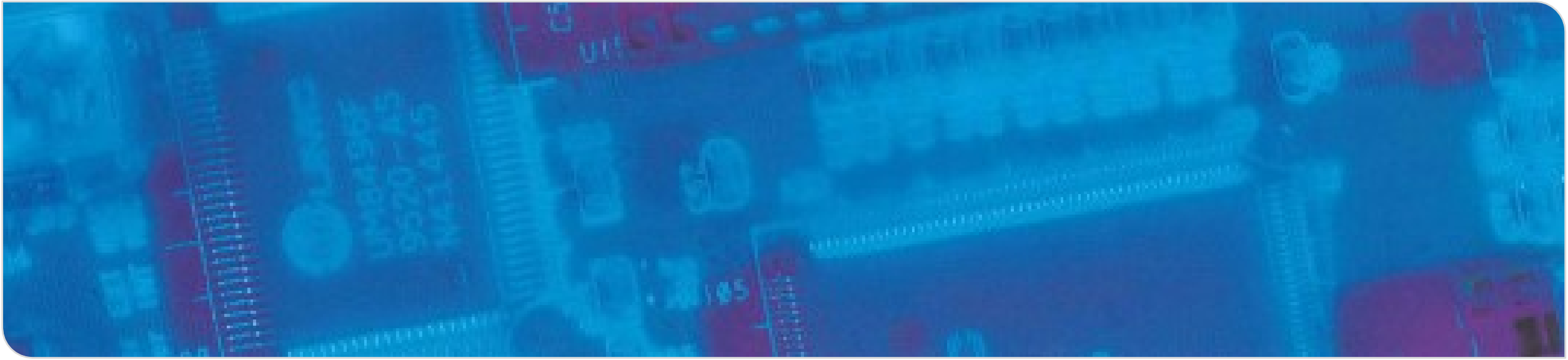
Karlsruhe Institute of Technology

Institut für Technik der Informationsverarbeitung

Heads of the Institute
Prof. Dr.-Ing. Dr.h.c. J. Becker (Speaker)
Prof. Dr.-Ing. E. Sax
Prof. Dr. rer. nat. W. Stork

Embedded Multi-Core Code Generation with Cross-Layer Parallelization

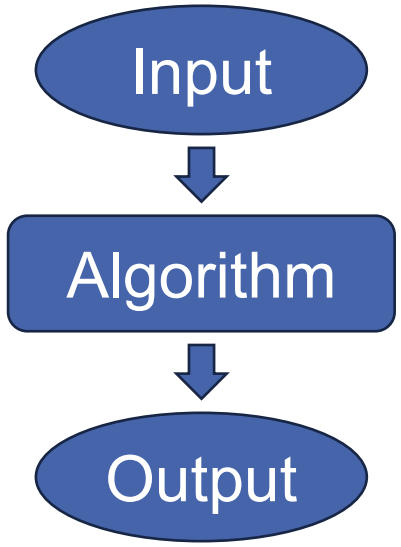
Oliver Oey

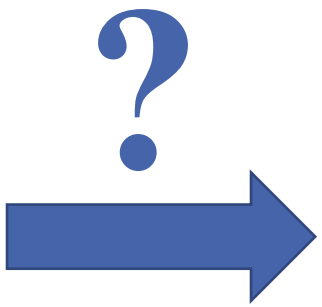


Overview

- Introduction
- Concept of Cross-Layer Parallelization
  - Algorithm layer
  - Code layer
  - Task layer
  - Data layer
- Evaluation with example
- Summary and Outlook

Introduction





Algorithm:

- Mathematical procedure
- Process input data to generate deterministic results

Embedded System:

