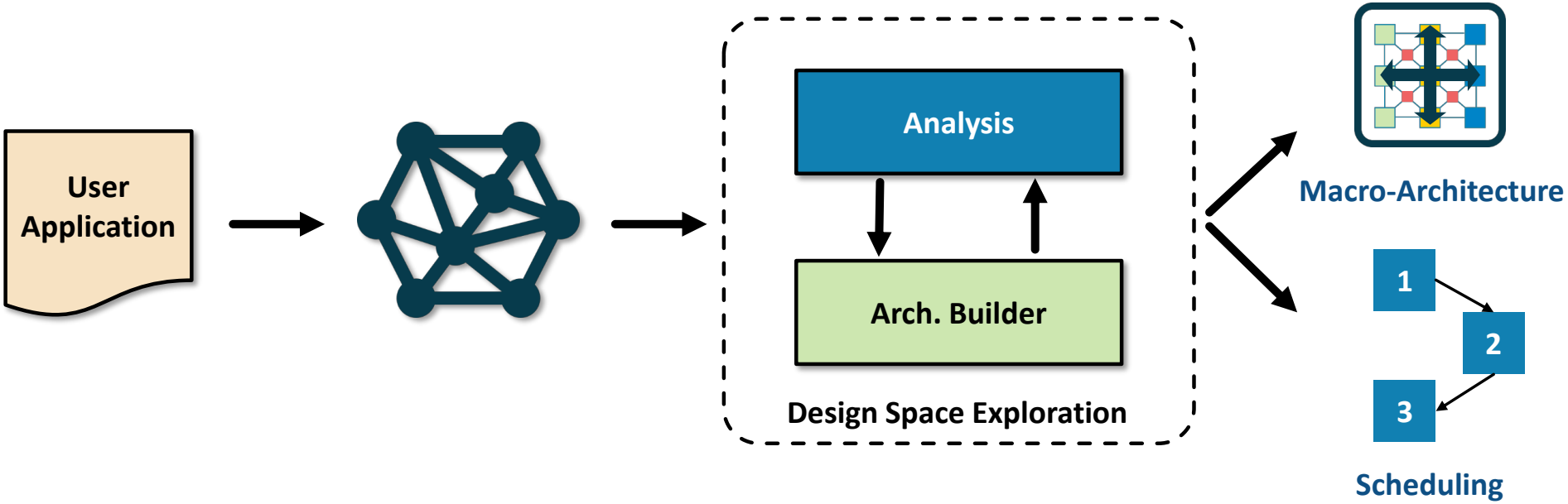
ASA: Application-Specific Architecture

Tiziano De Matteis, Torsten Hoefler

Motivations

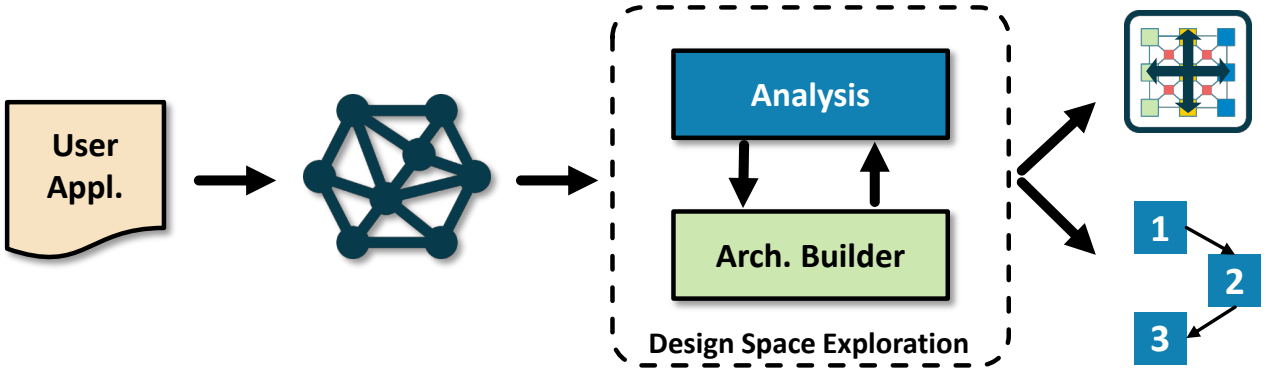
Instead of having an application being adapted and optimized for a given architecture (DSA or SoC), **how we can build a Spatial Specialized Architecture to efficiently support a given application? How we can map the application on such architecture?**

FCL Proposal



SDRs and Deep Learning as driving use cases

First milestone



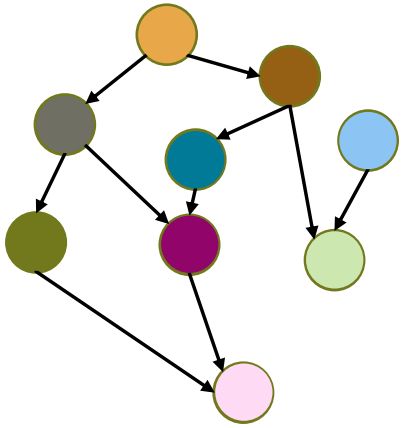
The application is described as a Task Graph, with explicit data-flow information

