# CALIPER FRAMEWORK ACCELERATOR

# What is Caliper?

It is a Framework to Compute TimeToFail of a CPU multicore

# What is **Time To Fail**?

Imagine Be a DataCenter manager...

**When should you change CPU of your servers, avoiding to wait them to die?**

(eg: if they die, they could cause Denial Of Service)

**Mean Time To Fail** is the expected time a device can work under some circumnstances before it fails.

# TTF Computation

**But how is TTF actually computed?**

**TTF of a single core CPU**

$$R(t,T) = e^{(-\frac{t}{\alpha(T)})^{\beta}}$$

$$MTTF = \int_0^{\infty} R(t,T)dt$$

# TTF Computation

**But how is TTF actually computed?**

**Reliability of a variable Temperature single core CPU**

$$R(t) = e^{-(\sum_{j=1}^{i} \frac{\tau_j}{\alpha_j(T)})}$$

# TTF Computation

**But how is TTF actually computed?**

TTF of a MultiCore CPU

$$P_{no\_f}(t) = \prod_{i=1}^{n} Ri(t)$$

.....

$$P2\_f(t) = \sum_{i=1}^{n} \sum_{j=1, j\neq i}^{n} \int_{0}^{\inf} \int_{0}^{t2} f_i(t_1) * f_j(t_2|t_1^i) \prod_{m=1, m\neq j,i}^{n} R_m(t|t_1^i, t_2^j) dt_1 dt_2$$

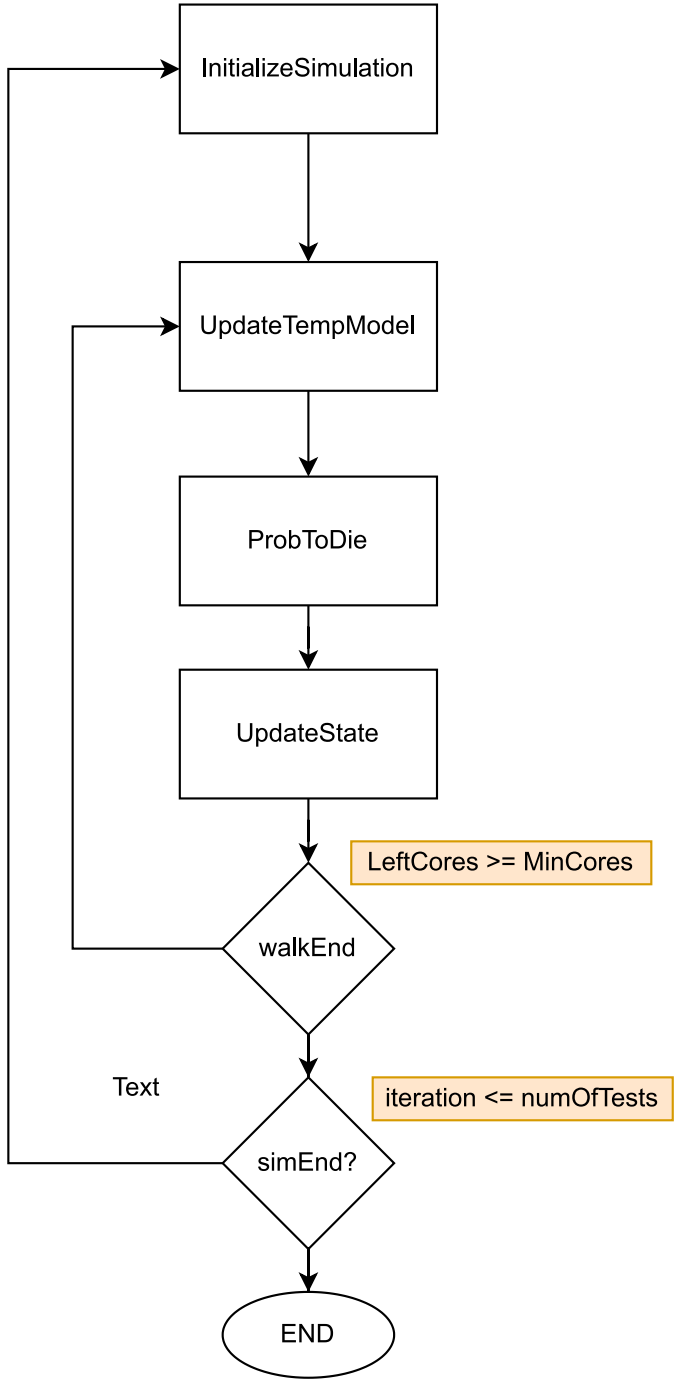
**Infeasible to solve with traditional algorithms...**

# What is Montecarlo?

**What if you need to solve a complex Integral...**
**- How much computational power do we need?**
**- Are classical algorithms capable to solve this particular algorithm?**

Montecarlo is an algorithm that uses "Randomization" techniques to give us an approximated result of this integral within a certain confidence interval.

# How it Works?


InitializeSimulation
UpdateTempModel
ProbToDie
UpdateState
LeftCores >= MinCores
walkEnd
Text
iteration <= numOfTests
simEnd?
END