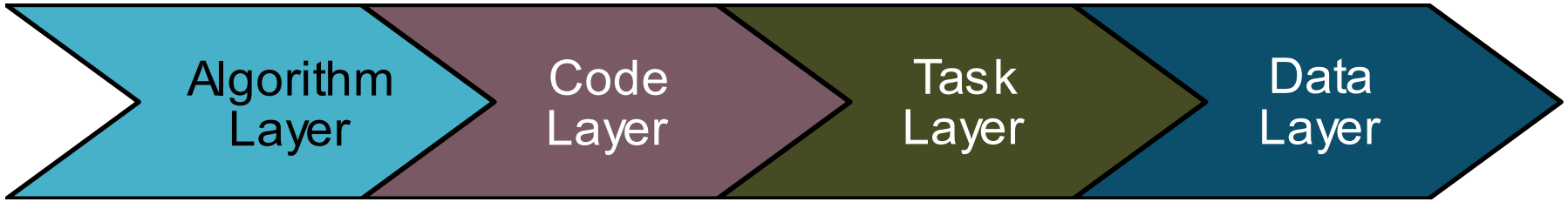
- Computing system with limited resources
- Heterogenous processing units
- Timing constraints

How can an algorithm be transferred to the embedded system as efficiently as possible?

Requirements

- Algorithm development independent of target platform
- Abstract representation on a mathematical level
- Efficient utilization of execution units: CPU + Accelerators
- Automate time-consuming and error-prone decisions

Layered Approach



From abstract representation to hardware

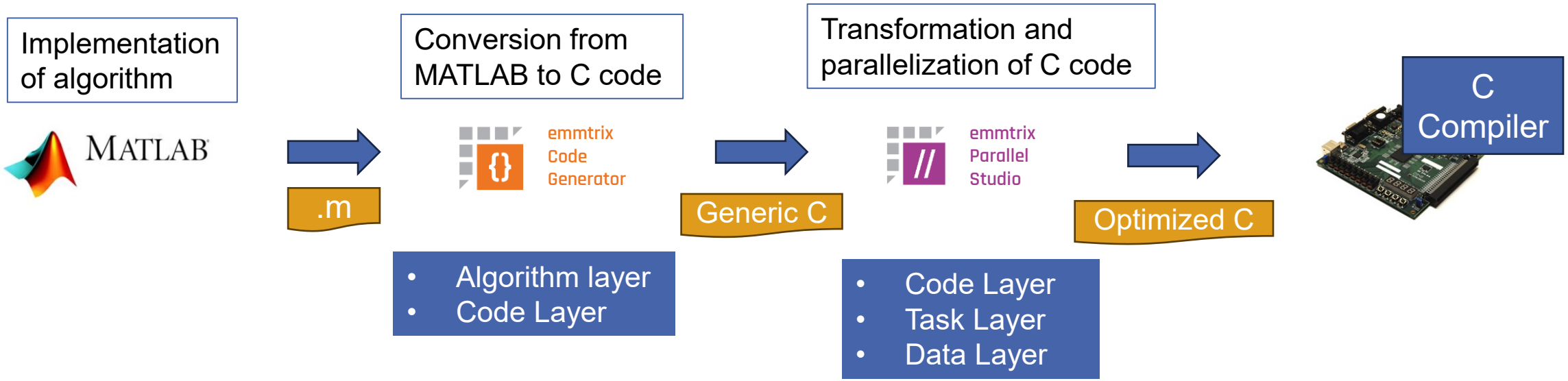
- Algorithm Layer
  - Optimization or implementation for known algorithms/functions
  - Typically implemented as libraries
- Code Layer
  - Representation of algorithm in source code
  - Transformations and special code optimized for target platform
- Task Layer
  - Coarse grain parallelization
  - Assignment of tasks to processing elements (PE)
- Data Layer
  - Data exchange between PEs
  - Efficient use of shared memory or interconnects

Concept of Cross-Layer Parallelization

- Layers allow different kinds of optimizations
  - Abstract: general execution of the algorithm
  - Hardware-related: specific adaptations to the target platform
- Optimizations on one layer affect the optimization potential of following layers
- Optimization potential per layer also depends on characteristics of algorithm
  - Data flow
  - Control flow
  - Issue size

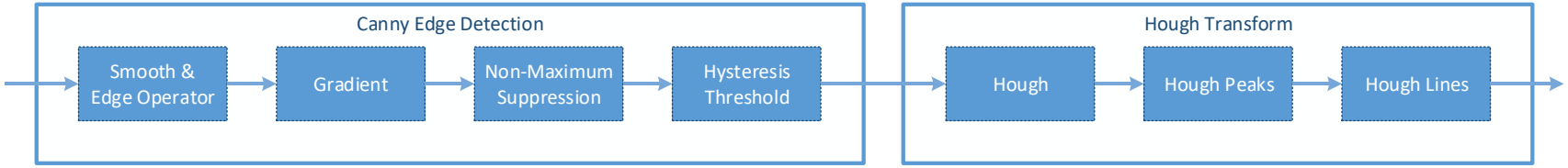
→ To achieve the best performance, parallelization needs to take into account all layers and the characteristics of the input algorithm