We want to analyze and understand how we can efficiently **map the application on a given architecture**:

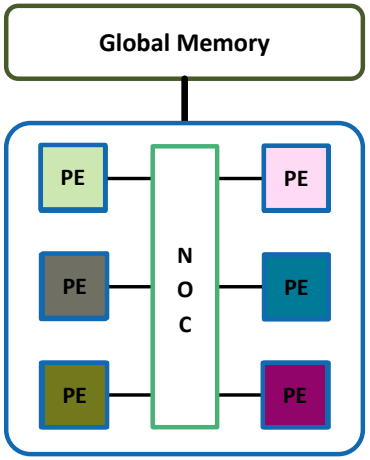
- taking into account spatial parallelism: the architecture is composed by **multiple homogeneous PEs**
- dealing with architecture specific features, such as the presence of a **fast on-chip interconnect**

Applications identification: PUSCH Frequency Algorithm, ML models such as Resnet, Transformer Encoder

Scheduling on a Spatial Device

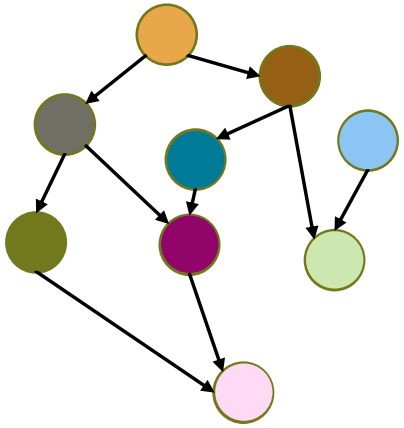


Schedule on

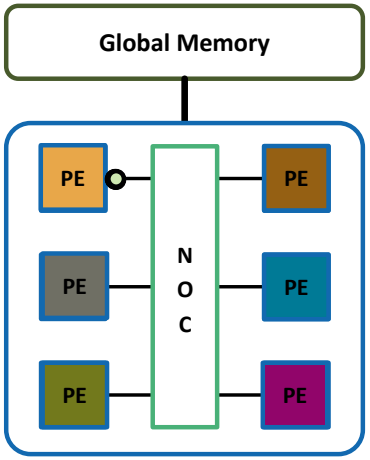


The computation can be performed both
spatially and **temporally**

Scheduling on a Spatial Device



Schedule on



The computation can be performed both **spatially** and **temporally**

Pipelining is crucial to fully exploit the device's spatial parallelism, allowing concurrent execution of multiple tasks

Problem definition

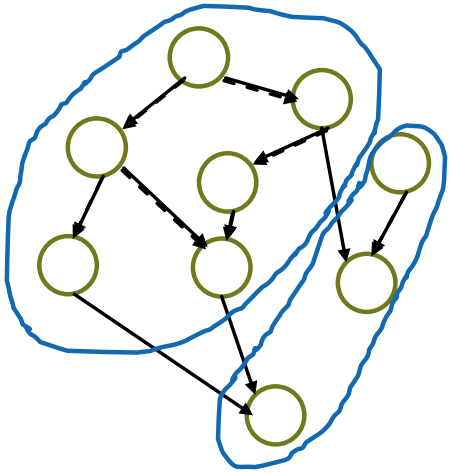
The user application is described by a **direct acyclic task graph** $G = (V, E)$. Task dependencies and communication volumes are known a priori

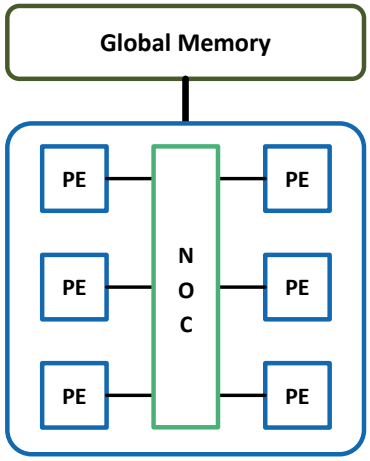
The target platform is constituted by **P** homogeneous PEs, that can communicate directly with each other (NoC) or with Global Memory

Communications among tasks can be **buffered** (solid edges) or **pipelined** (dashed edges)

Goal: find a static schedule for the application on the given architecture:

1. Partition the graph into temporally multiplexed **spatial blocks**
2. Schedule the spatial blocks one-after-the-other
3. Insert appropriate buffers to **prevent deadlocks**

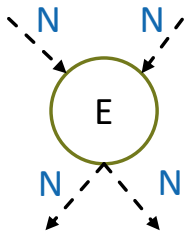




Canonical Task Graphs

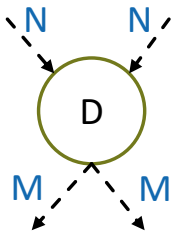
To facilitate the analysis and partitioning of the Task Graph, we consider graphs with certain characteristics

We define as **canonical**, a node that that has a bounded number of input and output edges, that receives the same amount of data from all its input edges, and produces the same amount of data to all its output edges



