REALIZATION OF RANDOM FOREST FOR REAL-TIME EVALUATION THROUGH TREE FRAMING

Sebastian Buschjäger, Kuan-Hsun Chen, Jian-Jia Chen and Katharina Morik
TU Dortmund University - Artificial Intelligence Group and Design Automation for Embedded Systems Group
November 18, 2018
Motivation

**FACT** First G-APD Cherenkov Telescope continuously monitors the sky for gamma rays

**Goal** Have a small, cheap telescope which can be deployed everywhere on earth
Motivation

**FACT** First G-APD Cherenkov Telescope continuously monitors the sky for gamma rays

**Goal** Have a small, cheap telescope which can be deployed everywhere on earth

- It produces roughly 180 MB/s of data
- Only 1 in 10,000 measurements is interesting
- Bandwidth to transmit measurements is limited
**Motivation**

**FACT** First G-APD Cherenkov Telescope continuously monitors the sky for gamma rays

**Goal** Have a small, cheap telescope which can be deployed everywhere on earth

- It produces roughly 180 MB/s of data
- Only 1 in 10,000 measurements is interesting
- Bandwidth to transmit measurements is limited

**Idea** Use a Random Forest to filter measurements before further processing

- Pre-train forest on simulated data, then apply it in the real world
- Physicist know Random Forests
- Very good black-box learner, no hyperparameter tuning necessary
Motivation

**FACT** First G-APD Cherenkov Telescope continuously monitors the sky for gamma rays

**Goal** Have a small, cheap telescope which can be deployed everywhere on earth

- It produces roughly 180 MB/s of data
- Only 1 in 10,000 measurements is interesting
- Bandwidth to transmit measurements is limited

**Idea** Use a Random Forest to filter measurements before further processing

- Pre-train forest on simulated data, then apply it in the real world
- Physicist know Random Forests
- Very good black-box learner, no hyperparameter tuning necessary

**Goal** Execute Random Forest in real-time and keep-up with 180 MB/s of data

**Constraint** Size and energy available is limited → Model must run on embedded system
Recap Decision Trees and Random Forest

- DTs split the data in regions until each region is “pure”
- Splits are binary decisions if $x$ belongs to certain region
- Leaf nodes contain actual prediction for a given region
- RFs built multiple DTs on subsets of the data/features
Recap Decision Trees and Random Forest

- DTs split the data in regions until each region is “pure”
- Splits are binary decisions if $x$ belongs to certain region
- Leaf nodes contain actual prediction for a given region
- RFs built multiple DTs on subsets of the data/features

Question How to implement a Decision Tree / Random Forest?
Recap Computer architecture

- CPU computations are much faster than memory access
- Memory-Hierarchy (Caches) is used to hide slow memory
- Caches assume spatial-temporal locality of accesses

Question: How to implement a Decision Tree / Random Forest?
Implementing Decision Trees (1)

**Fact** There are at-least two ways to implement DTs in modern programming languages

**Native-Tree** Store nodes in array and iterate it in a loop

```c
Node t[] = {/* ... */};
bool predict(short const * x){
    unsigned int i = 0;
    while(!t[i].isLeaf) {
        if (x[t[i].f] <= t[i].s) {
            i = t[i].l;
        } else {
            i = t[i].r;
        }
    }
    return t[i].pred;
}
```
Implementing Decision Trees (1)

**Fact** There are at-least two ways to implement DTs in modern programming languages

**Native-Tree** Store nodes in array and iterate it in a loop

```c
Node t[] = { /* ... */ };
bool predict(short const * x){
    unsigned int i = 0;
    while(!t[i].isLeaf) {
        if (x[t[i].f] <= t[i].s) {
            i = t[i].l;
        } else {
            i = t[i].r;
        }
    }
    return t[i].pred;
}
```

- Simple to implement
- Small ‘hot’-code
- Requires D-Cache (array)
- Requires I-Cache (code)
- Requires indirect memory access
Implementing Decision Trees (2)

**Fact** There are at-least two ways to implement DTs in modern programming languages

**If-Else-Tree** Unroll tree into if-else instructions
Implementing Decision Trees (2)

**Fact** There are at-least two ways to implement DTs in modern programming languages

**If-Else-Tree** Unroll tree into `if-else` instructions

```c
bool predict(short const * x){
  if(x[0] <= 8191){
    if(x[1] <= 2048){
      return true;
    } else {
      return false;
    }
  } else {
    if(x[2] <= 512){
      return true;
    } else {
      return false;
    }
  }
}
```