Workflow



- Reference implementation of flow: manual steps assisted by existing tools
- MATLAB: realization of algorithm without hardware-awareness. No data types required. Simple development with runtime interpreter
- C: close to hardware with low-level optimization potential

Example: Streak Detection Algorithm

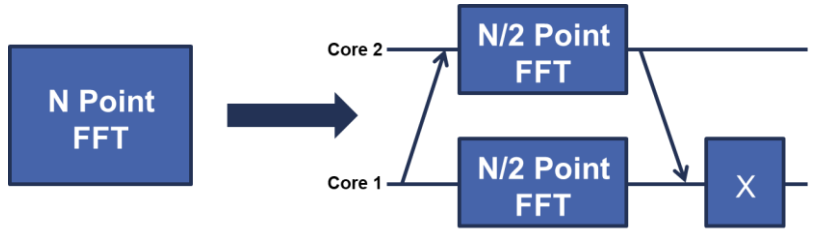



```
function [point1Arrays,point2Arrays] = streak_detection(img)
    %convert to grayscale, available in MATLAB
    gray = rgb2gray(img);
    %extract edges from image, available in MATLAB
    [E,thresh] = edge(double(gray),'Canny');
    %apply Hough transform, available in MATLAB
    [H,T,R] = hough(E);
    %extract peaks in Hough representation, available in MATLAB
    P = houghpeaks(H, 50);
    %draw a line to connect points, implemented manually
    [point1Arrays,point2Arrays] = custom_houghlines(E,T,R,P);
end
```

- Implemented in MATLAB, mostly with inbuilt functions
- No hardware-related implementations
- Image processing – dominated by processing of large data arrays

Algorithm Layer

- Goal: detect algorithms and provide different realizations. Examples:
  - A 1024 point FFT can be replace by 2 with 512 points → good for 2 cores
  - Sorting algorithm: same results, different memory requirements, parallel parts



- MATLAB as input:
  - clear specification of inbuilt functions
  - No complex detection required → usage clear
  - Different implementations can be used during MATLAB to C conversion
- Streak detection example:
  - rgb2gray and hough: processing of big data arrays.
    - Versions optimized for specified numbers of cores
    - Different accuracy of used data types
  - Edge: Canny works on X and Y direction, can be processed independently

Code Layer

- Goal: prepare source code for different processing elements. Same functionality, different representation in code. Examples:
  - Loops: data-driven algorithms typically use many loops. Code transformations (tiling, fission, unrolling) can be used to prepare for parallel execution or usage of accelerators
  - Data accesses: insert streaming buffer or move to shared/local memory
- Streak detection example:
  - Convolution in edge: each direction can further be split into independent parts

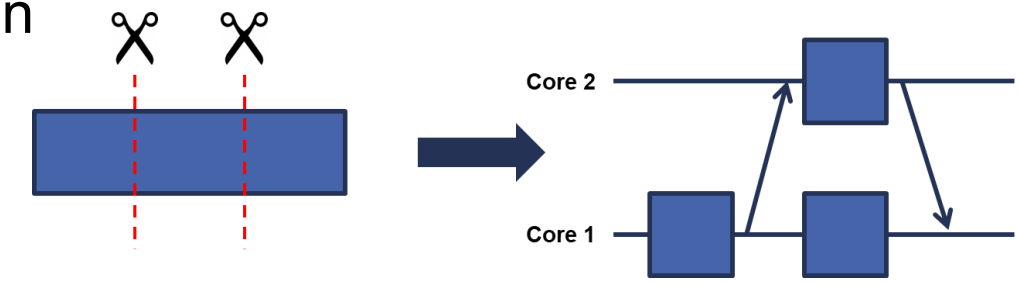
```
void conv(double B_0_data[1018][371],
          double B_1_data[1018][372],
                 double k_data[13]) {

  for (i11 = 1; i11 < 372; i11 = i11 + 1) {
    for (i12 = 1; i12 < 1019; i12 = i12 + 1) {
      sum2_data = 0.0;
      for (i13 = 1; i13 < 14; i13 = i13 + 1) {
        sum2_data = sum2_data + chain2_data;
      }
      B_0_data[i12 - 1][i11 - 1] = sum2_data;
    }
  }

  for (i6 = 372; i6 < 744; i6 = i6 + 1) {
    for (i7 = 1; i7 < 1019; i7 = i7 + 1) {
      sum_data = 0.0;
      for (i8 = 1; i8 < 14; i8 = i8 + 1) {
        sum_data = sum_data + chain1_data;
      }
      B_1_data[i7 - 1][i6 - 372] = sum_data;
    }
  }
}
```

