A CPU-FPGA Holistic Source-To-Source Compilation Approach for Partitioning and Optimizing C/C++ Applications

32nd International Conference on Parallel Architectures and Compilation Techniques (PACT 2023)

October 21-25, Vienna, Austria

Tiago Santos, João Bispo, João M. P. Cardoso

Faculty of Engineering of the University of Porto and INESC-TEC, Porto, Portugal

{tiagolascasas, jbispo, jmpc}@fe.up.pt

Context & Motivation

1. HW/SW Partitioning

How do we determine the regions for offloading? From offloading hotspots to offloading regions, augmenting the potential for optimizations that increase the overall performance!

```
void foo(int A[100], int B[100]){
    for (int i = 0; i < 100; i++)
        for (int j = 0; j < i; j++)
            A[i] = A[i] + B[j];
}
void bar(int A[100], int B[100]) {
    for (int i = 0; i < 100; i++)
        A[i] = A[i] + B[i];
}
```

To HW

2. Code Optimizations for HLS

How to select regions with the overall view of impactful code transformations and optimizations?

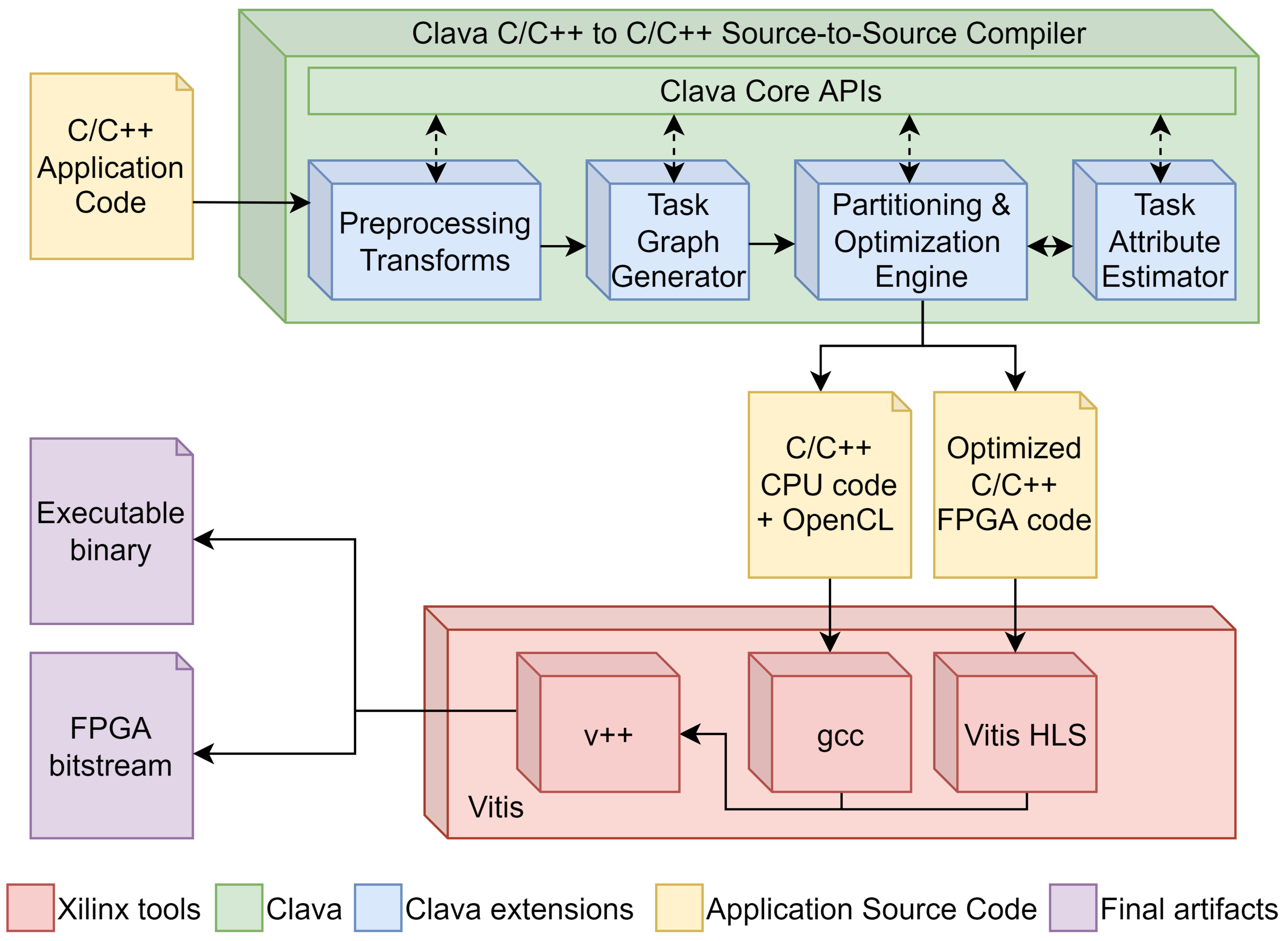
```
void bar(int A[100], int B[100]) {
#pragma HLS array_partition variable=A complete
    for (int i = 0; i < 100; i++) {
#pragma HLS unroll factor=20
#pragma HLS pipeline
        A[i] = A[i] + B[i];
    }
}
```