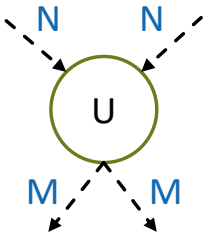
Element-Wise

$R(v) = 1$



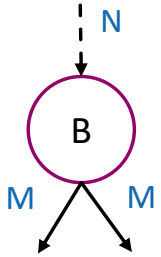
Down-sampler

$R(v) < 1$



Up-sampler

$R(v) > 1$



Buffer node

For a node v, we define the production rate as the ratio between output and input data $R(v) = O(v)/I(v)$

For any node its running time (in isolation) is given by: $\max\{I(v), O(v)\}$

We introduce **buffer nodes**: a buffer node buffers its inputs, and once all input elements have been stored, each element is output $R(v)$ times. We can not stream **through** a buffer node.

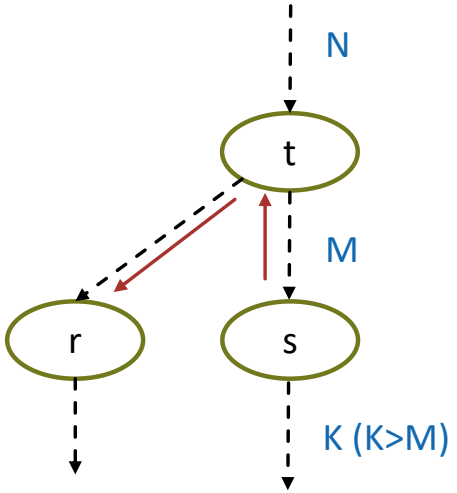
Canonical Task Graphs

A graph composed by only canonical nodes is a **canonical task graph**

Interesting computation that may have different input volumes, or more complicated behavior, can be mapped to a canonical task graph

The representation as canonical DAG depends on the actual implementation

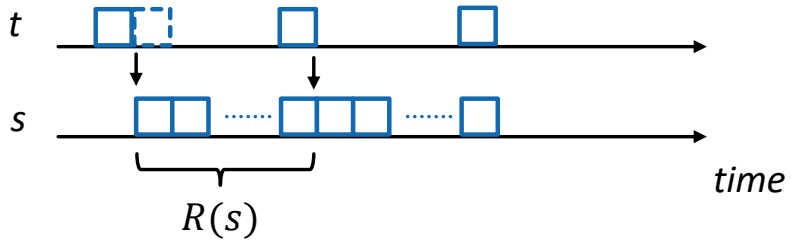
Tasks execution behavior: the upsampler case



The running time of a task t is given by the volume of data being ingested/consumed

$$\overline{T_t} = \max\{N, M\}$$

Given an up-sampler s with production rate $R(s)$, it will ingest an input element every $R(s)$ unit of time, producing a new output element every unit of time



An up-sampler node may slow-down its parents. This affects its running time and the running time of other parent's descendants

$$T_t \geq \overline{T_t}$$

The backpressure effect must be taken into account while scheduling the tasks

Streaming intervals

We are interested in analyzing the task graph at the **steady state**

We define for each edge *e* its **streaming interval *s(e)*** *a*s the average interval between elements going through the edge *e* while the edge is streaming.

All streaming intervals must satisfy the condition $s(e) \geq 1$

At the steady-state, a node will read data at an interval given by the maximum streaming interval of its incident input edges. It follows that given a node v, all its incident input edges will have the same streaming interval

For a given node v its output streaming interval depends on the production rate $S^+(v) = \frac{S^-(v)}{R(v)}$

Streaming intervals

We can prove the following interesting results. Let $WCC(v)$ be the weakly connected component that contains the node v. Then:

$$S^+(v) = \frac{\max_{u \in WCC(v)} I(u)}{O(v)}$$

Given a set of nodes, we can just look at the weight over the edges to understand their steady-state-behavior (no queuing theory, no complex analysis)

Last output: the last-out time LO (v) is the time the last element leaves node u

First output: the first-out time FO (v) is the time the first element leaves node u

$$LO(v) = \max_{(u,v) \in E(G)} LO(u) + \begin{cases} \lceil (R(v) - 1) S^+(v) \rceil + 1 & \text{if } R(v) > 1 \\ 1 & \text{else.} \end{cases}$$

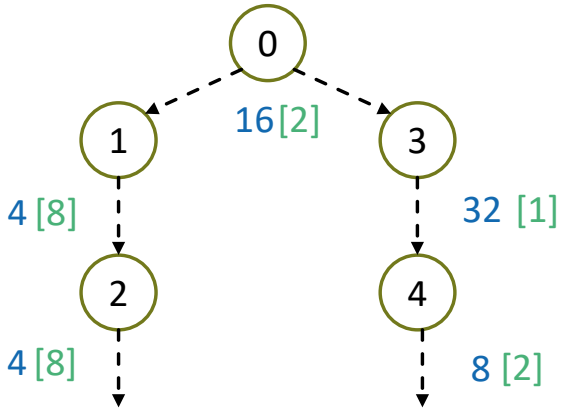
$$FO(v) = \max_{(u,v) \in E(G)} FO(u) + \begin{cases} \left\lceil \left(\frac{1}{R(v)} - 1 \right) S^-(v) \right\rceil + 1 & \text{if } R(v) < 1 \\ 1 & \text{else.} \end{cases}$$