Task Layer

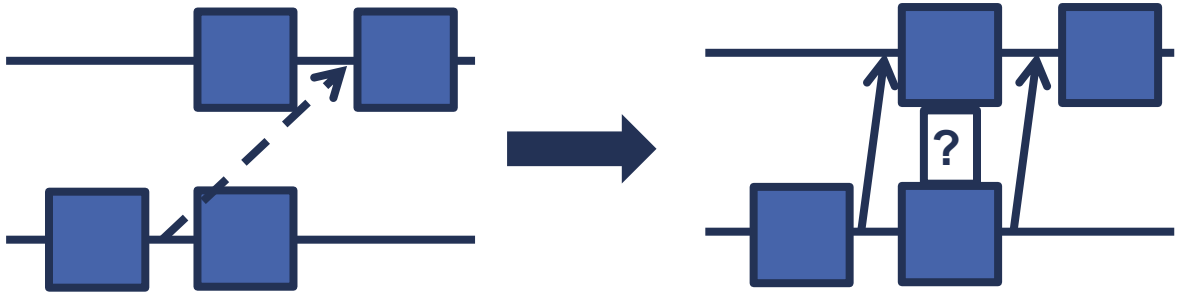
- Goal: Mapping and scheduling of tasks to processing elements (PE)
  - Performance model required to get runtime of task on PE
  - Overhead cost for data transfer/synchronization
  - Detect independent parts in the code
- Modelled as optimization algorithm:
  minimize runtime with parallel
  execution and low communication overhead



- Here:
  - Heterogeneous Earliest Finish Time (HEFT) algorithm
  - Tasks are assigned a rank depending on (data) dependencies and execution time
  - Heuristic: Tasks with low rank are scheduled first on available core

