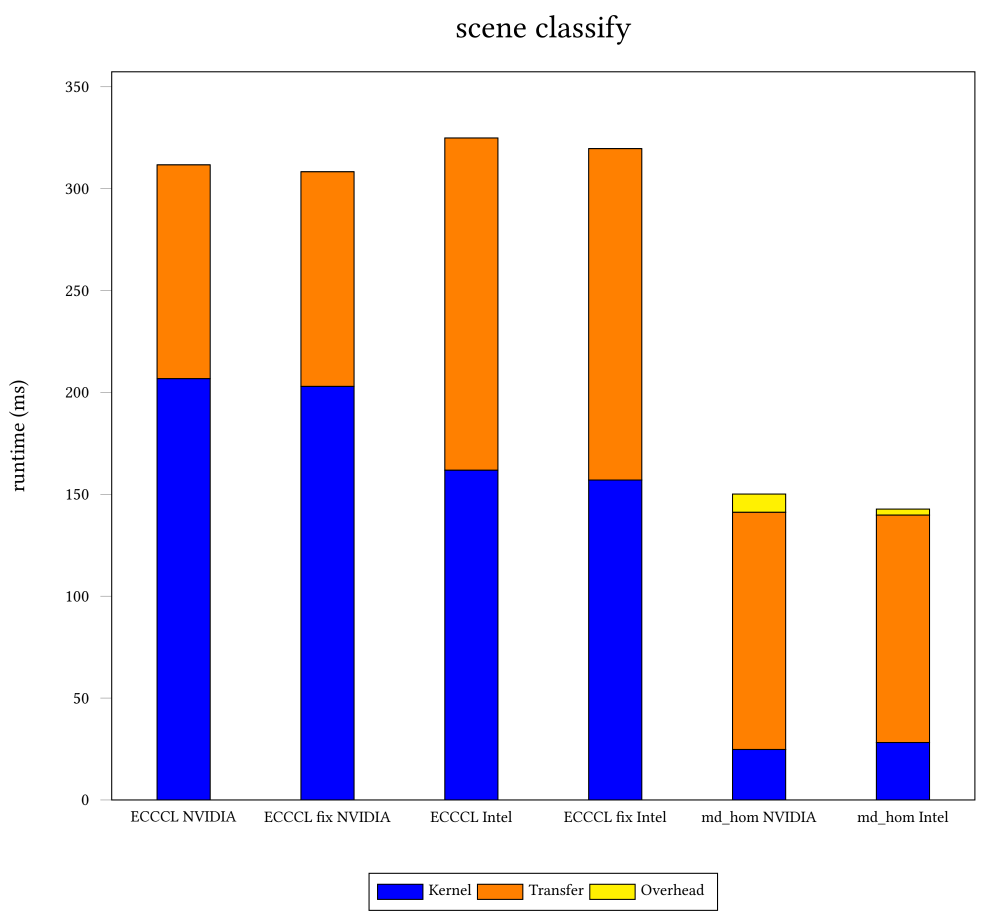


Figure 4.2.: Classification performance with the scene dataset

The other datasets show similar trends.

4.4. Further Work

The md_hom implementation can be used as a starting point for high perfromance ECC implementations. There are however further improvements that could be made for practical usage. The presented implementation is using ECC only with Random Forests of ID3 classifier trees. A general implementation of ECC as a programming pattern based on md_hom with the ability to quickly interchange the base classifier could be very useful. The implementation of the ID3 build algorithm, which originally comes form eccCL, has potential to be parallelized further. The leaf node training involves parallel traversal of the tree with multiple instances similar to classification, which can be parallelized similarly to the classification. The training of the internal nodes of the same level could be run in parallel as well. Implementing this using the md_hom pattern might be challenging as that would be a scalar function which is itself parallel, this might mean that the training process would have to be split into multiple md_homs similar to the classification. The limitation of requiring fixed depth trees could be removed to get closer to the original ID3 presented by Quinlan and remove some of the complexity in the way the votes are handled. Another practical consideration is that the auto-tuning takes alot of time and therefore only makes sense if the best parameters found are used often, especially the number of instances to be classifier might change frequently. Padding the input dataset to a fixed size could solve this issue.

Literature

[1] Mona Riemenschneider u. a. “Exploiting HIV-1 protease and reverse transcriptase cross-resistance information for improved drug resistance prediction by means of multi-label classification”. In: *BioData Mining* 9.1 (Feb. 2016), S. 10. ISSN: 1756-0381. DOI: 10.1186/s13040-016-0089-1. URL: https://doi.org/10.1186/s13040-016-0089-1 (siehe S. 1).

[2] Mona Riemenschneider u. a. “eccCL: parallelized GPU implementation of Ensemble Classifier Chains”. In: *BMC Bioinformatics* 18.1 (2017), 371:1–371:4. DOI: 10.1186/s12859-017-1783-9. URL: https://doi.org/10.1186/s12859-017-1783-9 (siehe S. 1, 16, 24).

[3] Jesse Read u. a. “Classifier chains for multi-label classification”. In: *Machine Learning* 85.3 (Juni 2011), S. 333. ISSN: 1573-0565. DOI: 10.1007/s10994-011-5256-5. URL: https://doi.org/10.1007/s10994-011-5256-5 (siehe S. 2).

[4] Leo Breiman. “Random Forests”. In: *Machine Learning* 45.1 (Okt. 2001), S. 5–32. ISSN: 1573-0565. DOI: 10.1023/A:1010933404324. URL: https://doi.org/10.1023/A:1010933404324 (siehe S. 4).

[5] J. R. Quinlan. “Induction of decision trees”. In: *Machine Learning* 1.1 (März 1986), S. 81–106. ISSN: 1573-0565. DOI: 10.1007/BF00116251. URL: https://doi.org/10.1007/BF00116251 (siehe S. 4).

[6] M. J. Flynn. “Very high-speed computing systems”. In: *Proceedings of the IEEE* 54.12 (Dez. 1966), S. 1901–1909. ISSN: 0018-9219. DOI: 10.1109/PROC.1966.5273 (siehe S. 7).

[7] Gorlatch S. Rasch A. “Multi-dimensional Homomorphisms and Their Implementation in OpenCL”. In: (2018). ISSN: 1573-7640. URL: https://doi.org/10.1007/s10766-017-0508-z (siehe S. 9, 11).

[8] Khronos OpenCL Working Group. *The OpenCL Specification.* Juli 2015. URL: https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf (siehe S. 15, 16).

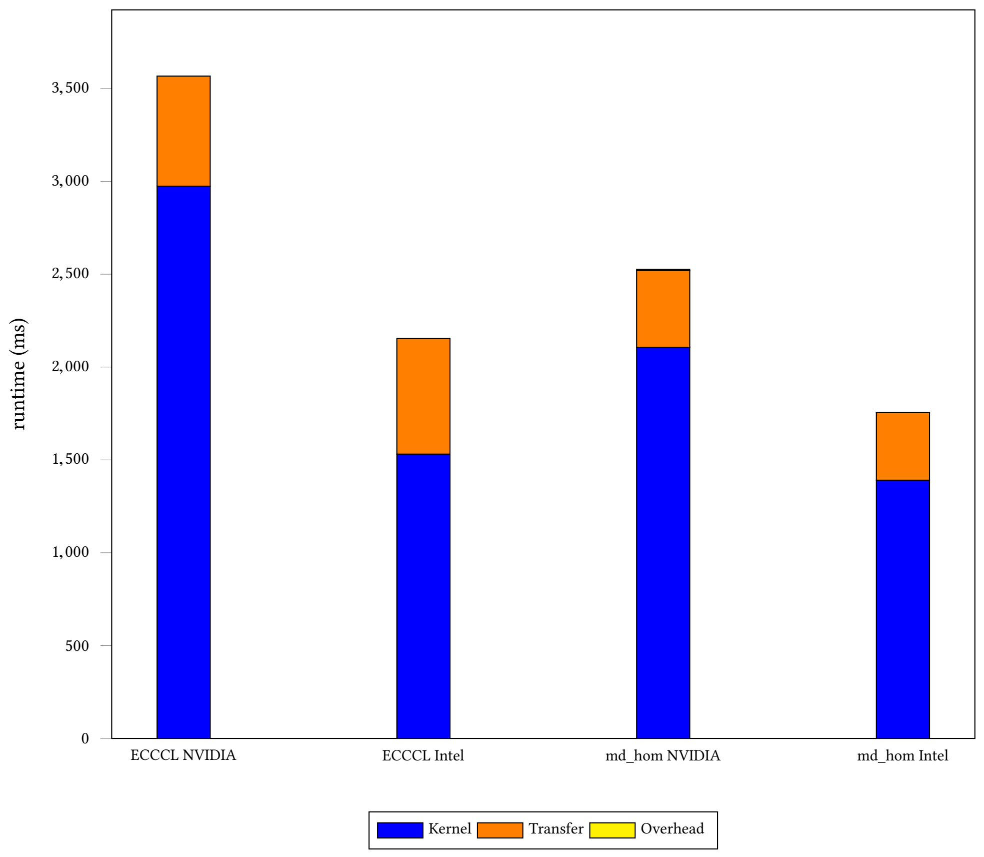
[9] A. Rasch, M. Haidl und S. Gorlatch. “ATF: A Generic Auto-Tuning Framework”. In: *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS).* Dez. 2017, S. 64–71. DOI: 10.1109/HPCC-SmartCity-DSS.2017.9 (siehe S. 22).