# Simulation Parameters:

-CURRR

-TEMP

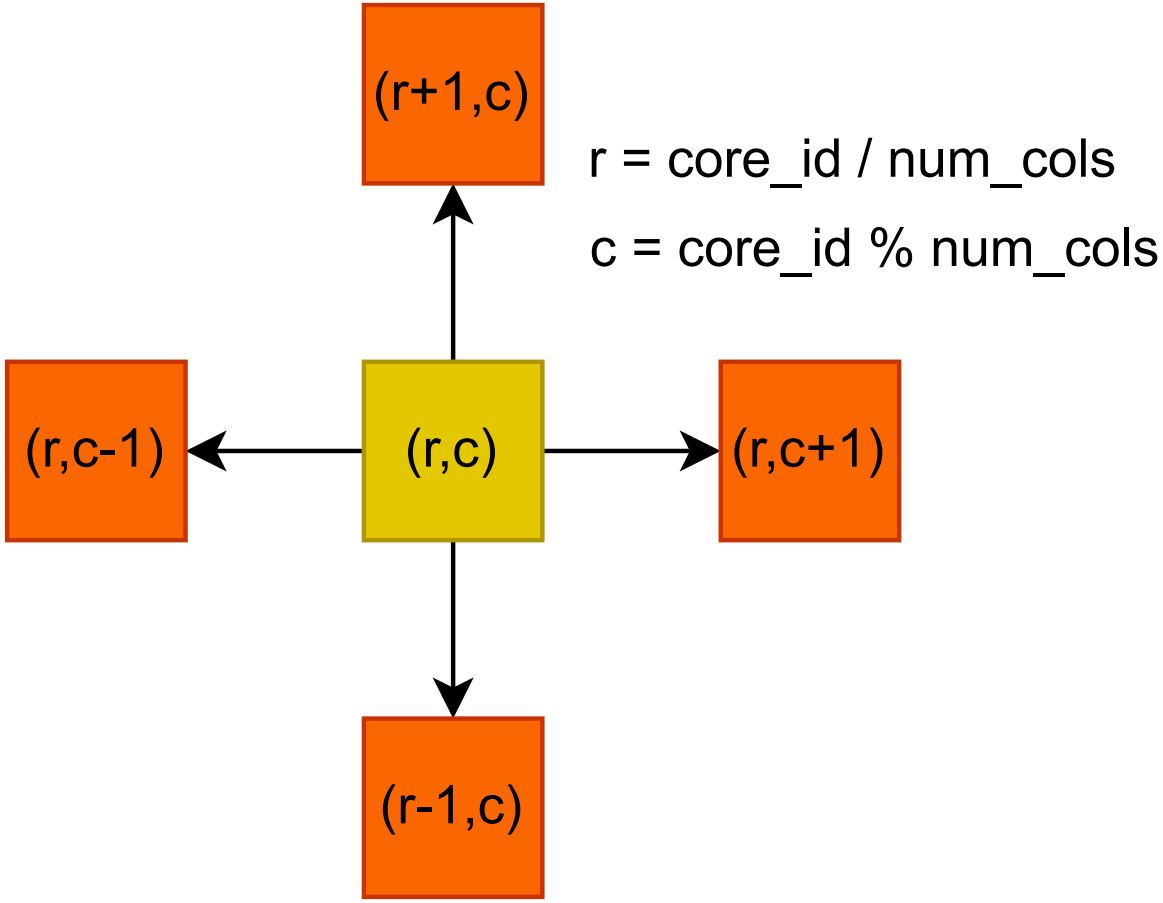
-ALIVE

-LOAD

# Temp Model: $Temp_{(r,c)} = EnvTmp + LD[r][c] * SelfTmp + \sum_{(x,y) \in Neigh} LD[x][y] * NeighTEMP$

**How do each core compute it's updated temperature?**

The temperature is updated by checking neighbours load of work by applying the following formula:



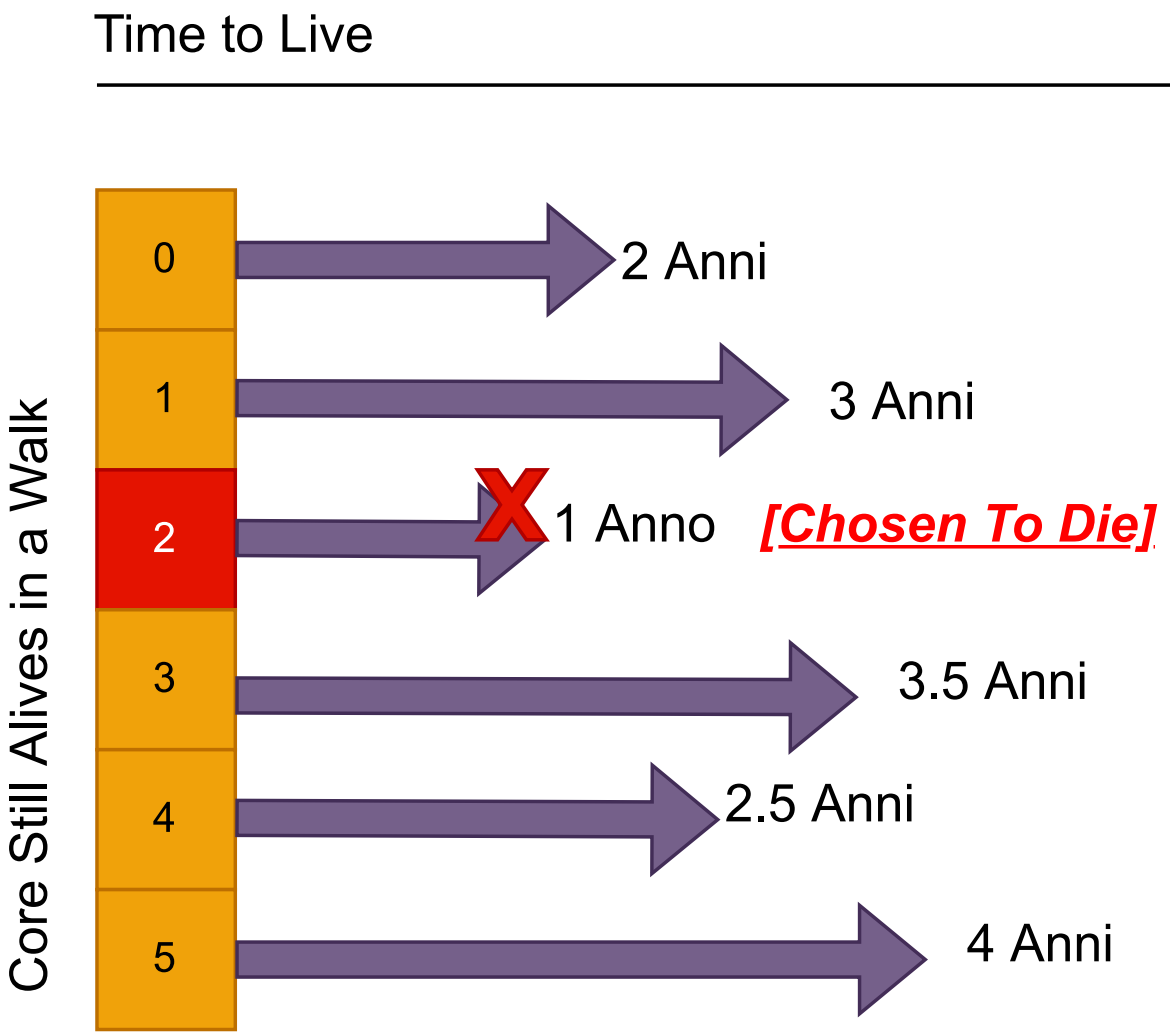
# Probability of Death:

**But how can we choose which core to kill on a certain step of the Random Walk**?

We calculate the estimated time of Death of a core using its reliability function (an exponential model)

Since each core will have different "CurrR" then each core will have different time of life.

We select the core with the shortest remaining time to live.



# Update The Simulation State:

**And Now? How can we update the Reliability function of each core?**

We save the time of death of the selected core and compute

DeltaT

Use DeltaT to calculate a new CurrR for each Core.

# Redistribute The Load

**How we redistribute the Load among cores that are still Alive?**

We simply divide the "ApplicationLoad" with the num of remaining cores.

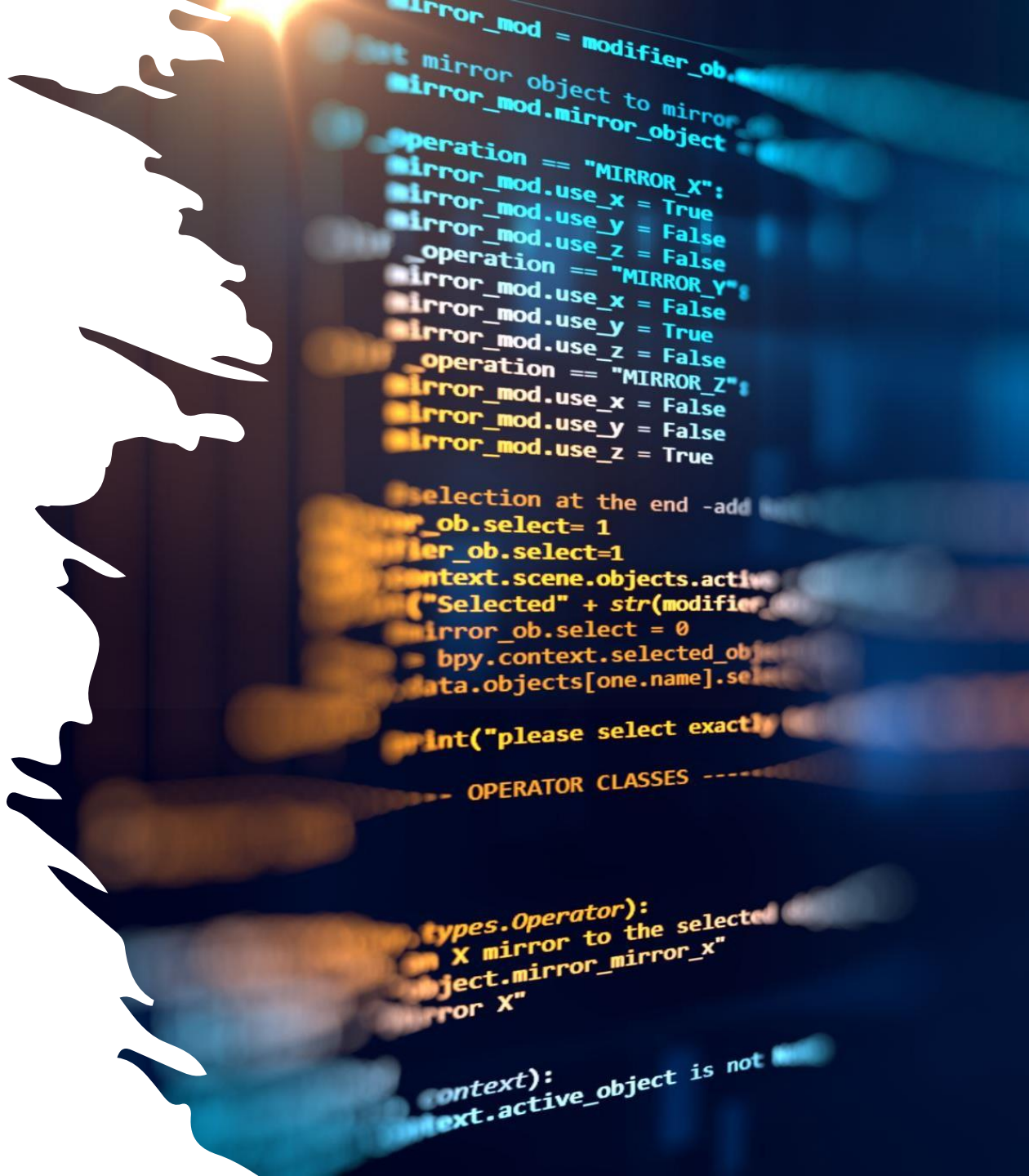

load = 0


load = InitialWorkload * leftcores/max_cores

# Code Optimizations:

The first optimization we tried to do, was on the bare CPU version.

