A Task Graph Representation for Flexible Hardware/Software Partitioning

HiPEAC 2024 conference

January 17-19, Munich, Germany

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

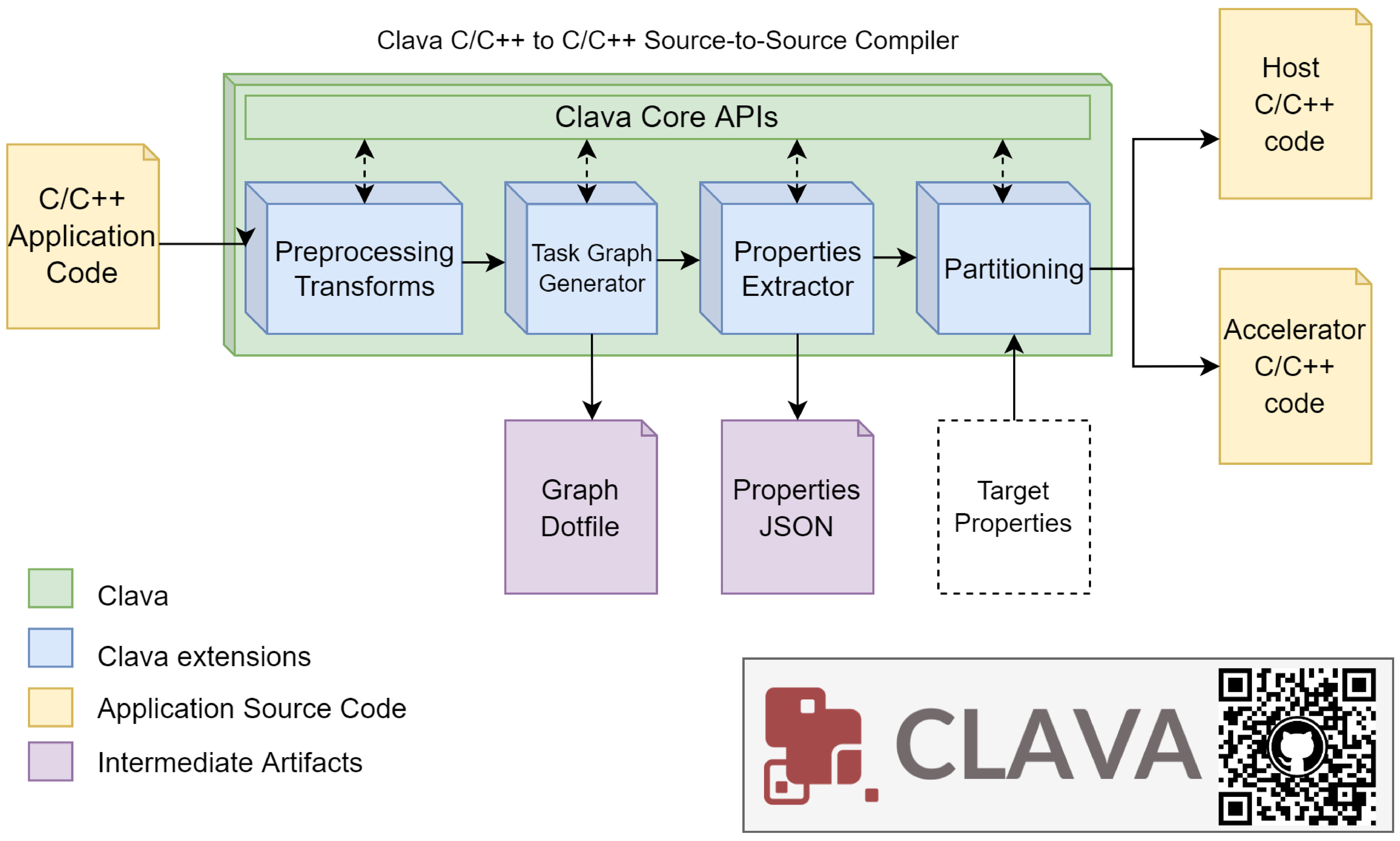
How to select regions with the overall view of impactful code transformations and optimizations?

```
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];
    }
}
```

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?

How can we represent an application as a task graph that enables this holistic approach?

Tool Flow



From C/C++ to a Task Graph

Granularity based on traditional code regions (e.g., expressions, loops, functions) is not enough, as we need to create clusters of tasks representing complex code regions with multiple functions. Therefore, we map every task in the task graph to a function in the AST, with flexible granularity achieved by merging and splitting tasks. These graph operations map to function inlining/outlining transformations on the AST, thus preserving the source code.