[10] J. Ansel u. a. “OpenTuner: An extensible framework for program autotuning”. In: *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. Aug. 2014, S. 303–315. DOI: `10.1145/2628071.2628092` (siehe S. 22).

[11] URL: `http://mulan.sourceforge.net/datasets-mlc.html` (siehe S. 25).

A. Data

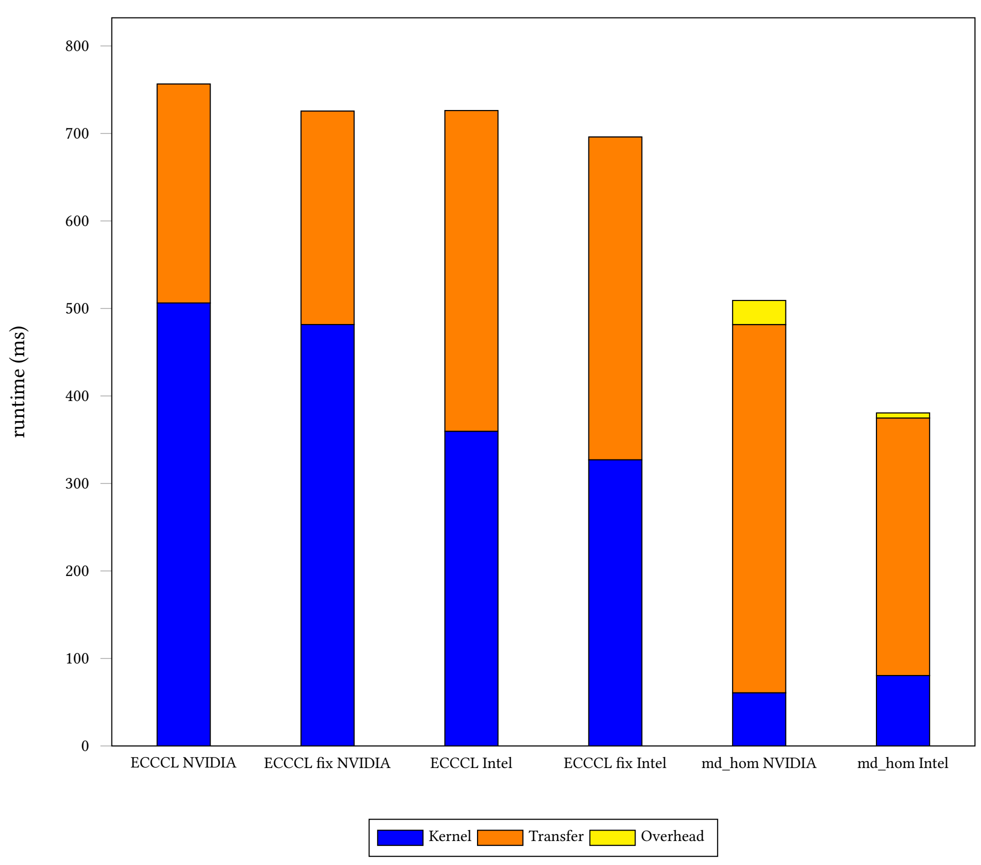
yeast build



- **Instances**: 2417, 1619 for training, 798 for evaluation
- **Attributes**: 103
- **Labels**: 14

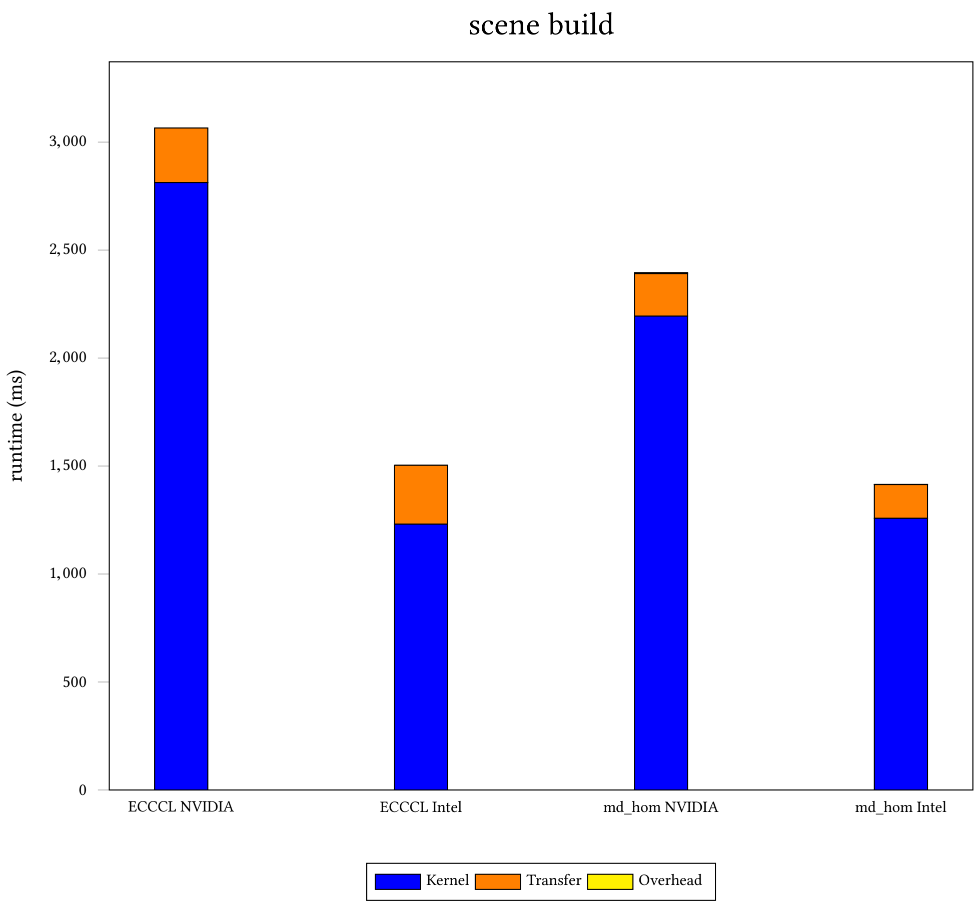
yeast build	Kernel	Transfer	Overhead
ECCCL NVIDIA	2,973.130 ms	593.810 ms	0.000 ms
ECCCL Intel	1,530.710 ms	622.486 ms	0.000 ms
md_hom NVIDIA	2,106.180 ms	413.549 ms	6.091 ms
md_hom Intel	1,390.400 ms	364.108 ms	1.292 ms

yeast classify



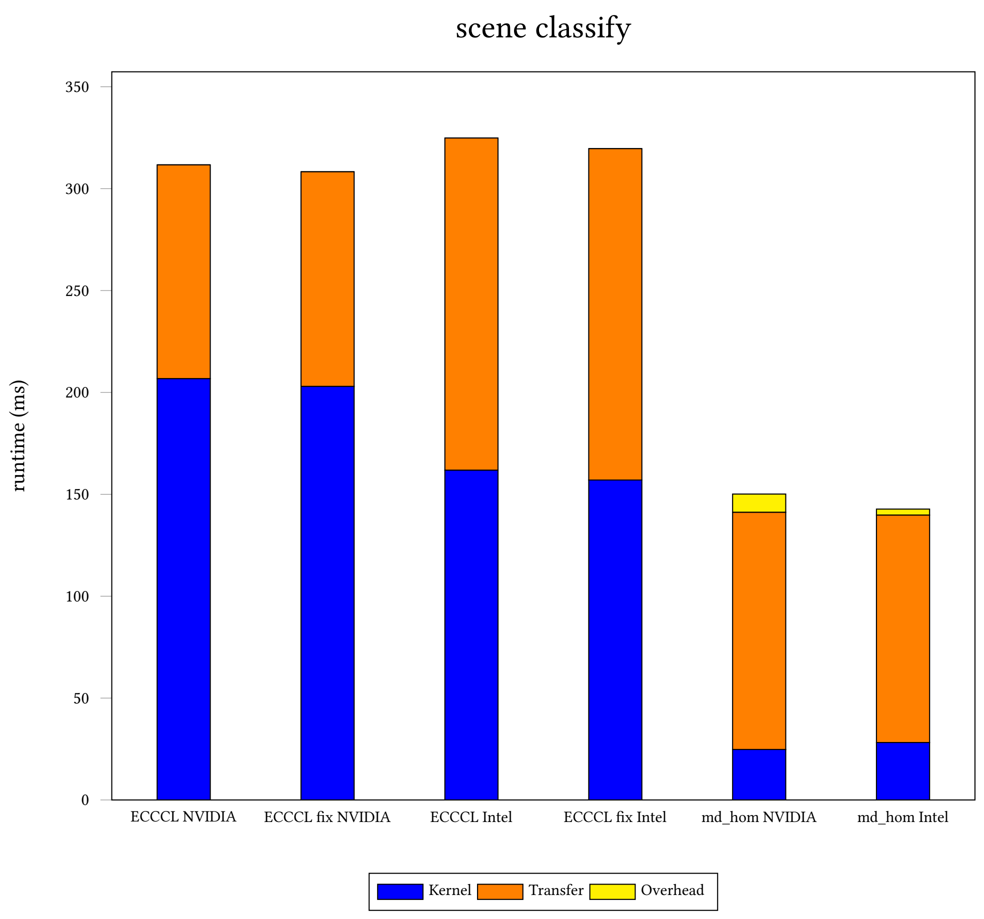
- **Instances**: 2417, 1619 for training, 798 for evaluation
- **Attributes**: 103
- **Labels**: 14