In the next slides will be shown the 2 most relevant optimizations.



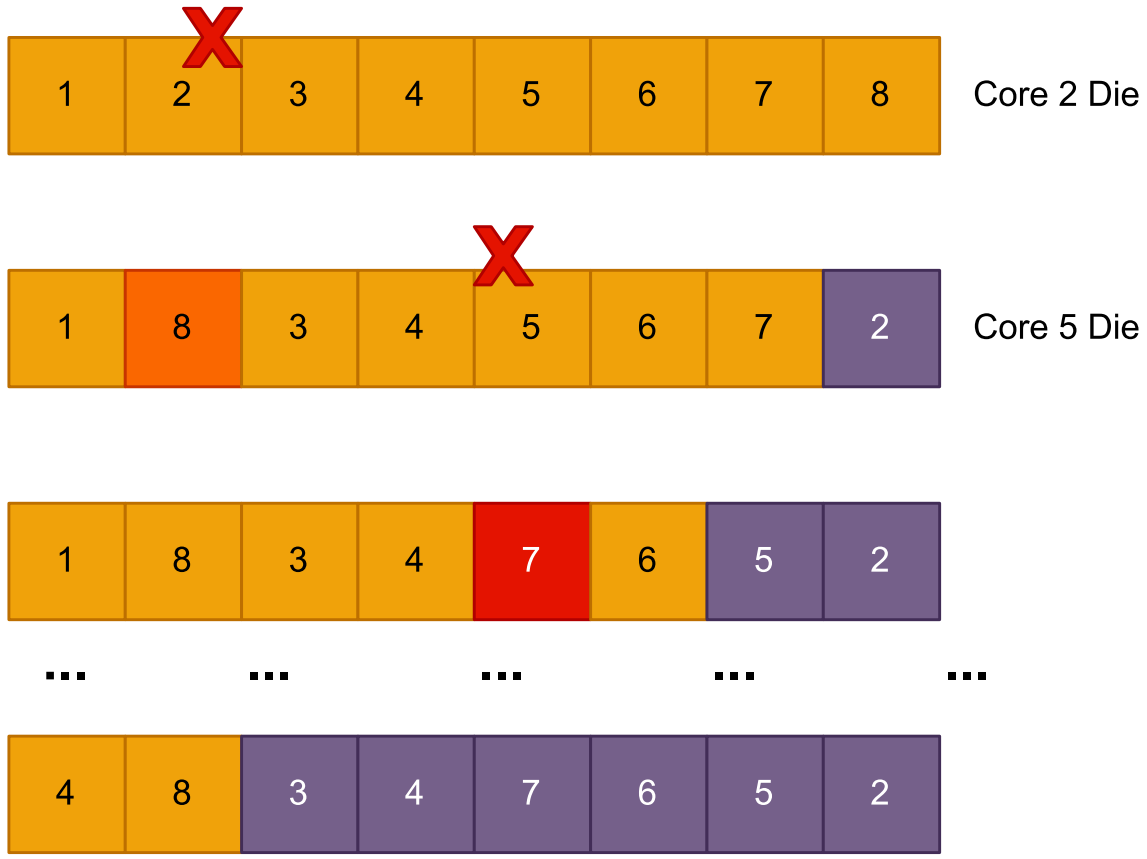
# IDEA 1:
# SWAP DEAD CORE

**Why do we need to cycle through ALL cores if lot of them may be Dead?**

We can Swap the dead cores to the end of the array and during each step of the walk do computation ONLY on alive cores.

This cut drasticly the number of cycles on each walk step.

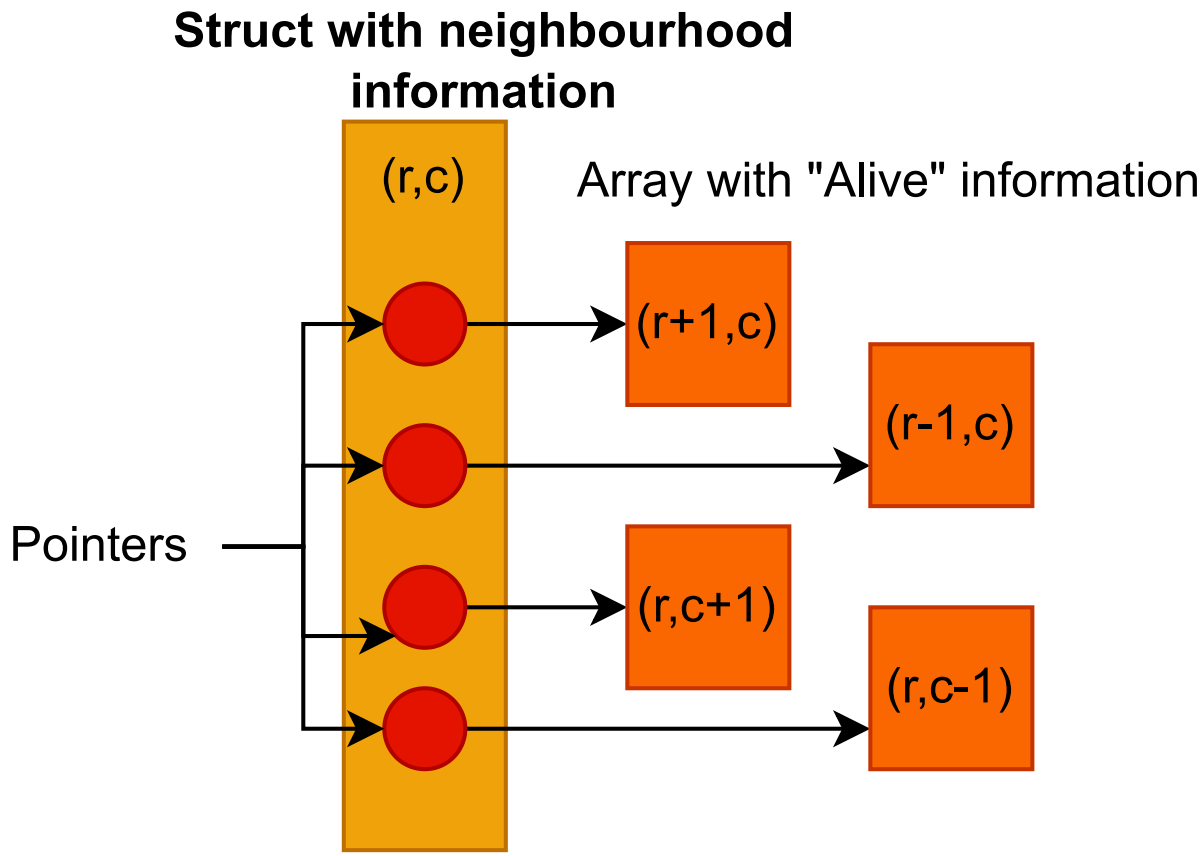
***N = (MaxCores – MinCores***)
- **Old version**
N^2 steps
- **New version**
(N * (N+1)) / 2) steps