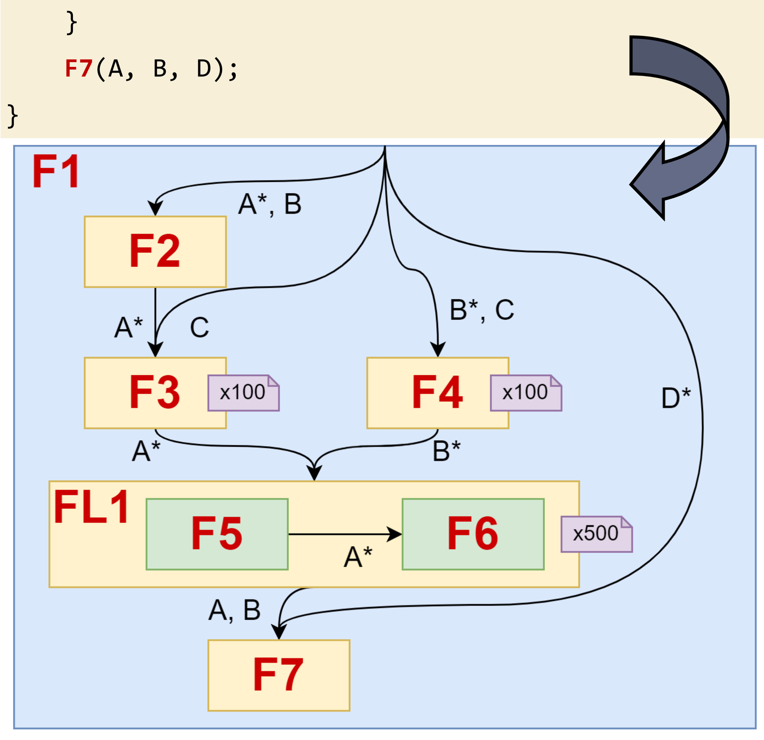
1. Preprocessing Transformations

- 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!

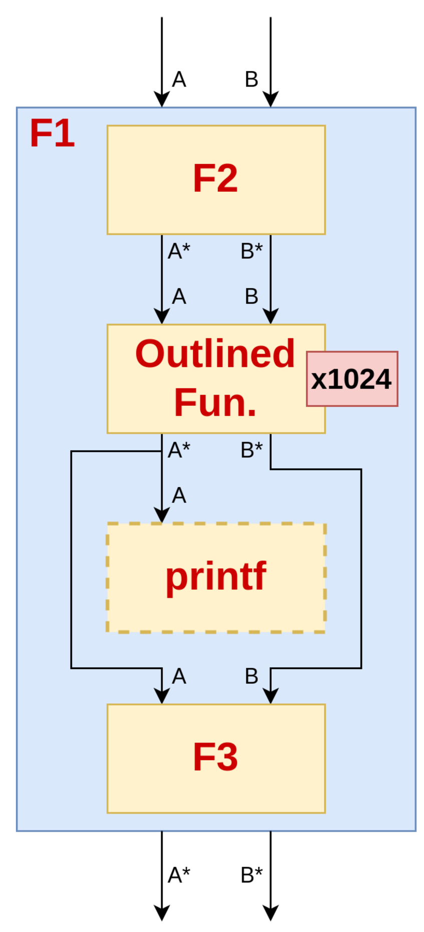
2. Task Graph Generation

```
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);
}
```