If v is a source node, than $LO(v) = (O(v) - 1)S^+(v)$ and $FO(v) = 0$

We can use this information to schedule streaming tasks (inside a spatial block)

Example

Let's see how to schedule the following streaming nodes over 5 PEs



We use the LO/FO to decide the starting/ending time of each task

Task	Start	LO	FO

$$LO(v) = \max_{(u,v)\in E(G)} LO(u) + \begin{cases} \lceil (R(v)-1)S^{+}(v) \rceil + 1 & \text{if } R(v) > 1 \\ 1 & \text{else.} \end{cases}$$

$$FO(v) = \max_{(u,v)\in E(G)} FO(u) + \begin{cases} \left\lceil \left(\frac{1}{R(v)} - 1 \right) S^{-}(v) \right\rceil + 1 & \text{if } R(v) < 1 \\ 1 & \text{else.} \end{cases}$$

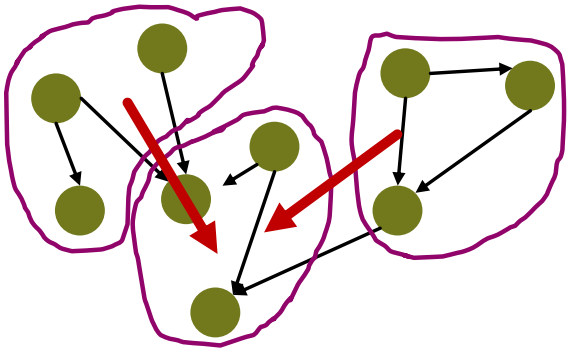
Partition into Spatial Blocks as an Optimization Problem

When we have P < N, we need to partition the DAG into Spatial Blocks of at most P nodes. Such partitioning must be done so that the overall execution time $(\max_{\{v \in V\}} LO(v))$ is minimized

The discussion in the previous slides show the last-out time relates to the maximum amount of data produced by the nodes in the graph. This result allows us to define the scheduling problem as an optimization problem.

Given a canonical task graph, we want to partition it into Spatial blocks containing at most P computational nodes, such that:

- The sum of the max value of each Spatial Block component is minimized
- The dependencies among SB still form an acyclic DAG

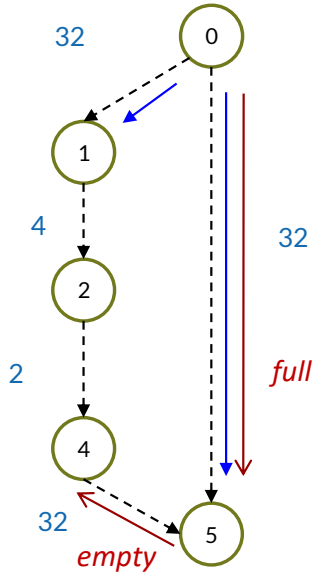


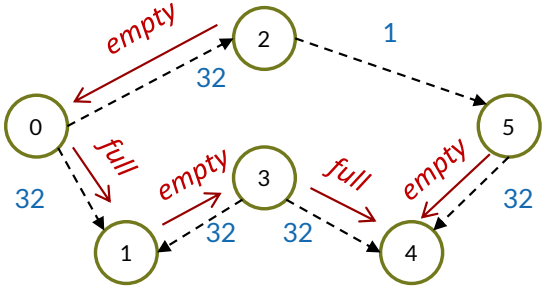
This allow us to define an effective heuristic for the Spatial Block partitioning:

- we add a node to a spatial block if its produced data volume is less than the data volume produced by the block's source(s) from which it depends (if any).
- We continue adding to the same spatial block until such node exists or the block is full. Otherwise, we create a new spatial block, and we start filling it.

Buffer Space

Despite dealing with DAGs, we can still experience deadlocks when we stream. What situation can lead to this?



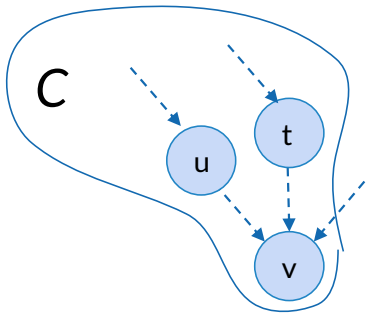


We have to look at **all the undirected cycles** inside a streaming component

We don't want to just avoid deadlocks, we want to avoid **bubbles** as well.