yeast classify	Kernel	Transfer	Overhead
ECCCL NVIDIA	506.244 ms	250.281 ms	0.000 ms
ECCCL fix NVIDIA	481.601 ms	243.984 ms	0.000 ms
ECCCL Intel	359.513 ms	366.661 ms	0.000 ms
ECCCL fix Intel	326.969 ms	369.012 ms	0.000 ms
md_hom NVIDIA	60.666 ms	420.852 ms	27.606 ms
md_hom Intel	80.456 ms	294.299 ms	5.841 ms



- **Instances**: 2407, 1612 for training, 795 for evaluation
- **Attributes**: 294
- **Labels**: 6

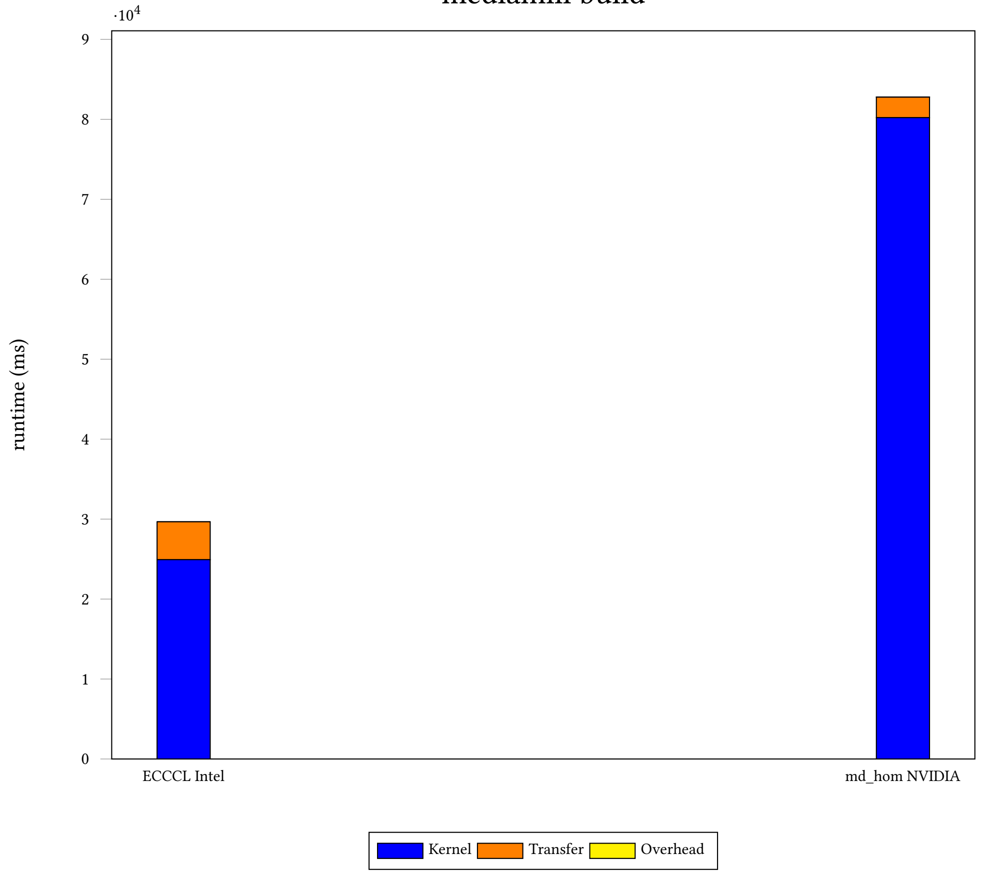
scene build	Kernel	Transfer	Overhead
ECCCL NVIDIA	2,812.510 ms	252.703 ms	0.000 ms
ECCCL Intel	1,230.650 ms	272.745 ms	0.000 ms
md_hom NVIDIA	2,194.150 ms	196.885 ms	4.435 ms
md_hom Intel	1,257.770 ms	156.454 ms	0.666 ms



- **Instances**: 2407, 1612 for training, 795 for evaluation
- **Attributes**: 294
- **Labels**: 6

scene classify	Kernel	Transfer	Overhead
ECCCL NVIDIA	206.748 ms	104.930 ms	0.000 ms
ECCCL fix NVIDIA	202.940 ms	105.350 ms	0.000 ms
ECCCL Intel	161.804 ms	163.060 ms	0.000 ms
ECCCL fix Intel	157.007 ms	162.638 ms	0.000 ms
md_hom NVIDIA	24.767 ms	116.386 ms	8.972 ms
md_hom Intel	28.188 ms	111.603 ms	2.954 ms

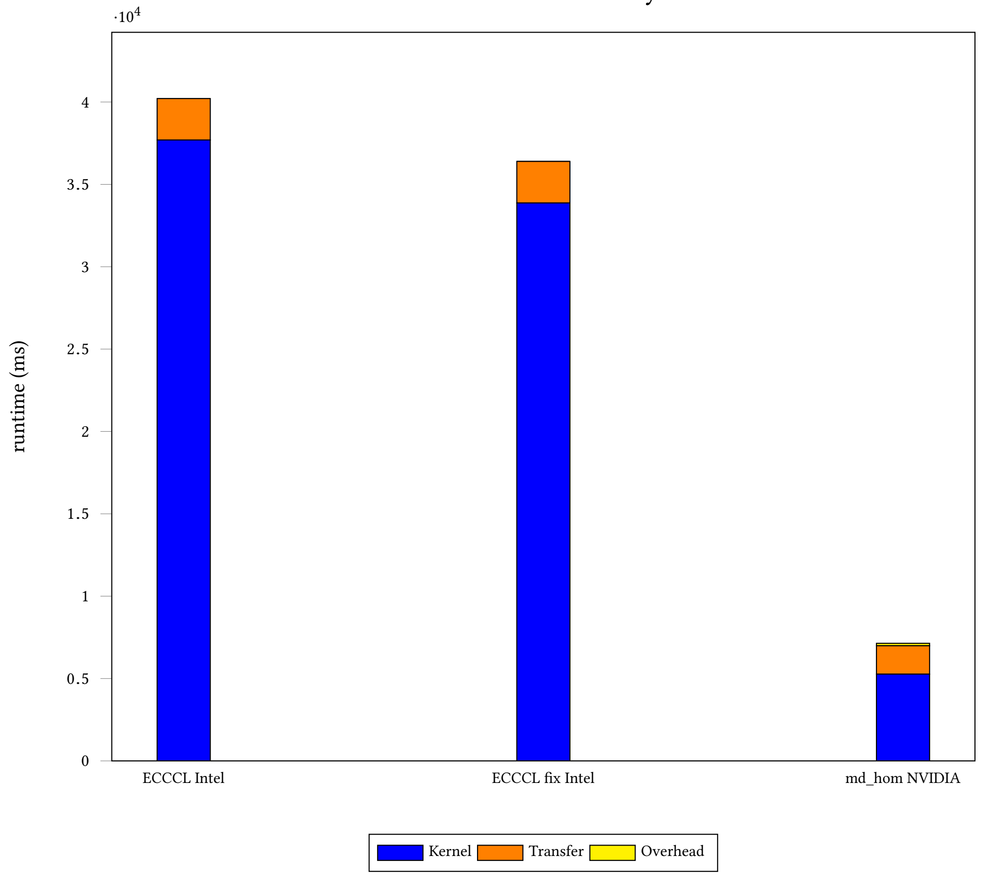
mediamill build



- **Instances**: 43907, 29417 for training, 14490 for evaluation
- **Attributes**: 120
- **Labels**: 101