Task Graph Properties

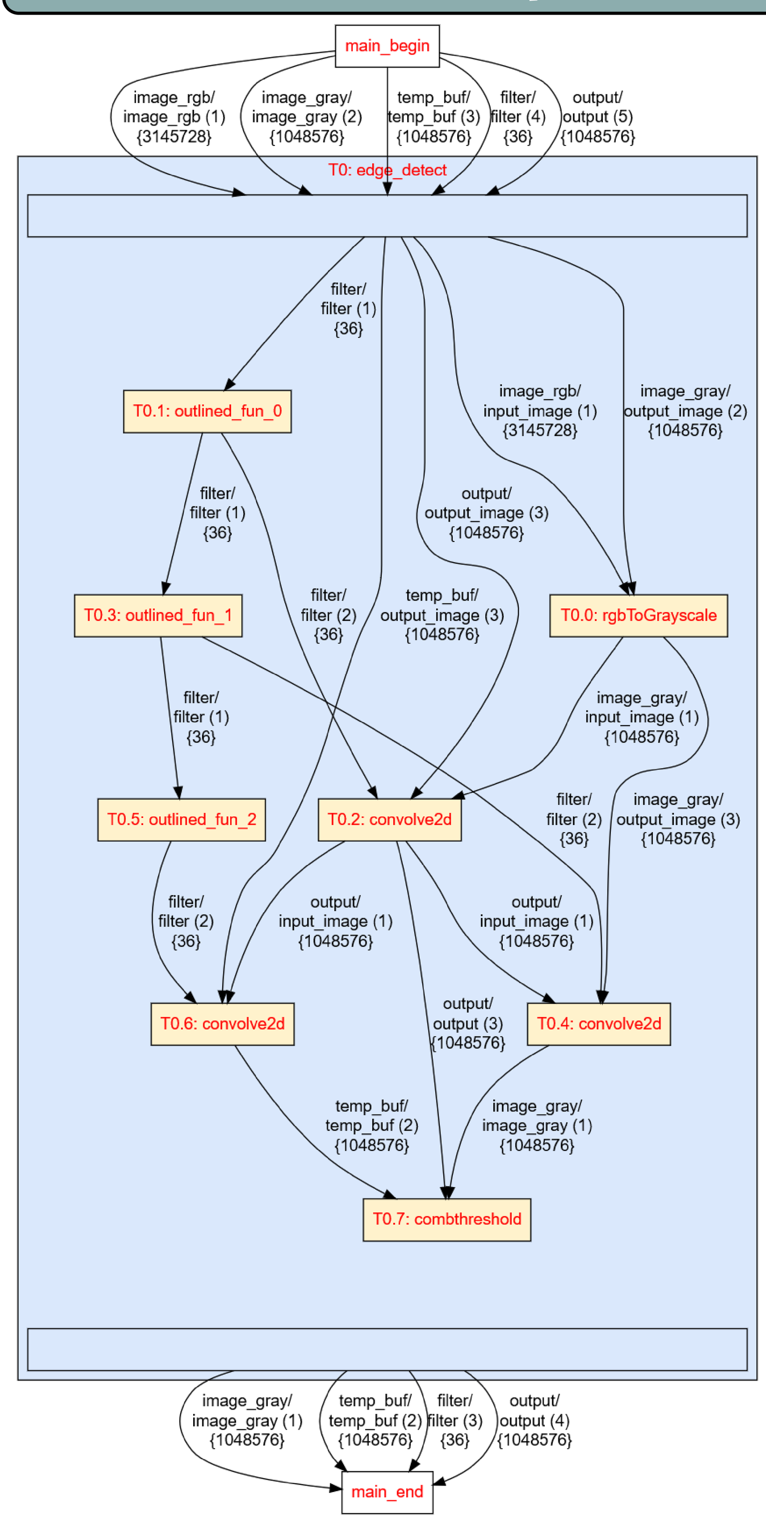
1. No-task Call Spots



Sometimes, we do not have an implementation for a function in a call spot:

- Functions without system calls, e.g., sqrt(), are considered operators and do not lead to the creation of a task, i.e., they are outlined into a function alongside the rest of the expression they are part of;
- Functions with system calls, e.g., printf(), lead to the creation of a task with no implementation, to be handled according to the target's handling of system calls, or lack thereof.

2. Data Properties and Paths



The basic units for communication are data items, with one communication edge existing per data item communicated (i.e., our task graph is a multigraph).

Each edge is annotated with the names of the data item, which may differ between caller and callee, and its size, as long as it is possible to determine at compile-time.

By keeping track of the different alias a data item may take, we can determine, from any task, the origin and eventual endpoint of the data item, which allows us to create clusters of tasks that use the same data items. This can be further refined by finding tasks that only use a copy of the data item (i.e., any modification is not persistent), and those that completely overwrite the data.