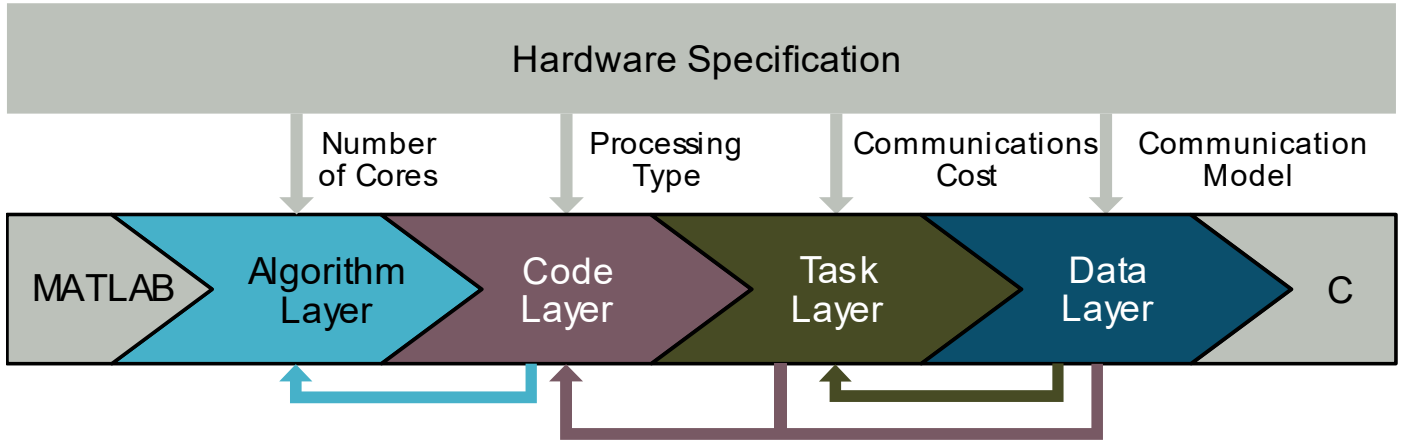
Data Layer

- Goal: Data availability on the individual cores
  - Insertion of communication and synchronization to ensure correct execution
  - Minimize runtime
  - Avoid deadlocks and data races
  - Determine synchronization points
  - Memory allocation on the cores



- Hardware dictates optimization potential
  - Shared memory: low overhead, placement can reduce waiting times
  - DMA: data transfer parallel to calculation
  - Availability of special instructions, e.g. circular addressing mode allows efficient use of ring buffers

Interactions Between Layers



- Iterative approach: later layers set options from previous layers. Layers are processed from left to right with refinement steps back
  - AL: CL requests implementation for certain number of cores
  - CL: TL and DL provide feedback on actual number of independent tasks required so that communication is not the bottleneck
  - TL: Feedback from DL is used to determine number of parallel tasks

Results for Streak Detection I

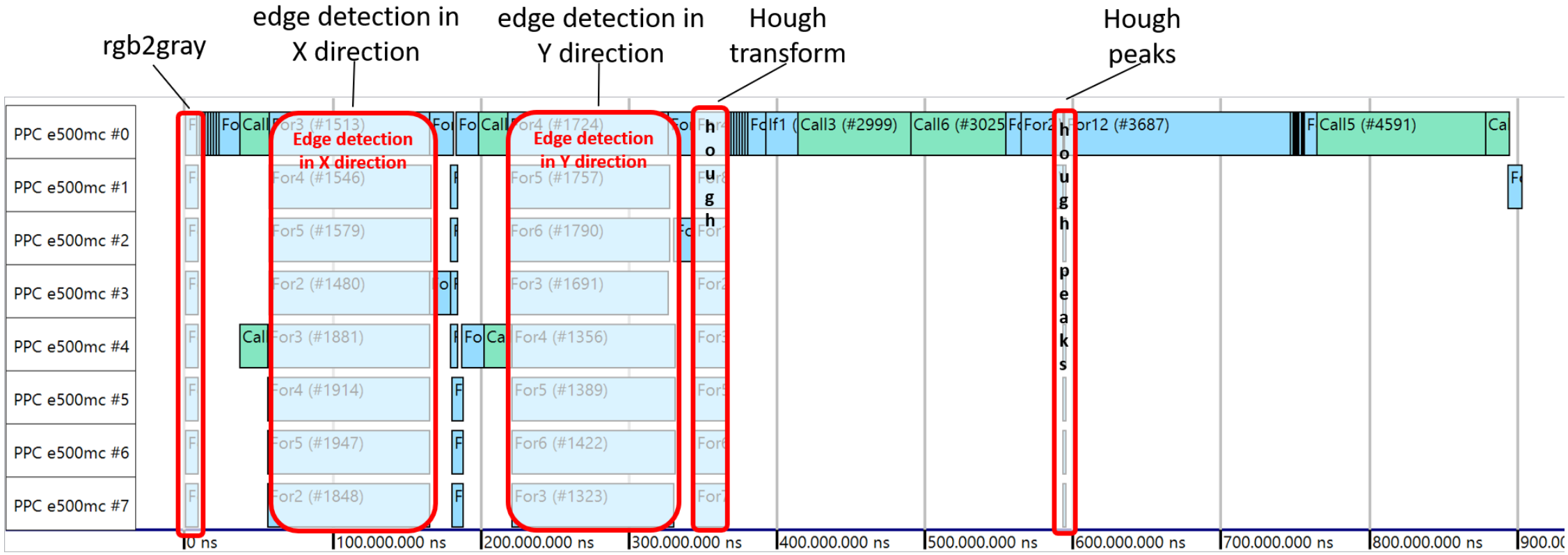
- Hardware platform: NXP P4080 DS with 8 PowerPC e500mc cores
- Algorithm layer:
  - Standard procedure for edge detection: Use of optimized Canny algorithm
    - Optimized thresholds: -6%
    - Simple accuracy: -28%
    - Optimized memory initialization: -19%
    - → In total: ~61% reduction in runtime
- Code layer:
  - Canny can be optimized for threads via loop transformations
    - X and Y direction can be executed in parallel
    - Each direction can be split into 8 cores
  - Canny takes 2/3 of the overall runtime, this change: +5% for sequential execution