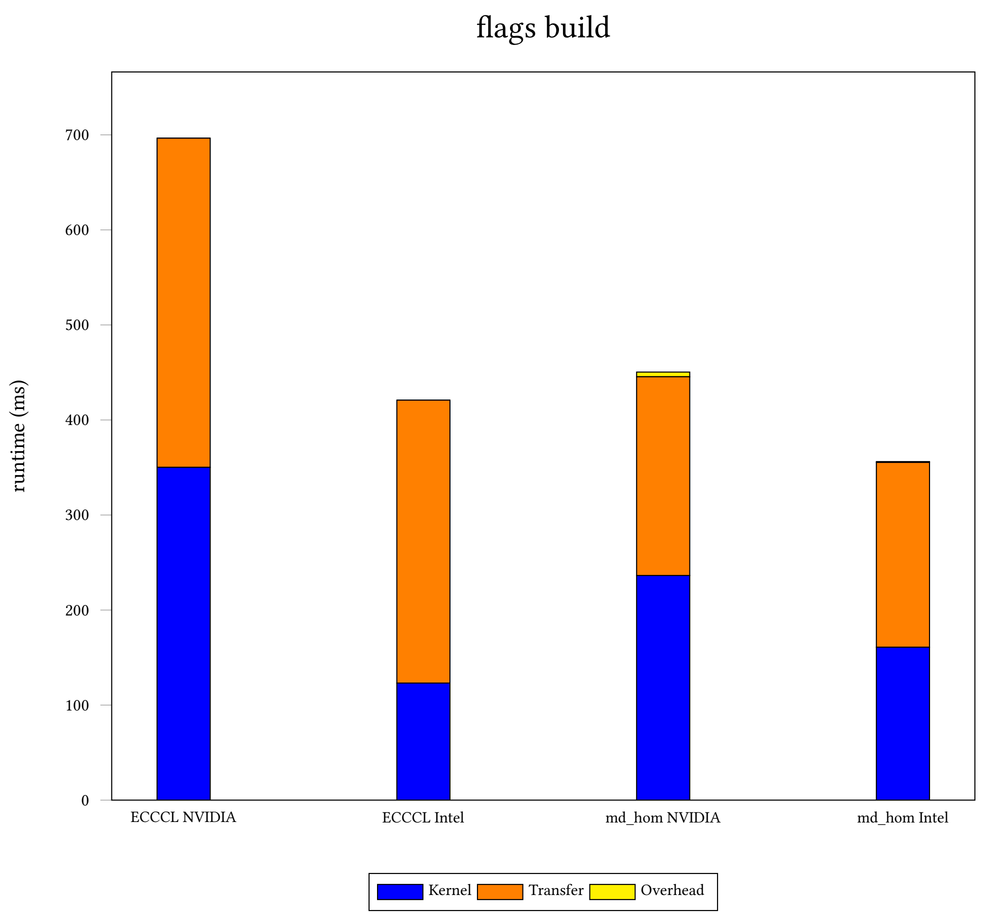
mediamill build	Kernel	Transfer	Overhead
ECCCL Intel	24,928.600 ms	4,737.962 ms	0.000 ms
md_hom NVIDIA	80,211.100 ms	2,562.089 ms	18.011 ms

mediamill classify



- **Instances**: 43907, 29417 for training, 14490 for evaluation
- **Attributes**: 120
- **Labels**: 101

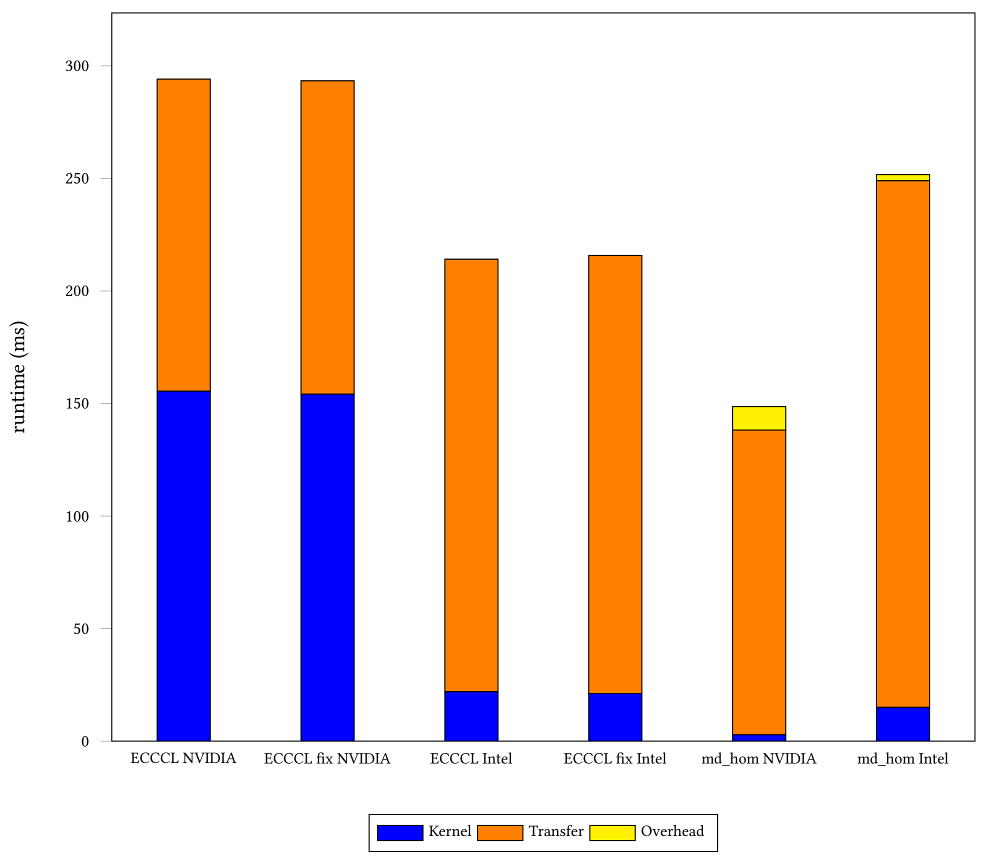
mediamill classify	Kernel	Transfer	Overhead
ECCCL Intel	37,693.200 ms	2,517.039 ms	0.000 ms
ECCCL fix Intel	33,870.400 ms	2,523.776 ms	0.000 ms
md_hom NVIDIA	5,265.553 ms	1,725.712 ms	148.291 ms



- **Instances**: 194, 129 for training, 65 for evaluation
- **Attributes**: 19
- **Labels**: 7

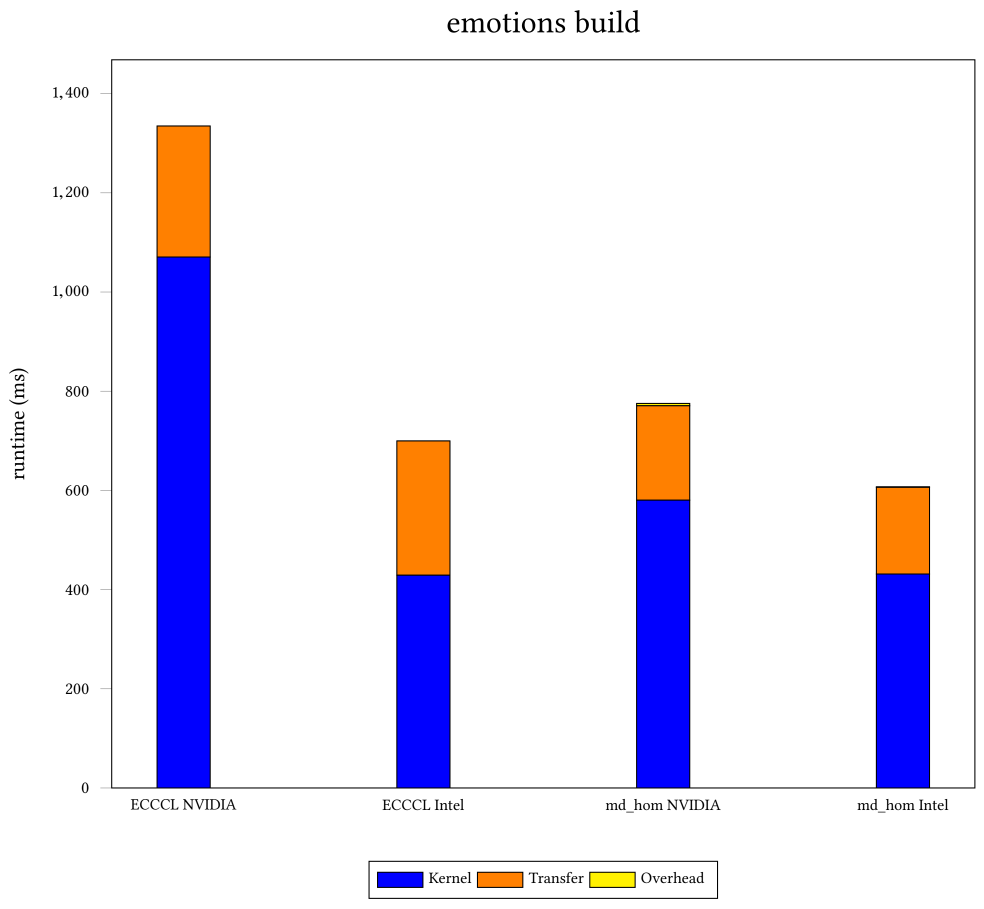
flags build	Kernel	Transfer	Overhead
ECCCL NVIDIA	350.136 ms	346.398 ms	0.000 ms
ECCCL Intel	123.118 ms	297.752 ms	0.000 ms
md_hom NVIDIA	236.351 ms	209.120 ms	4.914 ms
md_hom Intel	160.816 ms	194.531 ms	0.840 ms