- No indirect memory access
- Compiler can optimize aggressively
- Only I-Cache required
- I-Cache usually small
- No ‘hot’-code
Probabilistic execution model of DTs

**Basic idea** Analyse the structure of trained tree and keep most important paths in Cache

Branch-probability $p_{i \rightarrow j}$

Path-probability $p(\pi) = p_{\pi_0 \rightarrow \pi_1} \cdot \ldots \cdot p_{\pi_{L-1} \rightarrow \pi_L}$

Expected path length $\mathbb{E}[L] = \sum_{\pi} p(\pi) \cdot |\pi|$
Probabilistic execution model of DTs

**Basic idea** Analyse the structure of trained tree and keep most important paths in Cache

Example

\[ p((0, 1, 3)) = 0.3 \cdot 0.4 \cdot 0.25 = 0.03 \]
\[ p((0, 2, 6)) = 0.7 \cdot 0.8 \cdot 0.85 = 0.476 \]
Probabilistic optimizations for DTs

**Capacity misses** Cache memory is not enough to store all code

**But** Computation kernel of tree might fit into cache
Probabilistic optimizations for DTs

**Capacity misses** Cache memory is not enough to store all code

**But** Computation kernel of tree might fit into cache

**Solution** Compute computation kernel for budget $\beta$

$$\mathcal{K} = \arg \max \{ p(T) | T \subseteq \mathcal{T} \text{ s.t. } \sum_{i \in T} s(i) \leq \beta \}$$
Probabilistic optimizations for DTs

**Capacity misses** Cache memory is not enough to store all code

**But** Computation kernel of tree might fit into cache

**Solution** Compute computation kernel for budget $\beta$

$$\mathcal{K} = \arg \max \{ p(T) \mid T \subseteq \mathcal{T} \text{ s.t. } \sum_{i \in T} s(i) \leq \beta \}$$

- Start with the root node
- Greedily add nodes until budget is exceeded

**Note**

- Estimate $s(\cdot)$ based on assembly analysis
- Choose $\beta$ based on the properties of specific CPU model
Probabilistic optimizations for DTs (2)

Further optimizations

- Reduce memory consumption of nodes for native trees with clever implementation
- Increase cache-hit rate for if-else trees by swapping nodes with higher probability
Probabilistic optimizations for DTs (2)

Further optimizations

- Reduce memory consumption of nodes for native trees with clever implementation
- Increase cache-hit rate for if-else trees by swapping nodes with higher probability

In total Compare 1 baseline method and 4 different implementations
Probabilistic optimizations for DTs (2)

Further optimizations

- Reduce memory consumption of nodes for native trees with clever implementation
- Increase cache-hit rate for if-else trees by swapping nodes with higher probability

In total Compare 1 baseline method and 4 different implementations

Questions

- What is the performance-gain of these optimizations?
- How do these optimizations perform on different CPU architectures?
- How do these optimizations perform with different forest configurations?
Experimental Setup

Approach

- Use a Code-Generator to compile `sklearn` forests (DTs, RF, ET) of varying size to C-Code
- Test resulting code + optimizations on 12 datatest on 3 different CPU architectures
Experimental Setup

Approach

- Use a Code-Generator to compile `sklearn` forests (DTs, RF, ET) of varying size to C-Code
- Test resulting code + optimizations on 12 datatest on 3 different CPU architectures

Hardware

- **X86** Desktop PC with Intel i7-6700 with 16 GB RAM
- **ARM** Raspberry-Pi 2 with ARMv7 and 1GB RAM
- **PPC** NXP Reference Design Board with T4240 processors and 6GB RAM
## Experimental Setup (2)