Results for Streak Detection II

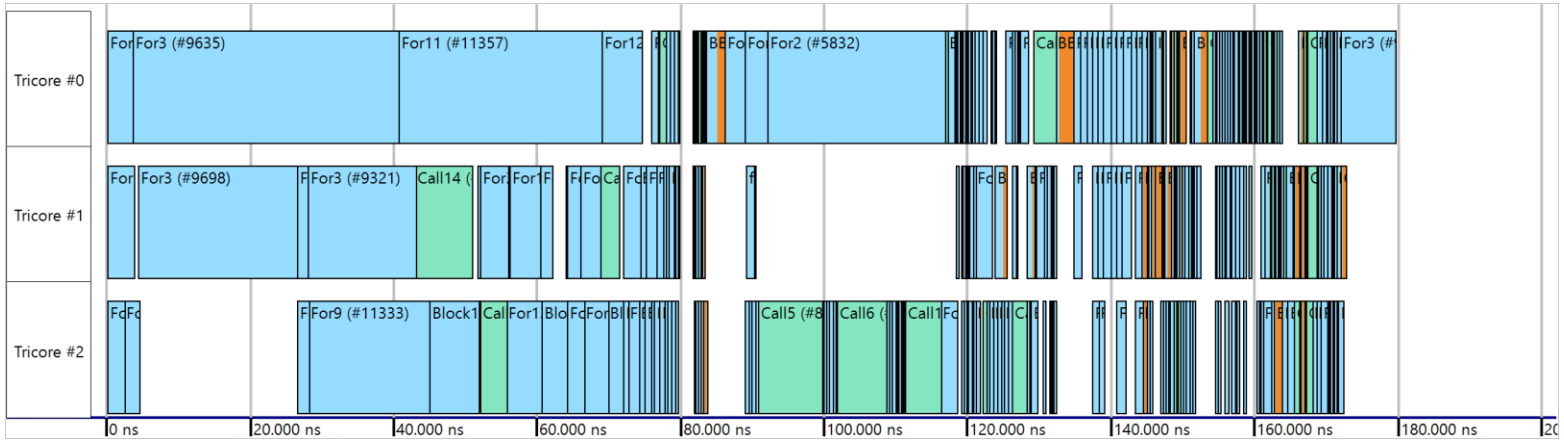
- Task layer:
  - Canny for 8 cores: Speedup of 6.28 achievable
  - Smaller functions such as RGB2Gray, hough and hough_peak can be accelerated with 8 cores, but speedup < 2
- Data layer:
  - Communication overhead relatively large
  - Optimization leads to ~ -19% runtime

- Overall: Preparations on AL and CL were vital to achieve speedup on TL
  - Final speedup: 2.86 for complete algorithm compared to sequential execution

Final Schedule of Streak Detection Algorithm



Example for Control Flow Driven Algorithm



- Characteristics
  - Many short tasks (~1400)
  - Lots of conditions that lead to decisions
  - Only few loops for data processing
  - Time to synchronize data is in the range of the runtime of a single task
- Key insight: AL not that important, CL important to cluster short tasks (→ reduce communication), then the assignment on TL can be handled efficiently
- Achieved speedup: 1.4 on an Infineon AURIX TC297 with 3 cores

Summary and Outlook

- Cross-layer parallelization across 4 abstraction layers
  - Later layers are used to iteratively refine decisions from previous layers
  - Based on the characteristics of the input algorithms, certain layers are more important than others to achieve best efficiency
- Reference implementation shows promising results for data-driven and control-flow-driven algorithms

- More automation potential: automated decision making and interactions between layers
- Methodology is also applicable to FPGA, GPU and other accelerators flow
- Optimizations can also be used to reduce power consumption

Any Questions?

Contact: oliver.oey@emmtrix.com