There is already extensive research about each process, when looked at in isolation. But what if they could be a single process, enriched by a holistic view of the application? We posit the question:

Given a heterogeneous CPU-FPGA system, does a combined partitioning and optimization scheme for an application achieve higher speedups than those achieved by applying both processes independently?

Can the whole be more than the sum of its parts?

Our Toolchain



From C/C++ to a Task Graph

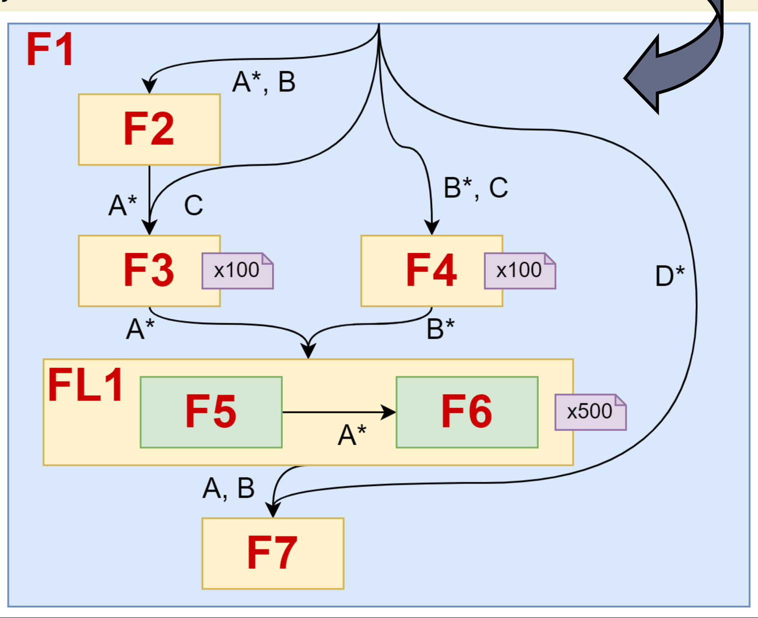
1. Preprocessing Transformations

- Constant folding and propagation
- Converting N-dimensional arrays into 1D
- Ensuring all functions return void
- Ensuring all branching evaluations are performed over variables, and not expressions
- Outlining of every computation into individual functions, so that functions either only have computations, or only have calls to other functions

The task graph has a 1:1 mapping between a task and a function, which simplifies code generation!