Computing the buffer space

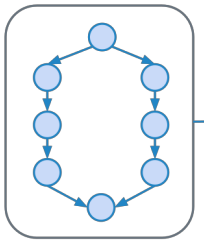
Ideally, we want to check all nodes in an undirected cycle that have at least two predecessor (from that cycle), and evaluate the difference in the FOs



$$B(u,v) = (\max_{i \in \{u,t\}} FO\,(i) - FO(u))/\,S(v)$$

Proper algorithms to do this efficiently (linear time in the number of nodes and edges)

General scheduling toolchain



INPUT

Results

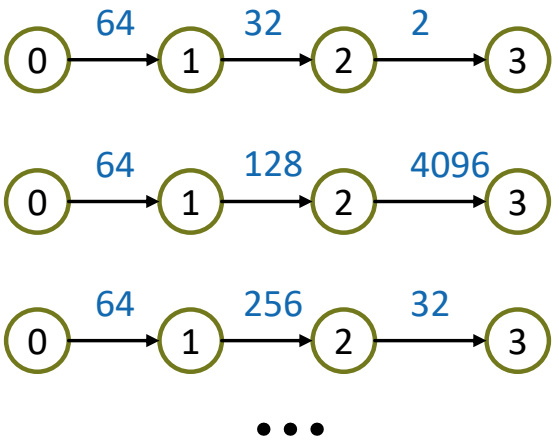
Implemented the analysis, heuristics (spatial blocks, scheduling), and buffer space computation to assess our findings:

- Written in Python
- DAG is represented by using a direct Graph (networkx) with weights over the edge
- The framework takes in input the DAG, the number of PEs and applies the toolchain discussed before

Considered random graph generated from different use cases (Chain, FFT, Gaussian Elimination, Cholesky)

Compared our heuristic vs. a non-streaming one (Earliest Finishing Time).

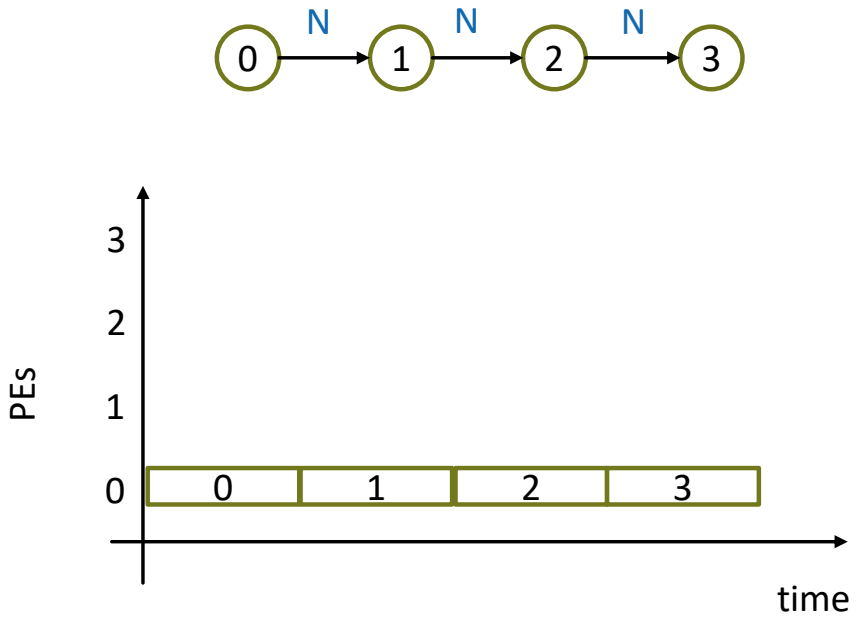
In the following the results are derived considering 100 random DAGs