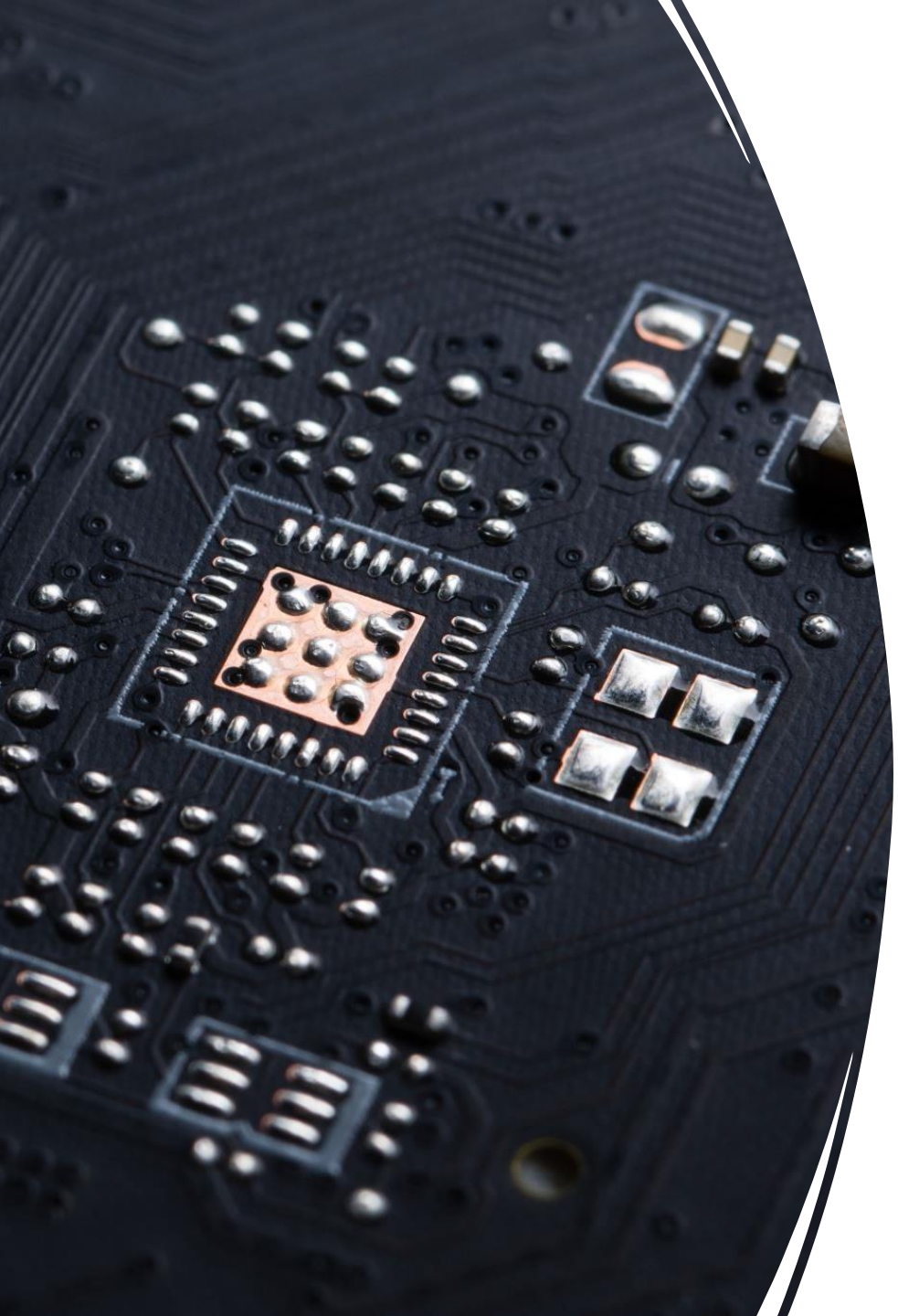


# IDEA 2:

**What if each core save the state of its neighbours?**

We can now compute the temperature of a core without any "matrixBorder" check or any for cycle