flags classify



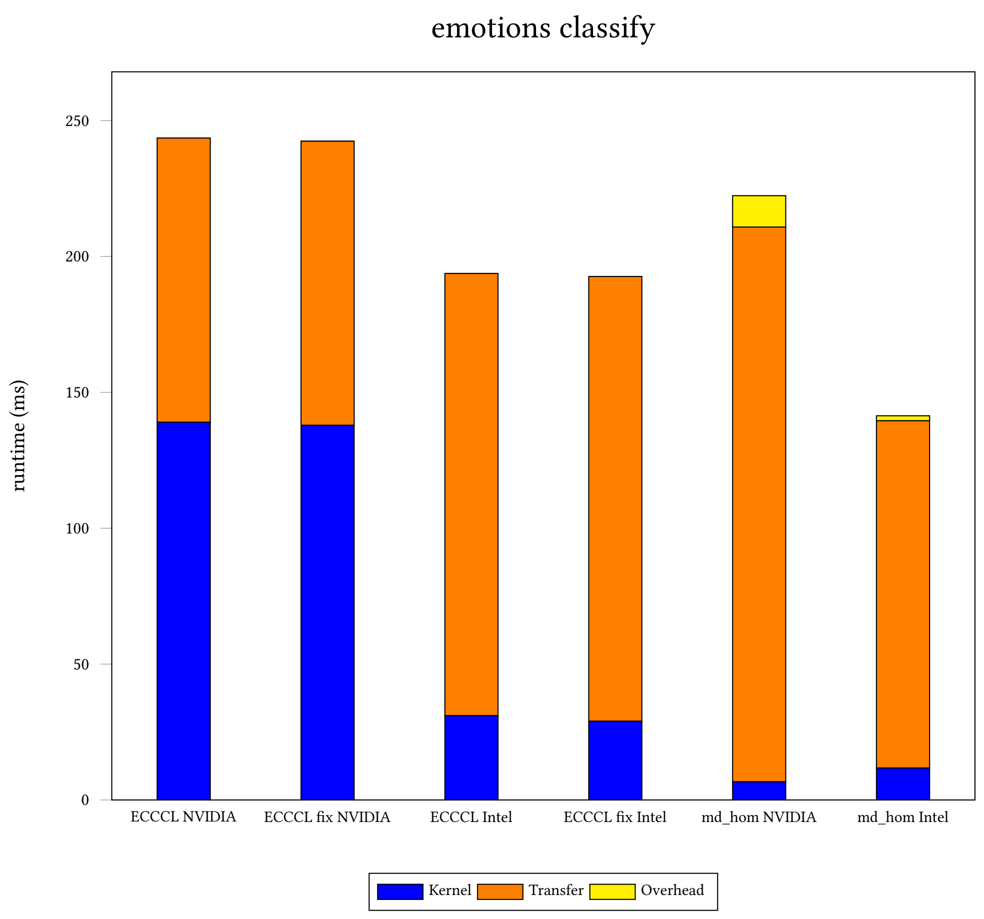
- **Instances**: 194, 129 for training, 65 for evaluation
- **Attributes**: 19
- **Labels**: 7

flags classify	Kernel	Transfer	Overhead
ECCCL NVIDIA	155.477 ms	138.639 ms	0.000 ms
ECCCL fix NVIDIA	154.140 ms	139.228 ms	0.000 ms
ECCCL Intel	21.954 ms	192.146 ms	0.000 ms
ECCCL fix Intel	21.147 ms	194.627 ms	0.000 ms
md_hom NVIDIA	2.815 ms	135.379 ms	10.435 ms
md_hom Intel	15.000 ms	233.976 ms	2.730 ms



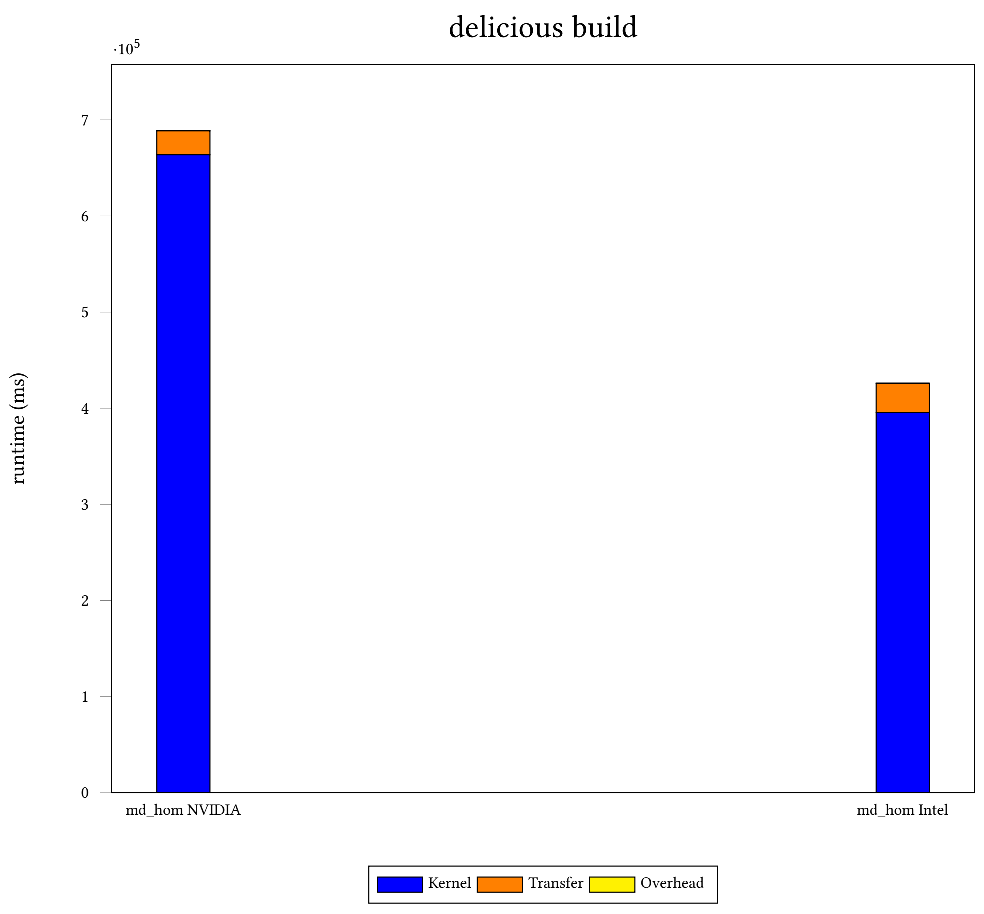
- **Instances**: 593, 397 for training, 196 for evaluation
- **Attributes**: 72
- **Labels**: 6

emotions build	Kernel	Transfer	Overhead
ECCCL NVIDIA	1,070.570 ms	264.191 ms	0.000 ms
ECCCL Intel	428.988 ms	270.756 ms	0.000 ms
md_hom NVIDIA	580.281 ms	190.402 ms	4.658 ms
md_hom Intel	431.394 ms	174.786 ms	1.159 ms



- **Instances**: 593, 397 for training, 196 for evaluation
- **Attributes**: 72
- **Labels**: 6