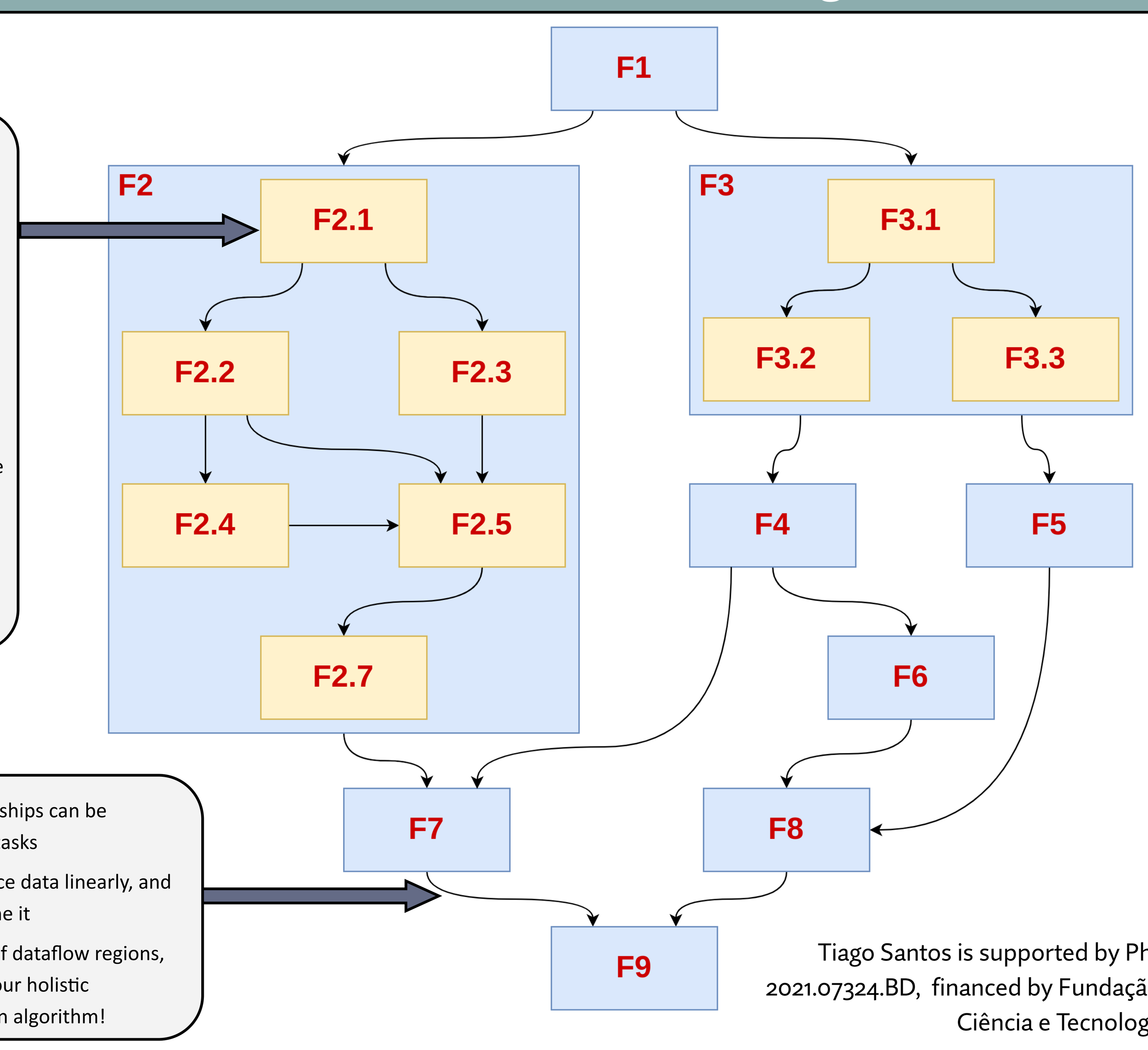
3. Task-level Parallelism and Dataflow Regions

Critical Path Length = 5

#Tasks in Subgraph = 6

Relative level of parallelism = 6 / 5 = 1.2

Subgraphs with high level of parallelism may indicate regions suitable for execution in both the host and accelerator, and parallelism may be exposed through task splitting operations in regions with low values.



- Producer-consumer relationships can be identified between pairs of tasks
- The source task must produce data linearly, and the target task must consume it
- May indicate the presence of dataflow regions, which may be exploited by our holistic partitioning and optimization algorithm!

Experimental Results

Suite	Benchmark	#Tasks	#Edges	#Subgraphs	Avg. Parallelism Level	%Producer/ Consumer Relationships
MachSuite	backprop	23	130	5	1.22	22.3%
	fft-transpose	21	121	6	1	19.8%
	kmp	4	20	2	1.5	10.0%
	sort-merge	5	28	2	4	17.9%
	sort-radix	15	46	3	3.5	21.7%
Rosetta	3d-rendering	23	96	6	1.52	11.5%
	digit-recognition	12	45	4	1	15.6%
	face-detection	26	219	8	1.62	6.8%
	optical-flow	9	84	2	8	0.0%
	spam-filter	6	29	2	1	10.3%

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