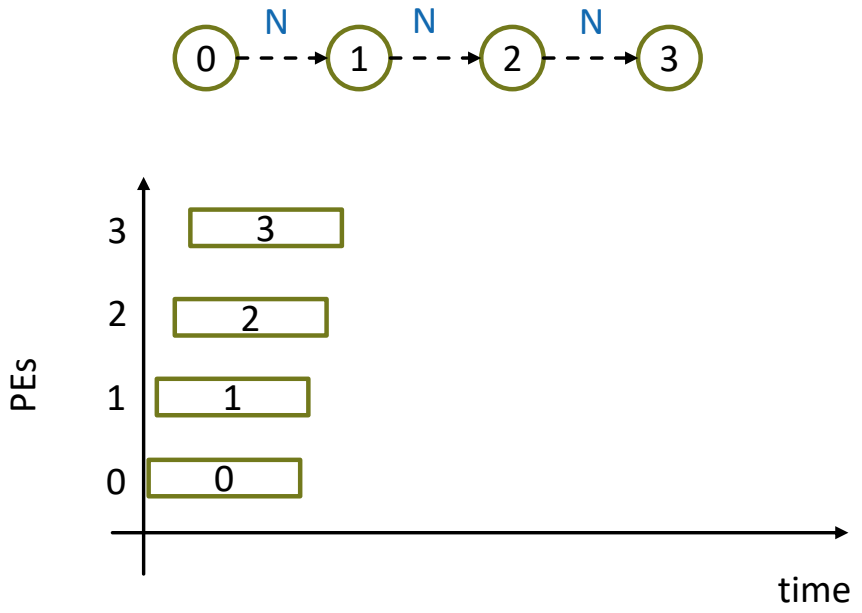


Speedup

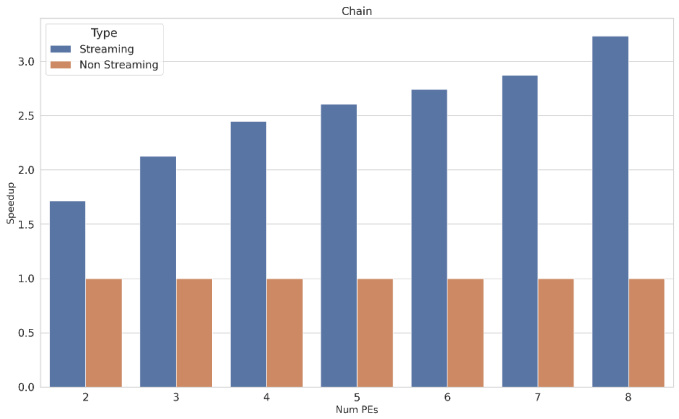
We define the **speedup** as the ratio between the makespan (application running time) of the schedule with P Processing Elements, over the makespan obtained with 1 PE

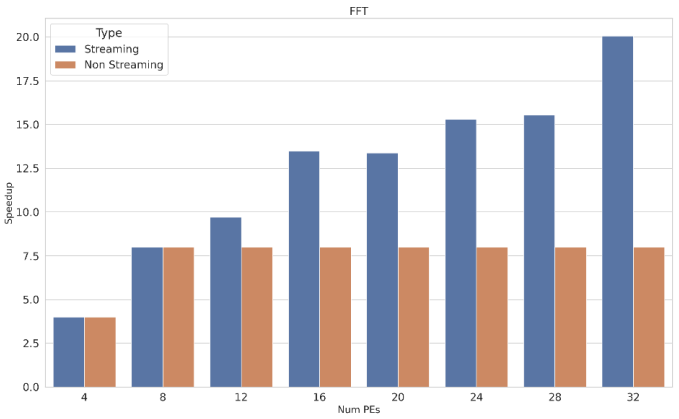
We expect a greater speedup wrt non-streaming scheduling, given by the pipelining of consecutive tasks