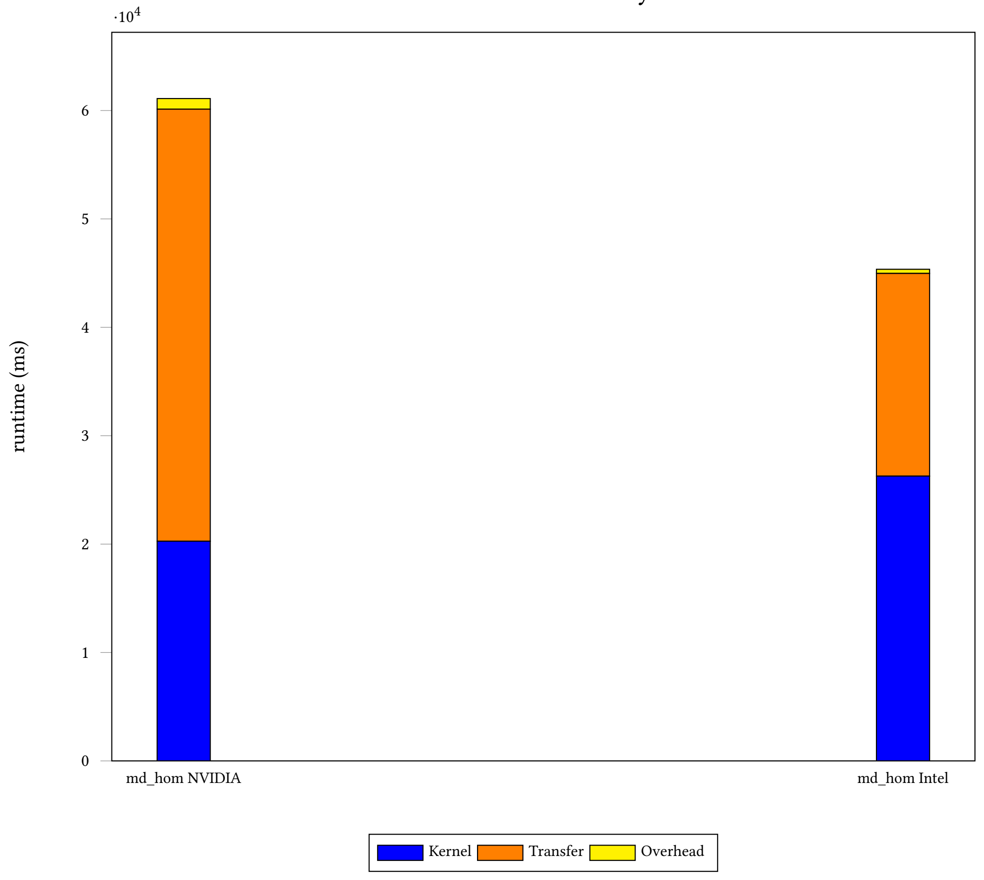
emotions classify	Kernel	Transfer	Overhead
ECCCL NVIDIA	138.997 ms	104.599 ms	0.000 ms
ECCCL fix NVIDIA	137.862 ms	104.558 ms	0.000 ms
ECCCL Intel	31.025 ms	162.717 ms	0.000 ms
ECCCL fix Intel	28.982 ms	163.646 ms	0.000 ms
md_hom NVIDIA	6.675 ms	204.155 ms	11.534 ms
md_hom Intel	11.772 ms	127.763 ms	1.850 ms



- **Instances**: 16105, 10790 for training, 5315 for evaluation
- **Attributes**: 500
- **Labels**: 983

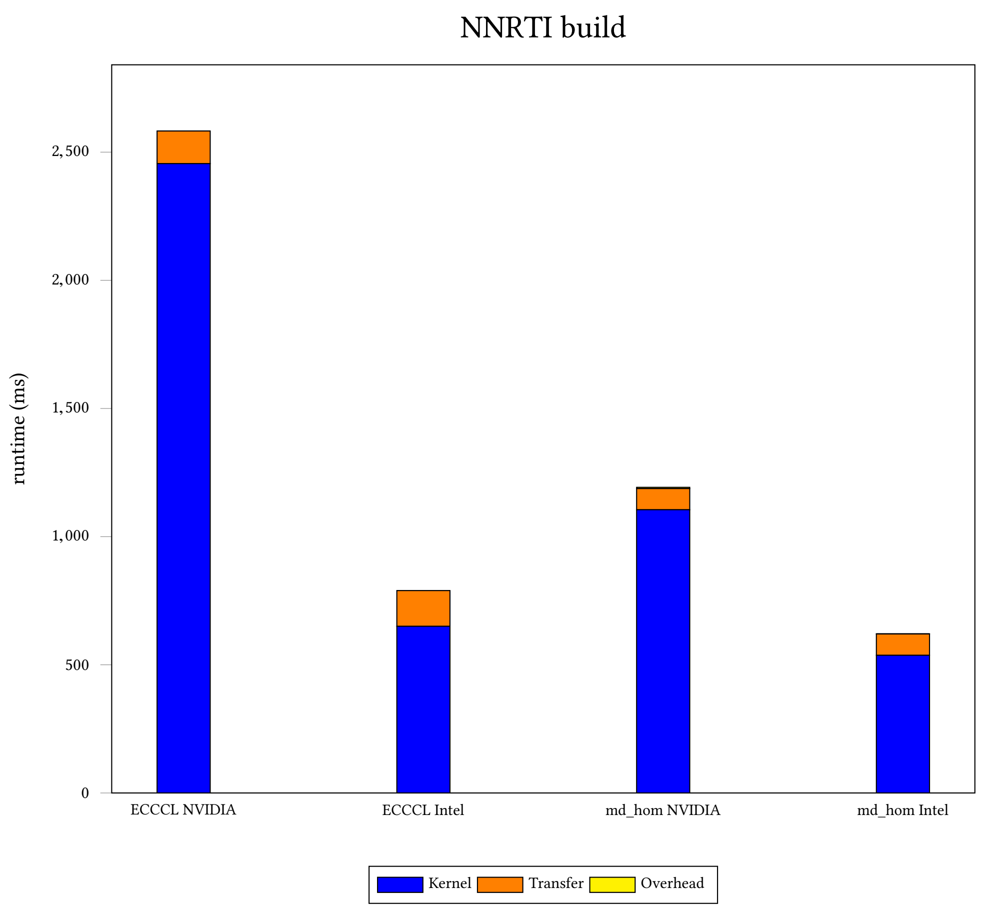
delicious build	Kernel	Transfer	Overhead
md_hom NVIDIA	663,651.000 ms	24,892.634 ms	130.366 ms
md_hom Intel	395,694.000 ms	30,333.199 ms	58.801 ms

delicious classify



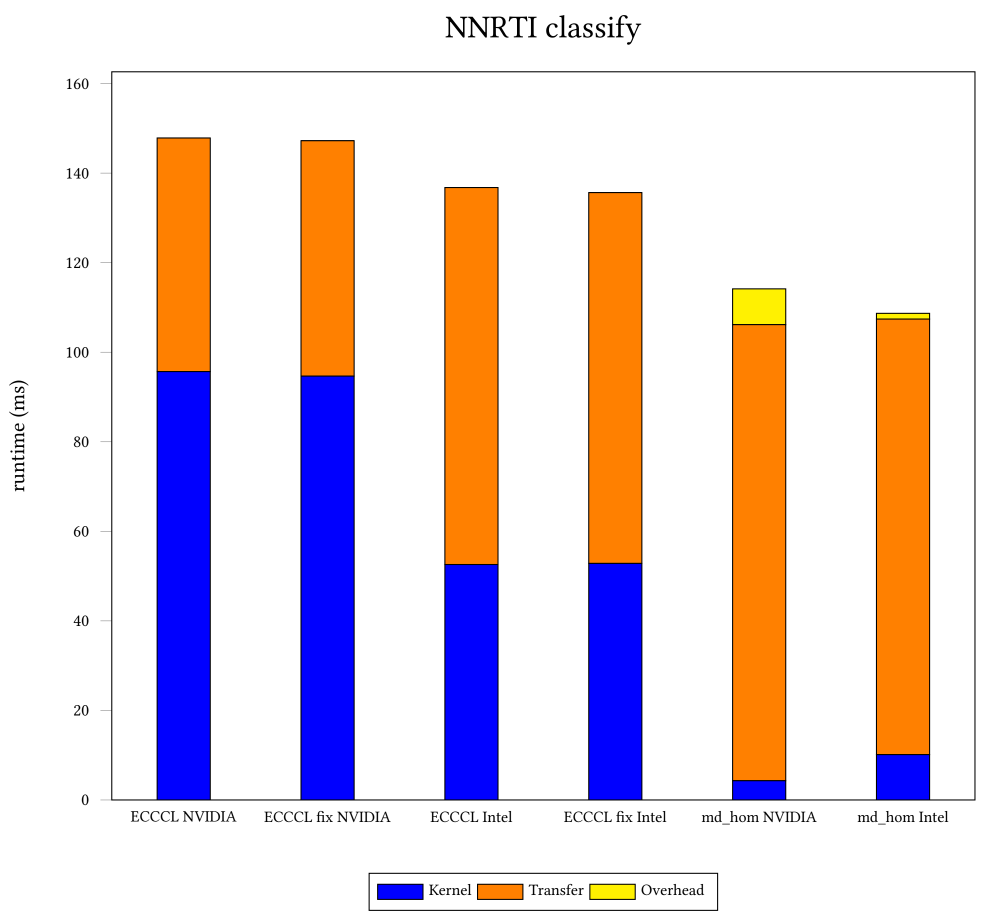
- **Instances**: 16105, 10790 for training, 5315 for evaluation
- **Attributes**: 500
- **Labels**: 983

delicious classify	Kernel	Transfer	Overhead
md_hom NVIDIA	20,265.886 ms	39,860.880 ms	976.697 ms
md_hom Intel	26,272.084 ms	18,700.038 ms	382.559 ms