2. Task Graph Generation

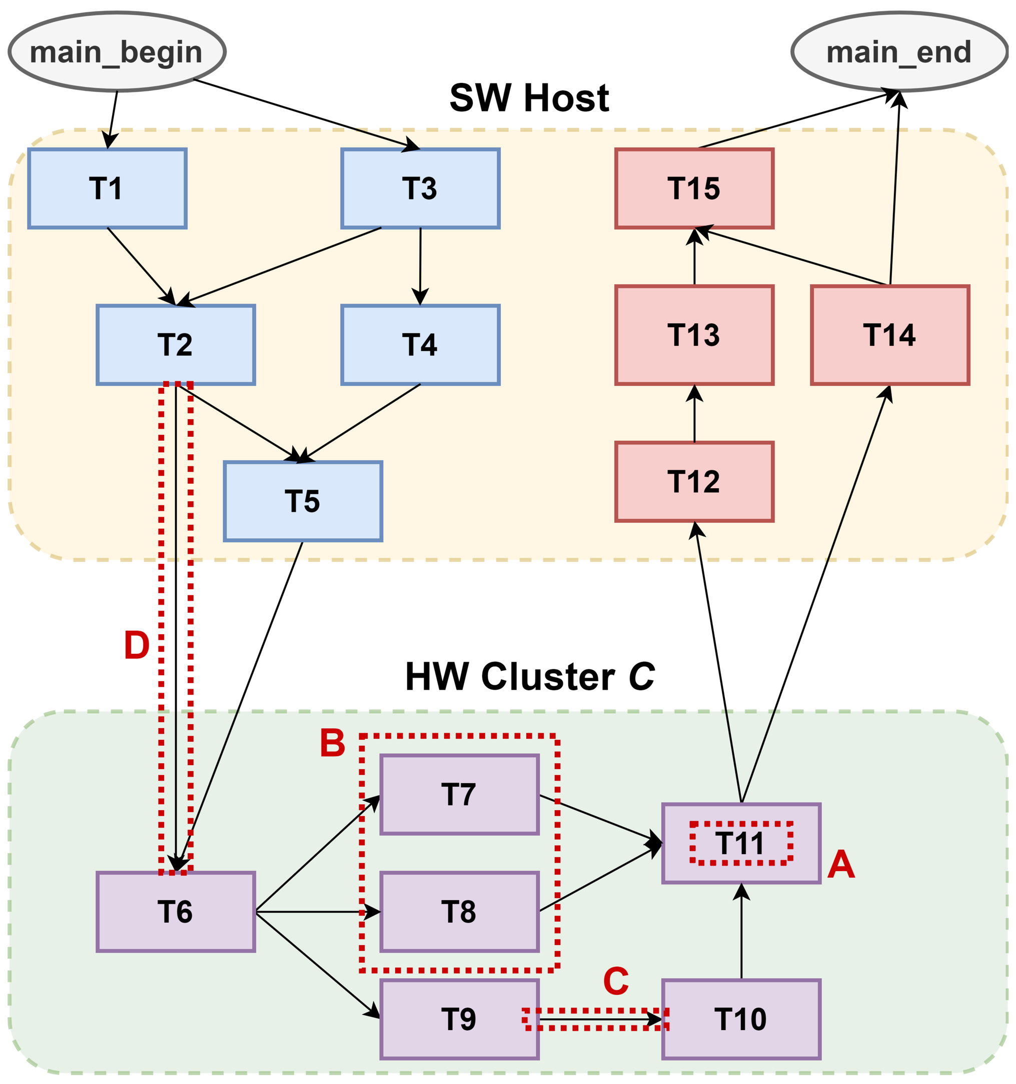
```
void F1(int *A, int *B, int *C, int *D) {
    F2(A, B);
    for (int i = 0; i < 100; i++) {
        F3(A, C[i]);
        F4(B, C[i]);
    }
    for (int i = 0; i < 500; i++) {
        F5(A, B);
        F6(A);
    }
    F7(A, B, D);
}
```