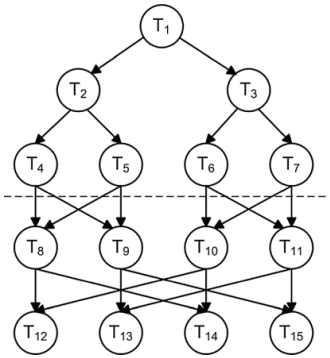


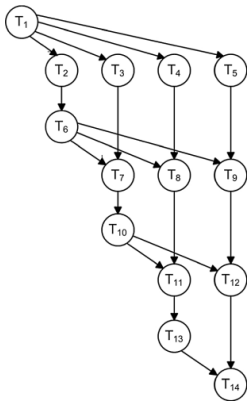


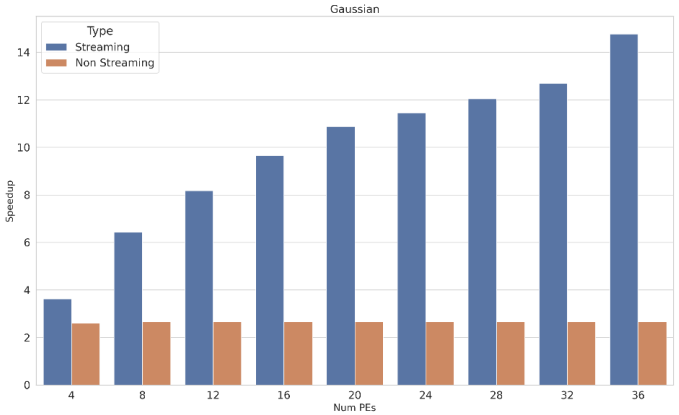
Speedup

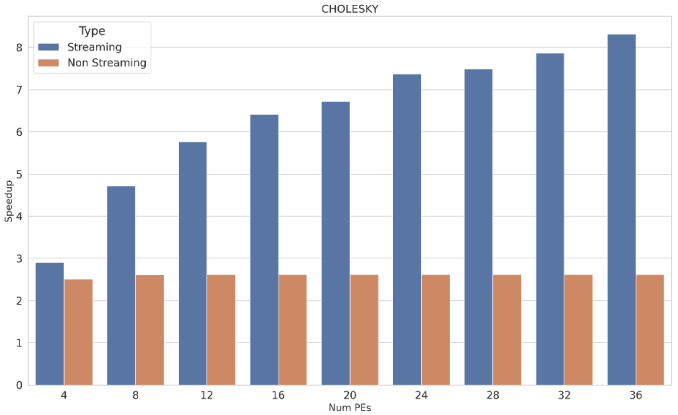

Chain
Type
Streaming
Non Streaming
Speedup
Num PEs


FFT
Type
Streaming
Non Streaming
Speedup
Num PEs


T1
T2
T3
T4
T5
T6
T7
T8
T9
T10
T11
T12
T13
T14
T15


T1
T2
T3
T4
T5
T6
T7
T8
T9
T10
T11
T12
T13
T14

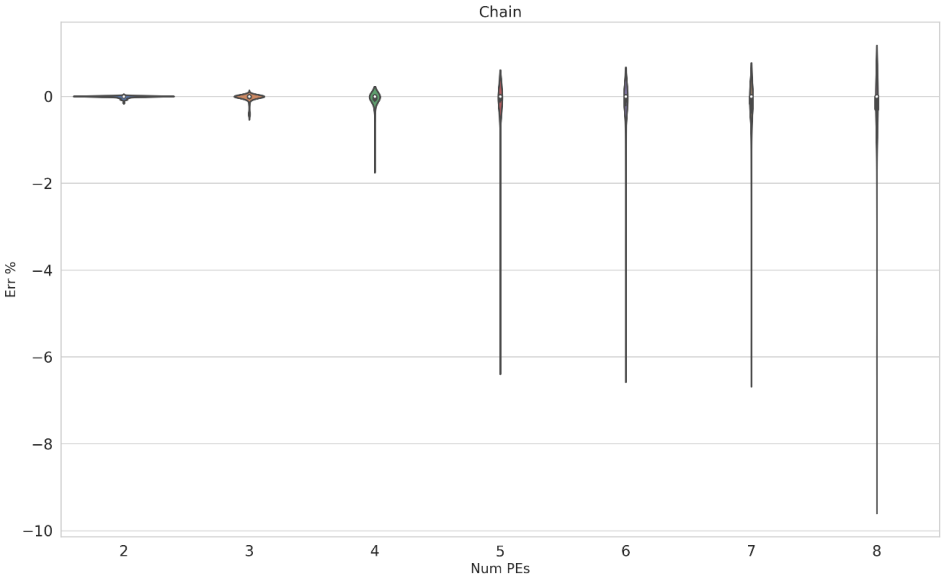

Gaussian
Type
Streaming
Non Streaming
Speedup
Num PEs

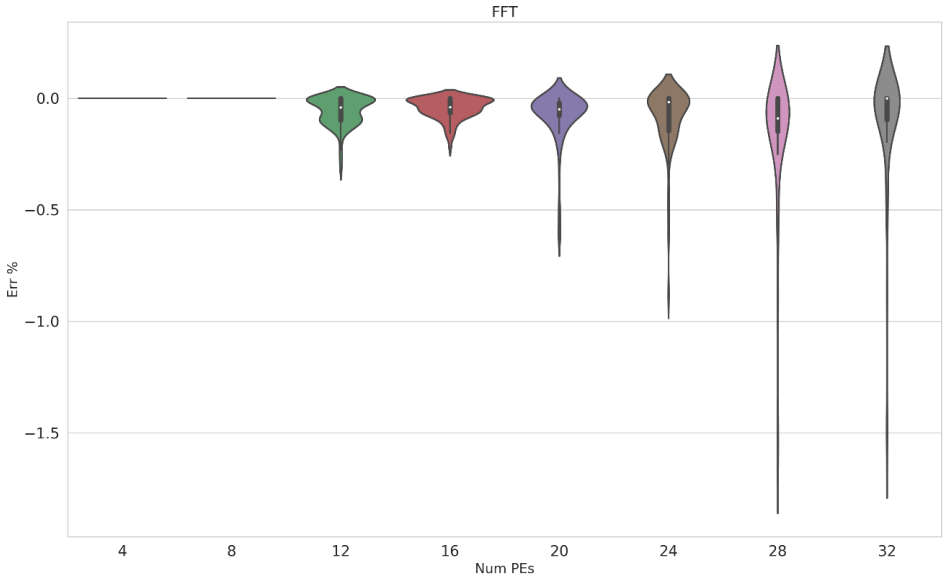

CHOLESKY
Type
Streaming
Non Streaming
Speedup
Num PEs

Simulation

We implemented a Discrete Event Simulation that mimics the Task Graph execution. We want to:

- Verify that the buffer space we computed is sufficient to prevent deadlocks
- Check how far are the scheduling and simulation makespan, to assess the quality of our analysis