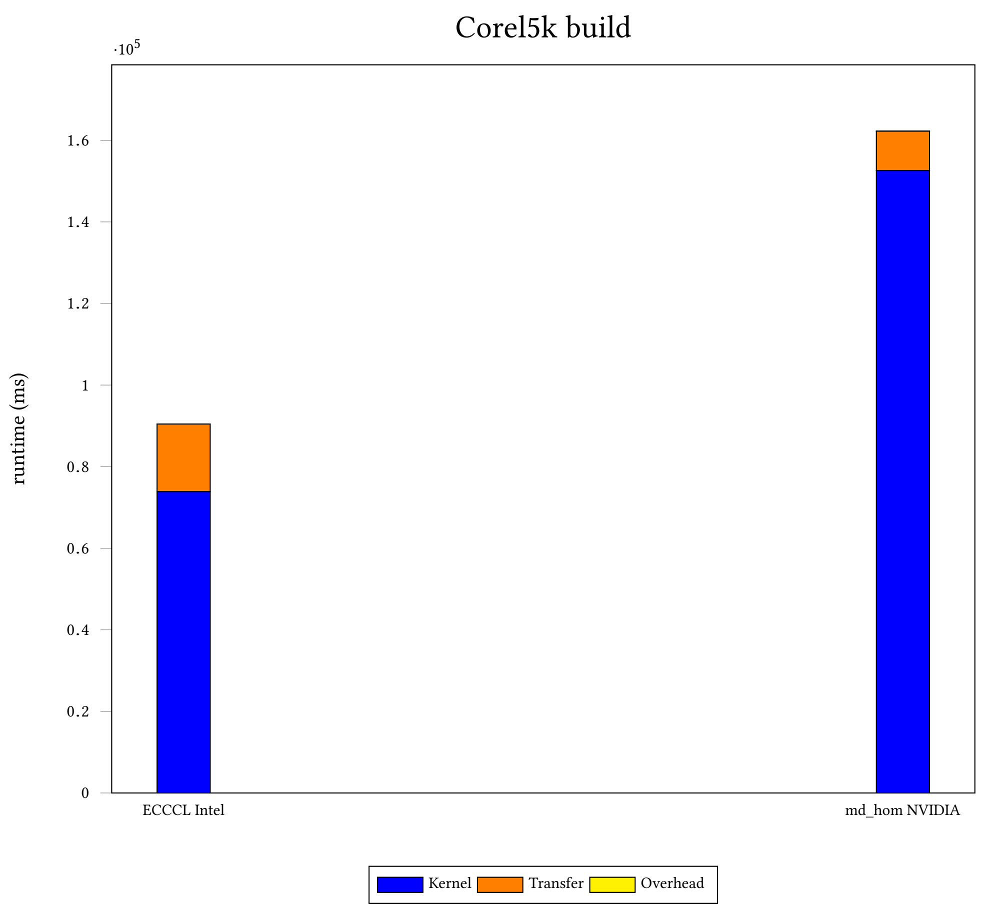
- **Instances**: 715, 479 for training, 236 for evaluation
- **Attributes**: 241
- **Labels**: 3

NNRTI build	Kernel	Transfer	Overhead
ECCCL NVIDIA	2,455.250 ms	127.255 ms	0.000 ms
ECCCL Intel	650.107 ms	139.451 ms	0.000 ms
md_hom NVIDIA	1,104.960 ms	82.580 ms	4.820 ms
md_hom Intel	537.019 ms	83.303 ms	0.527 ms



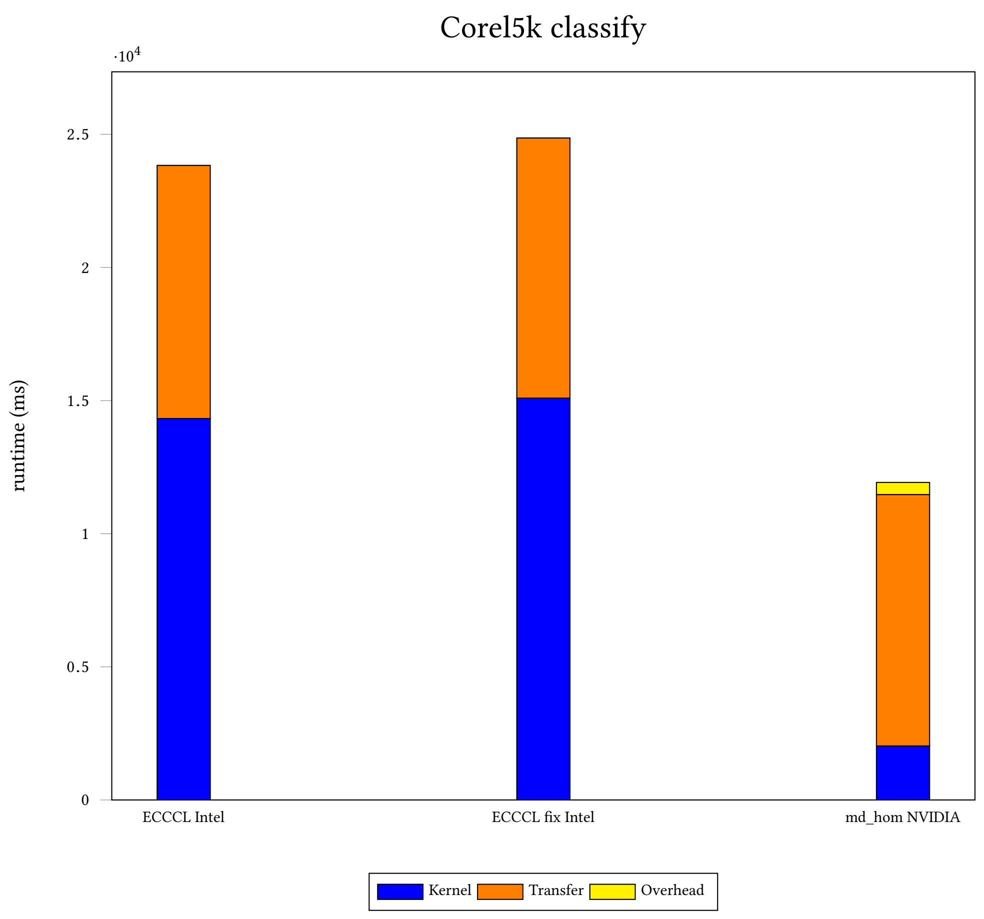
- **Instances**: 715, 479 for training, 236 for evaluation
- **Attributes**: 241
- **Labels**: 3

NNRTI classify	Kernel	Transfer	Overhead
ECCCL NVIDIA	95.678 ms	52.166 ms	0.000 ms
ECCCL fix NVIDIA	94.665 ms	52.571 ms	0.000 ms
ECCCL Intel	52.585 ms	84.194 ms	0.000 ms
ECCCL fix Intel	52.831 ms	82.820 ms	0.000 ms
md_hom NVIDIA	4.302 ms	101.858 ms	7.992 ms
md_hom Intel	10.132 ms	97.270 ms	1.281 ms