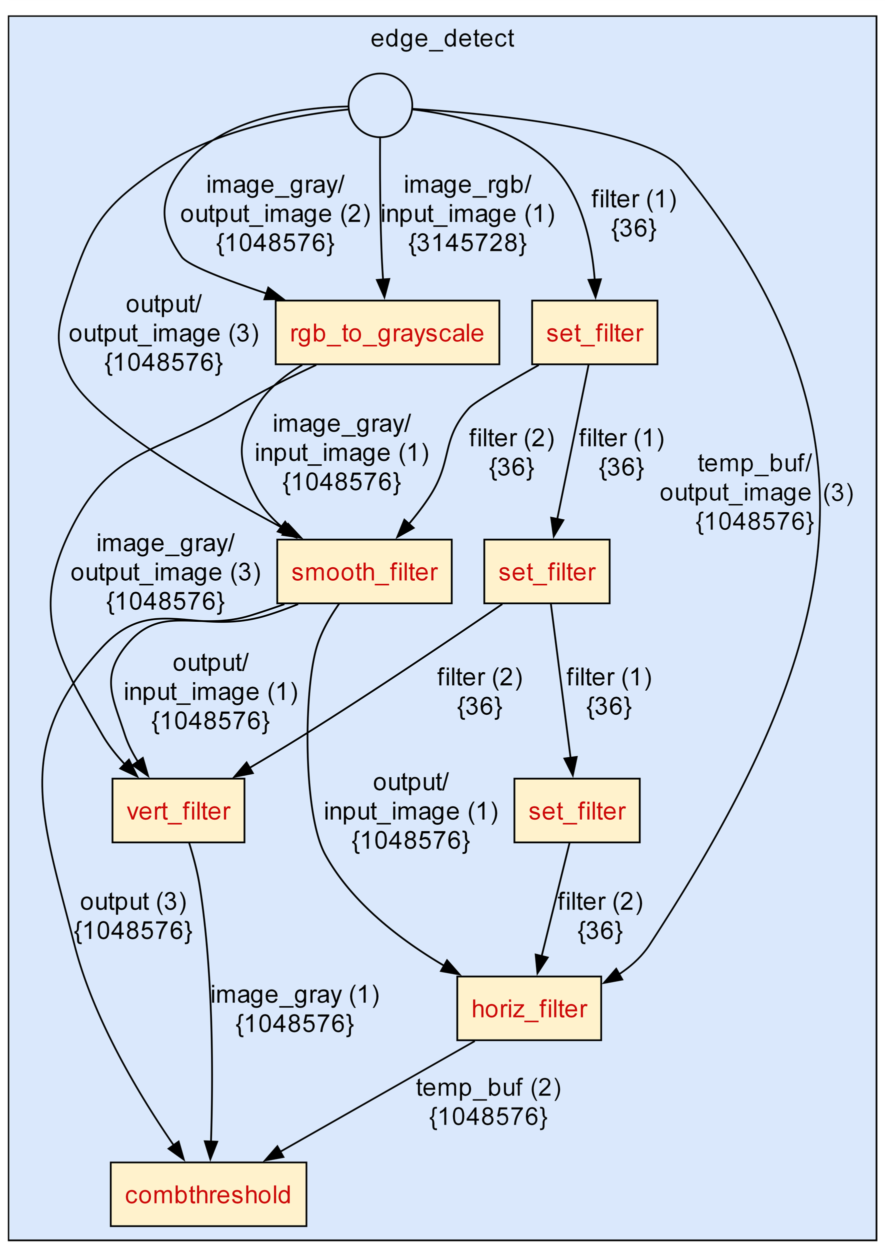
Holistic Partitioning and Optimization

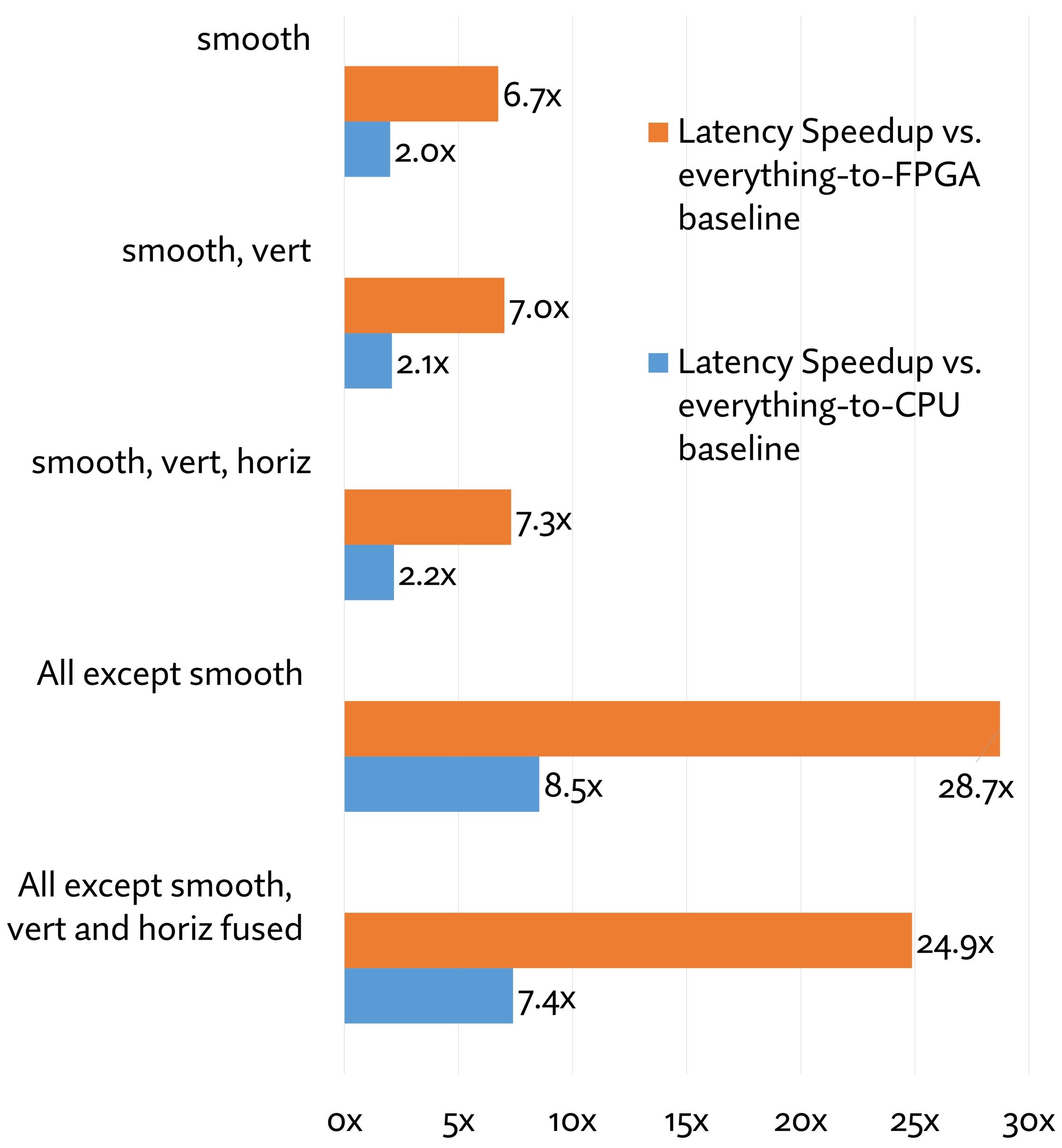
- Initial clusters based on hotspot tasks, measured through CPU profiling
- Each cluster is then increasingly expanded with promising tasks from outside the cluster, **in a single pass**
- **Each cluster then offers several design decisions, with optimizations enabled at multiple levels:**

A: intra-task optim. (e.g., loop unrolling)
B: inter-task optim. (e.g., task fusion)
C: intra-cluster optim. (e.g., dataflow patterns)
D: inter-cluster optim. (e.g., FIFO communication from the CPU to the FPGA)



Motivating Example





Target Evaluation Platform:

Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit

Current Progress

Code Preprocessing Transformations	**Done**
Test Environment Setup	**Done**
Task Graph Generation	**Done**
Characterizing Task Graphs	**WIP**
Generating Hotspot-based Clusters	**WIP**
Generating Different Optim. Designs	**TBD**
Design Selection and Refinement	**TBD**

Tiago Santos is supported by PhD grant 2021.07324.BD, financed by Fundação para a Ciência e Tecnologia (FCT)