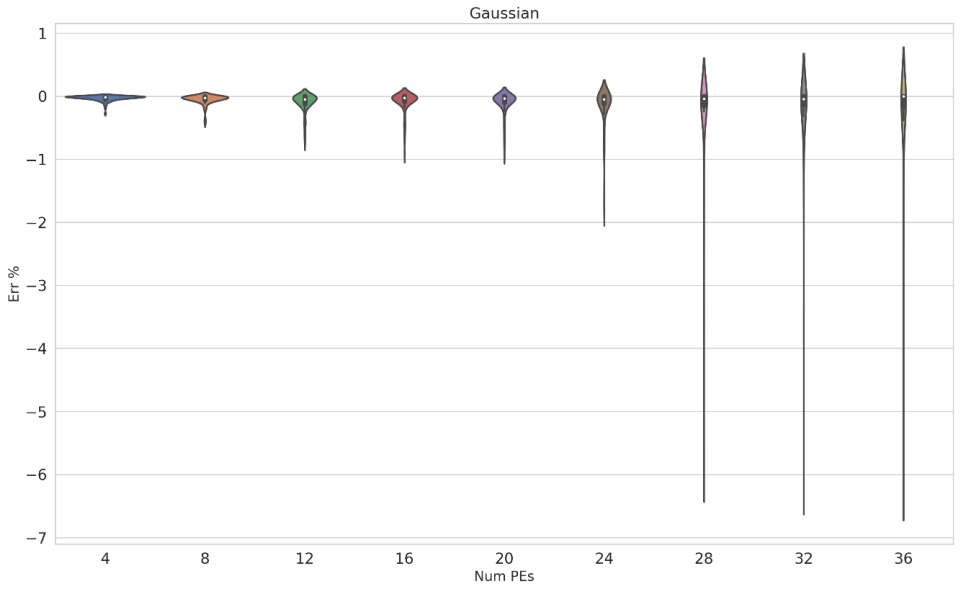


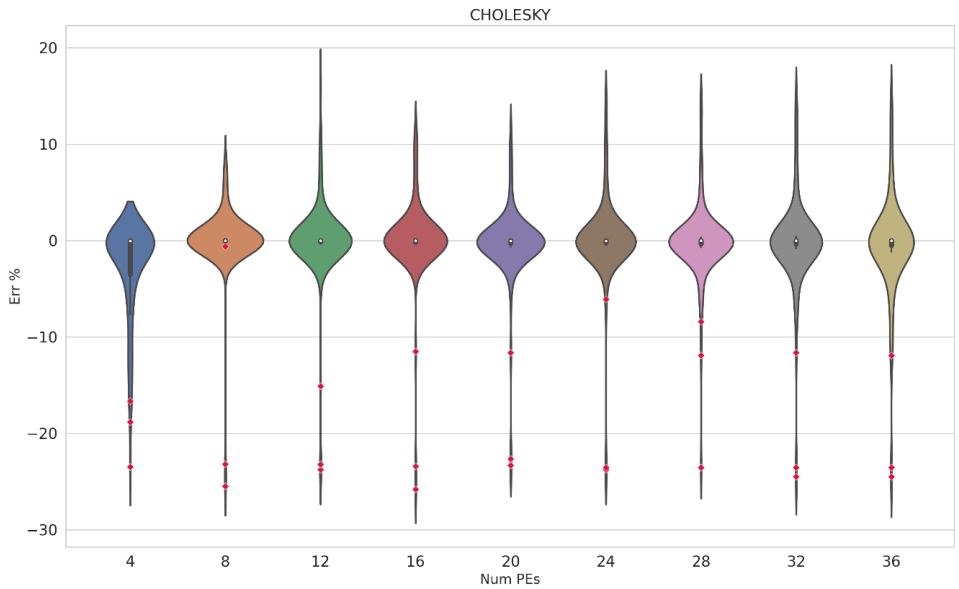


Simulation

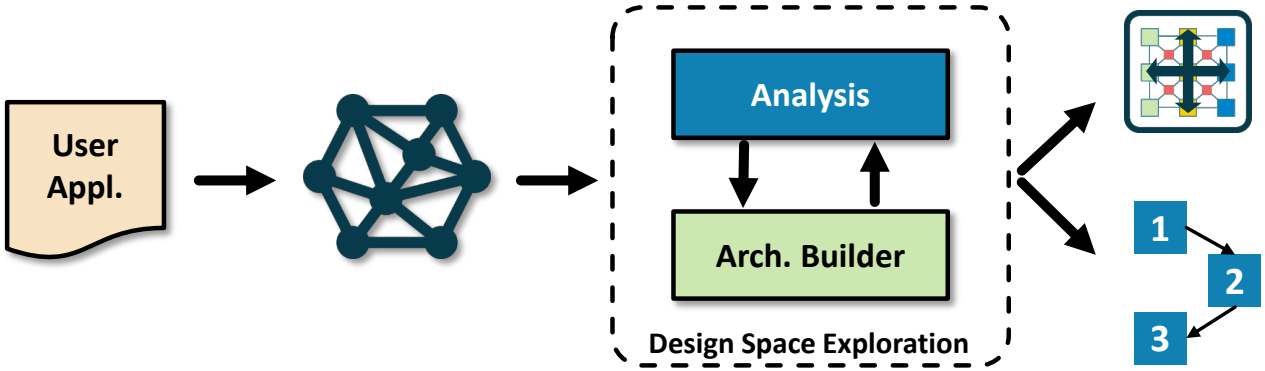
We implemented a Discrete Event Simulation that mimics the Task Graph execution. We want to:

- Verify that the buffer space we computed is sufficient and the Task Graph does not deadlock
- Check how far are the scheduling and simulation makespan, to assess the quality of our analysis





Conclusions and next steps



In this first part of the project we focused on the problem of scheduling a given application on a given architecture. The scheduling toolchain will be open-sourced

Now we want to deal with:

- Front-end: the user writes the application in high-level language and we extract the useful information
- Design Space Exploration: not just increasing the number of PEs till we meet the performance requirement.
- Work toward our use cases

Thank you!

spcl.inf.ethz.ch
@spcl_eth

ETH zürich