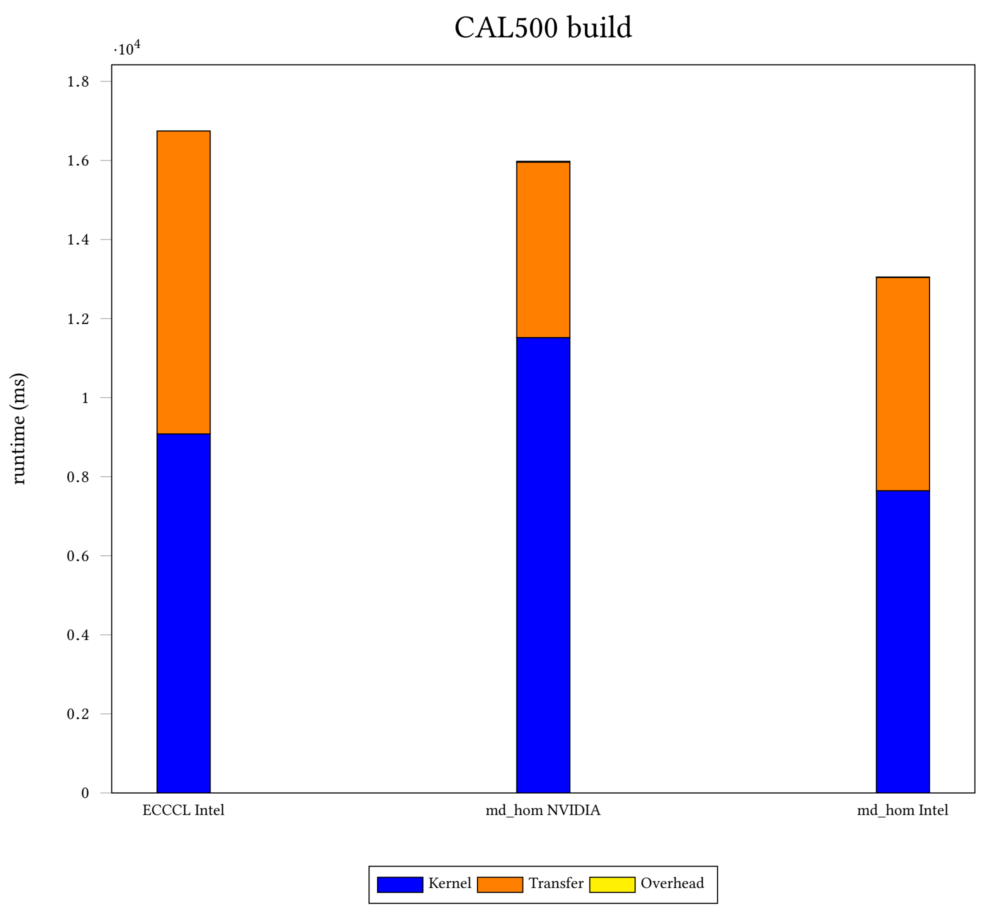
- **Instances**: 5000, 3350 for training, 1650 for evaluation
- **Attributes**: 499
- **Labels**: 374

Corel5k build	Kernel	Transfer	Overhead
ECCCL Intel	73,883.800 ms	16,566.729 ms	0.000 ms
md_hom NVIDIA	152,597.000 ms	9,641.287 ms	48.713 ms



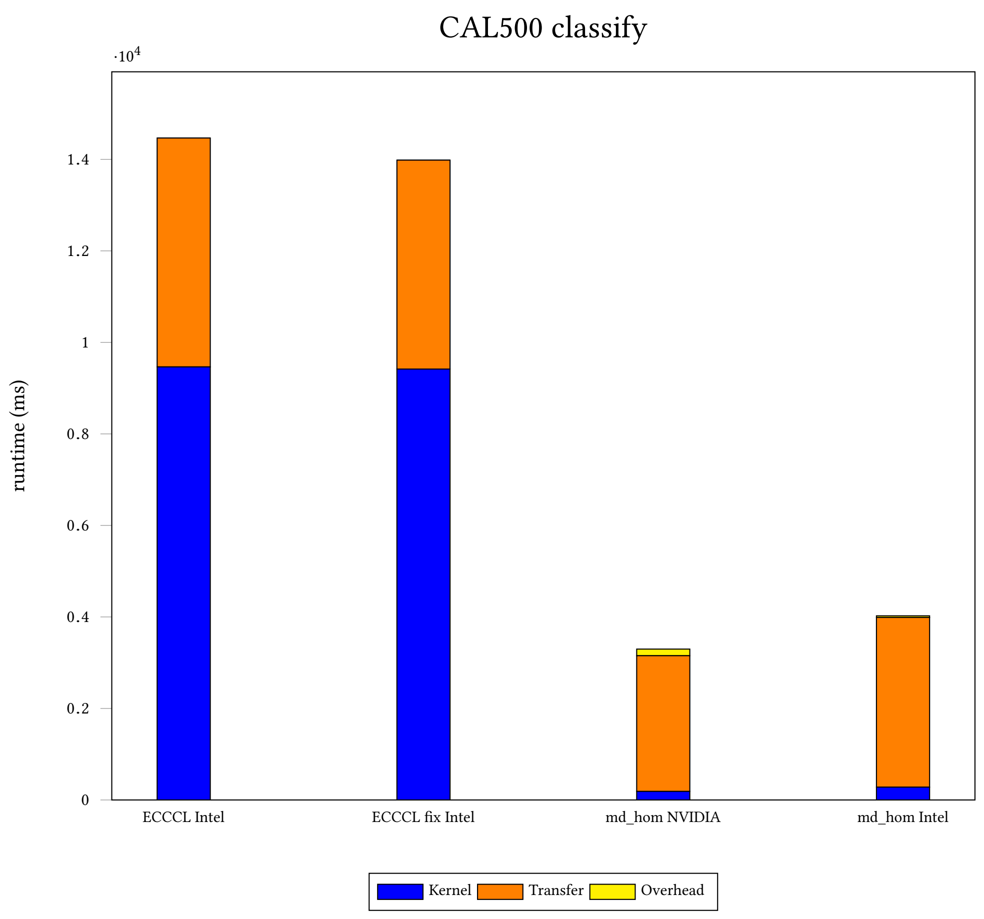
- **Instances**: 5000, 3350 for training, 1650 for evaluation
- **Attributes**: 499
- **Labels**: 374

Corel5k classify	Kernel	Transfer	Overhead
ECCCL Intel	14,324.600 ms	9,507.934 ms	0.000 ms
ECCCL fix Intel	15,096.300 ms	9,767.443 ms	0.000 ms
md_hom NVIDIA	2,027.877 ms	9,440.903 ms	457.139 ms



- **Instances**: 502, 336 for training, 166 for evaluation
- **Attributes**: 68
- **Labels**: 174

CAL500 build	Kernel	Transfer	Overhead
ECCCL Intel	9,081.170 ms	7,664.031 ms	0.000 ms
md_hom NVIDIA	11,515.500 ms	4,440.148 ms	22.652 ms
md_hom Intel	7,642.830 ms	5,396.570 ms	10.800 ms



- **Instances**: 502, 336 for training, 166 for evaluation
- **Attributes**: 68
- **Labels**: 174

CAL500 classify	Kernel	Transfer	Overhead
ECCCL Intel	9,464.390 ms	5,002.705 ms	0.000 ms
ECCCL fix Intel	9,416.660 ms	4,567.258 ms	0.000 ms
md_hom NVIDIA	187.047 ms	2,965.914 ms	144.385 ms
md_hom Intel	279.861 ms	3,708.591 ms	36.010 ms

B. Working with the md_hom implementation

Compilation using clang:

```
 clang++ *.cpp ARFFparser/*.cpp atf_library/src/*.cpp
-std=c++14
-I/usr/include/python2.7/
-I/usr/local/cuda-9.0/include/
-L/usr/bin/python2.7/
-L/usr/local/cuda-9.0/lib64
-lOpenCL
-lpython2.7
-lpthread
-o ecc
```

The *atf_library/libraries/opentuner/* subdirectory has to be in PYTHONPATH for running

Command line: *ecc [command] [options]*
possible commands:

- **tuneBuild** auto-tune the build MDA, produces a file with the best parameters found
- **tuneStep** auto-tune the step MDA, only works after first running tuneBuild with the same options, appends best configuration to the file created by tuneBuild
- **tuneFinal** auto-tune the final MDA, only works after first running tuneStep with the same options, appends best configuration to the file created by tuneStep
- **measure** measure performance and several prediction accuracy metrics, uses the output of tuneBuild, tuneStep and tuneFinal
- **measureold** measure performance and several prediction accuracy metrics for the fixed eccCL kernels
- **measureoldorig** measure performance and several prediction accuracy metrics for the original eccCL kernels

possible options:

- **-d** path to the dataset file in .arff format *required*

- **-l** number of labels in the dataset *required*
- **-depth** depth of each tree *default: 10*
- **-t** trees per forest *default: 32*
- **-c** chains in the ensemble *default: 64*
- **-ie** instance subset for training each chain *default: 100*
- **-if** instance subset for training each forest *default: 50*
- **-nl** maximum number of nodes to be created for each run of the build md_hom *default: all*
- **-platform** name of OpenCL platform *default: NVIDIA*
- **-device** name of OpenCL device *default: Tesla K20m*

Example run:

```
./ecc tuneBuild
-d data/bibtex.arff
-l 159
-c 64
-t 32
-nl 250000000
-platform Intel
```

Eigenständigkeitserklärung

Hiermit versichere ich, dass die vorliegende Arbeit

Evaluation of the md_hom Pattern Using the Example “Ensemble of Classifier Chains“

selbstständig verfasst worden ist, dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt worden sind und dass die Stellen der Arbeit, die anderen Werken – auch elektronischen Medien – dem Wortlaut oder Sinn nach entnommen wurden, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht worden sind.

(Ort, Datum) (Unterschrift)

Ich erkläre mich mit einem Abgleich der Arbeit mit anderen Texten zwecks Auffindung von Übereinstimmungen sowie mit einer zu diesem Zweck vorzunehmenden Speicherung der Arbeit in eine Datenbank einverstanden.

(Ort, Datum) (Unterschrift)