<table>
<thead>
<tr>
<th>Dataset</th>
<th># Examples</th>
<th># Features</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>adult</td>
<td>8141</td>
<td>64</td>
<td>0.76 - 0.86</td>
</tr>
<tr>
<td>bank</td>
<td>10297</td>
<td>59</td>
<td>0.86 - 0.90</td>
</tr>
<tr>
<td>covertype</td>
<td>145253</td>
<td>54</td>
<td>0.51 - 0.88</td>
</tr>
<tr>
<td>fact</td>
<td>369450</td>
<td>16</td>
<td>0.81 - 0.87</td>
</tr>
<tr>
<td>imdb</td>
<td>25000</td>
<td>10000</td>
<td>0.54 - 0.80</td>
</tr>
<tr>
<td>letter</td>
<td>5000</td>
<td>16</td>
<td>0.06 - 0.95</td>
</tr>
<tr>
<td>magic</td>
<td>4755</td>
<td>10</td>
<td>0.64 - 0.87</td>
</tr>
<tr>
<td>mnist</td>
<td>10000</td>
<td>784</td>
<td>0.17 - 0.96</td>
</tr>
<tr>
<td>satlog</td>
<td>2000</td>
<td>36</td>
<td>0.40 - 0.90</td>
</tr>
<tr>
<td>sensorless</td>
<td>14628</td>
<td>48</td>
<td>0.10 - 0.99</td>
</tr>
<tr>
<td>wearable</td>
<td>41409</td>
<td>17</td>
<td>0.57 - 0.99</td>
</tr>
<tr>
<td>wine-quality</td>
<td>1625</td>
<td>11</td>
<td>0.49 - 0.68</td>
</tr>
</tbody>
</table>
Results: Desktop PC with Intel (X86)

**Note**  Behaviour similar for DTs, RF and ET → Focus in RF here

**Results**
- Optimizations improve performance
- if-else trees are clear winner

**Interpretation**
- Large I-Cache (256 KiB) favors if-else
- Compiler can utilize CISC architecture for if-else
- Native trees do not benefit from I-Cache and CISC
Results: Desktop PC with Intel (X86)

**Note** Behaviour similar for DTs, RF and ET → Focus in RF here

**Results**
- Optimizations improve performance
- if-else trees are clear winner

**Interpretation**
- Large I-Cache (256 KiB) favors if-else
- Compiler can utilize CISC architecture for if-else
- Native trees do not benefit from I-Cache and CISC

**Take-away** On X86 CPUs, if-else trees should be favoured
**Results: Raspberry Pi with ARMv7 (ARM)**

**Note** Behaviour similar for DTs, RF and ET → Focus in RF here

**Results**
- Optimizations improve performance
- No clear winner for larger trees

**Interpretation**
- Smaller I-Cache (32 KiB) only fits small trees
- Smaller D-Cache (512 KiB) only fits small trees
- Requires more instructions than CISC
Results: Raspberry Pi with ARMv7 (ARM)

Note Behaviour similar for DTs, RF and ET → Focus in RF here

Results
- Optimizations improve performance
- No clear winner for larger trees

Interpretation
- Smaller I-Cache (32 KiB) only fits small trees
- Smaller D-Cache (512 KiB) only fits small trees
- Requires more instructions than CISC

Take-away Use if-else version for small trees. For larger ones there is no clear recommendation
Summary and Take-Aways

**Modern physics** experiments generate huge amounts of data

**But** We can use ML to filter-out unwanted measurements before further processing
Summary and Take-Aways

Modern physics experiments generate huge amounts of data

But We can use ML to filter-out unwanted measurements before further processing

Our approach Use a code-generator to generate optimized RF code

▶ There are at-least two ways to implement Decision Trees in modern languages
▶ Native trees mostly rely on the data cache
▶ If-else trees mostly rely on the instruction cache
▶ Careful cache management can increase performance by 2 – 6 (1500 compared to sklearn)
▶ Optimizations & implementations behave differently on different CPU architectures
**Summary and Take-Aways**

**Modern physics** experiments generate huge amounts of data

**But** We can use ML to filter-out unwanted measurements before further processing

**Our approach** Use a code-generator to generate optimized RF code

- There are at-least two ways to implement Decision Trees in modern languages
- Native trees mostly rely on the data cache
- If-else trees mostly rely on the instruction cache
- Careful cache management can increase performance by 2 – 6 (1500 compared to sklearn)
- Optimizations & implementations behave differently on different CPU architectures

**Now** Physicist can generate optimized C code for each new experiment

**And you as well!**

https://bitbucket.org/sbuschjaeger/arch-forest