# GPU VERSION

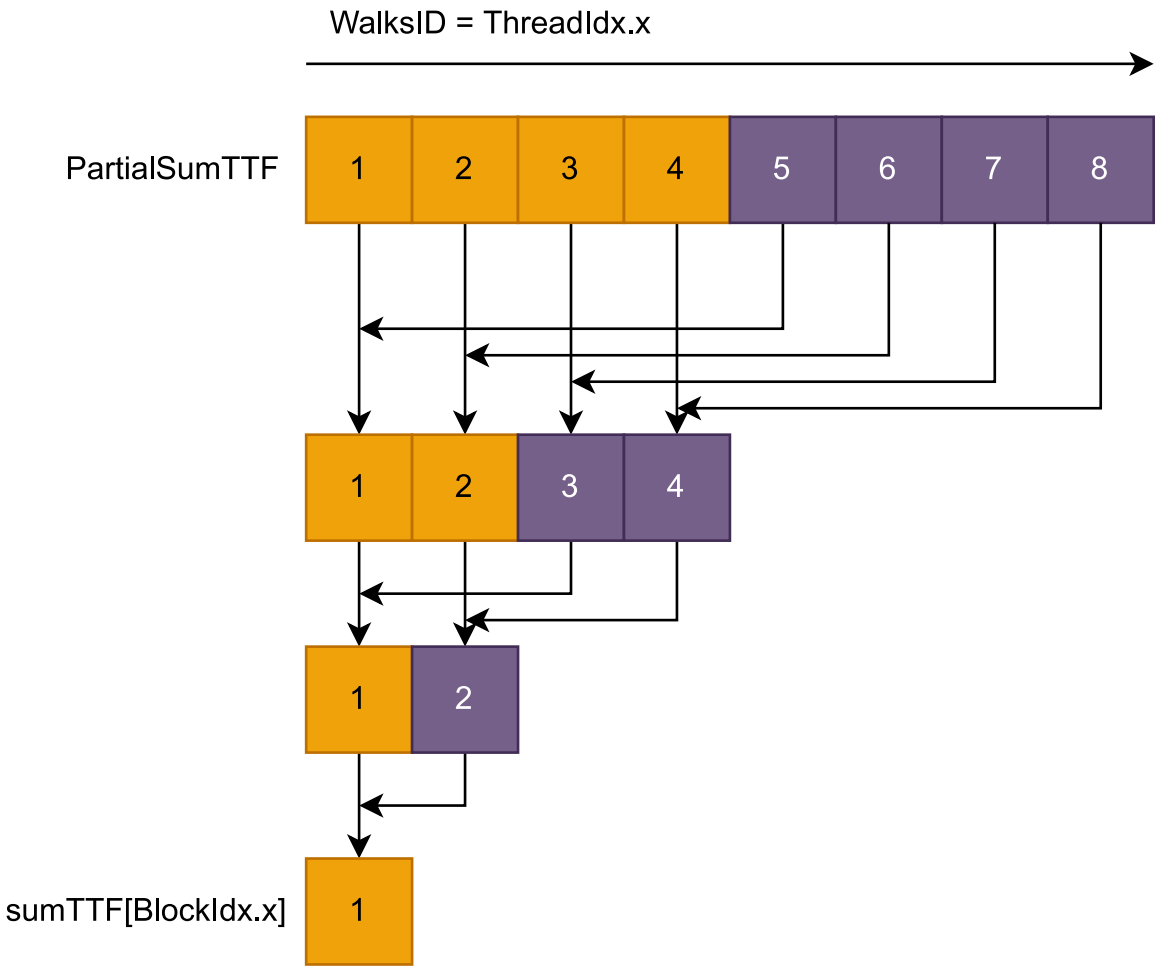
# First Approach: AtomicAdd

The first Approach was to simply parallelize the walks and sum the final **Time To Fail** using an **AtomicAdd**....

This **Trivially results in a bottleneck** on the AtomicAdd

We can Do Better!

# Parallel Reduction (Caliper Redux)




**Why Dont we use Parallel Reduction Design Pattern instead of AtomicAdd?**

Good Idea!

We implemented Parallel reduction optimizing it using **shared memory** to accumulate TTFs of **sameBlock** using Faster memory
Then we use "**GlobalParallelRed**" kernel to accumulate the remaining data.

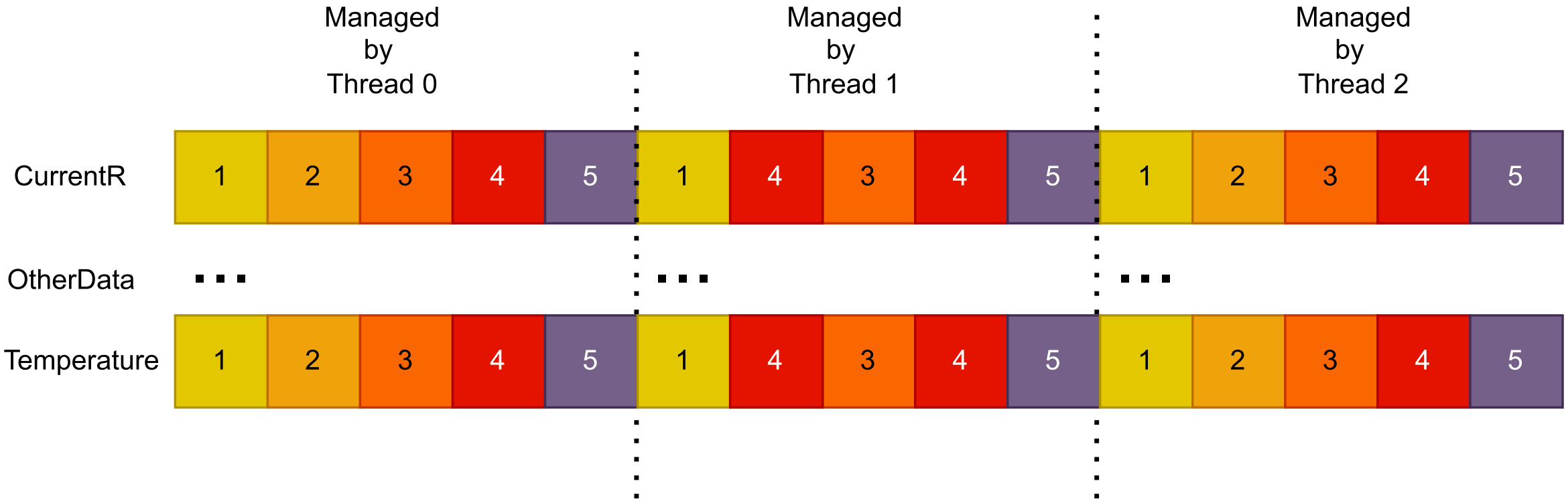
# Memory Models:

**What can we improve of the basic "Redux" Version?**

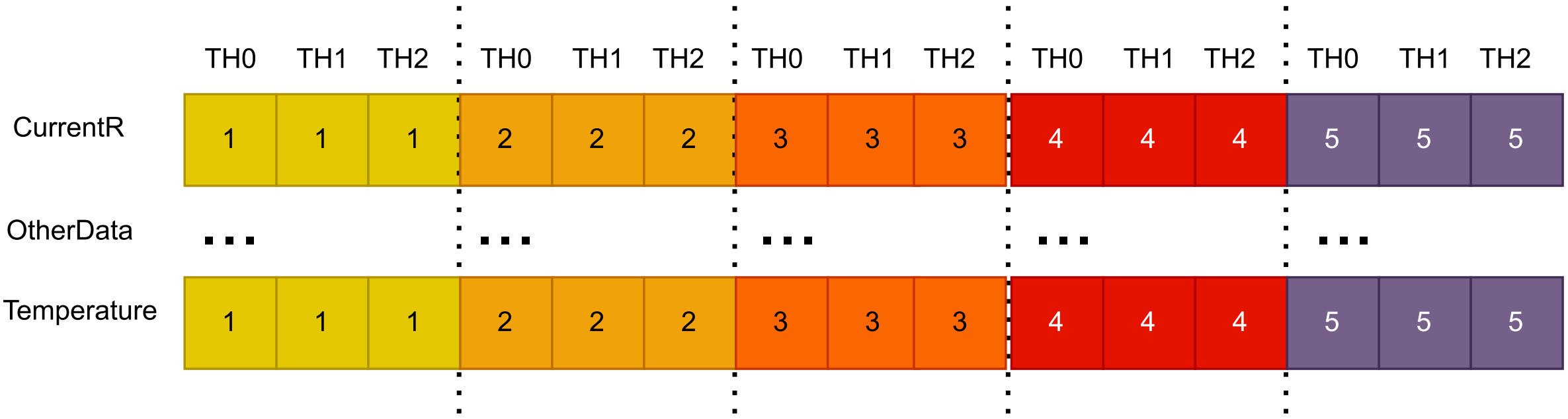
Gpus are complex architectures and a wrong memory management could bring to bad performances, but it is true also the opposite!

In the following slides will be shown different memory Models we tried.

# Sparse Memory Access:



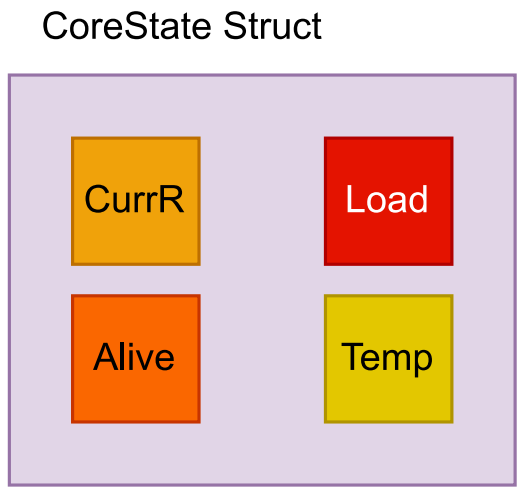
# Coalesced Memory Access:

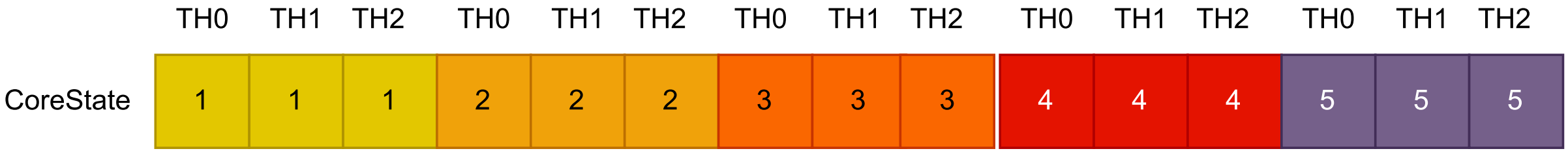


# Struct Coalesced Access(1):

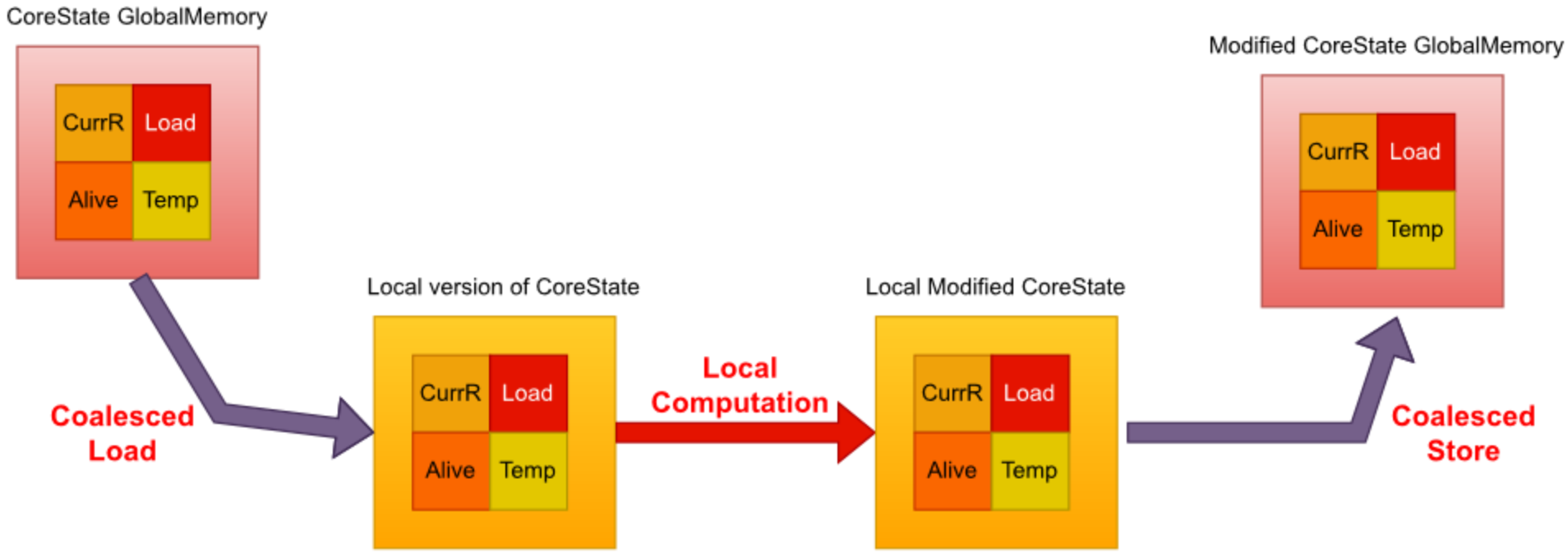
**What if we insert all the simulation parameters into an array of struct???**

Good Idea! But We need some adjustment on how access them to mantain coalesced acceses (Next Slide)



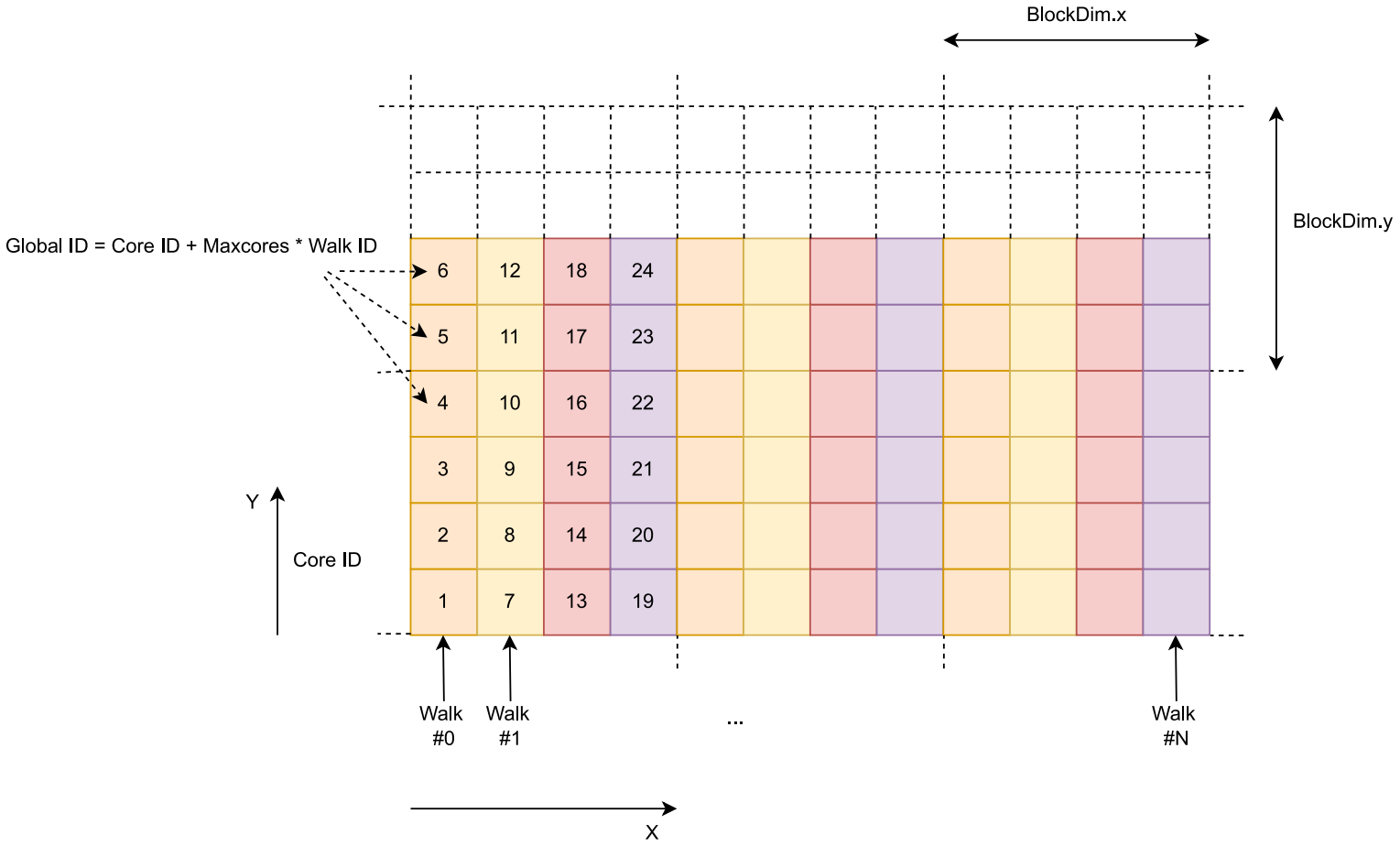
# Struct Coalesced Access(2):



# IDEA for further parallelization

Why don't let one thread account for one single core?

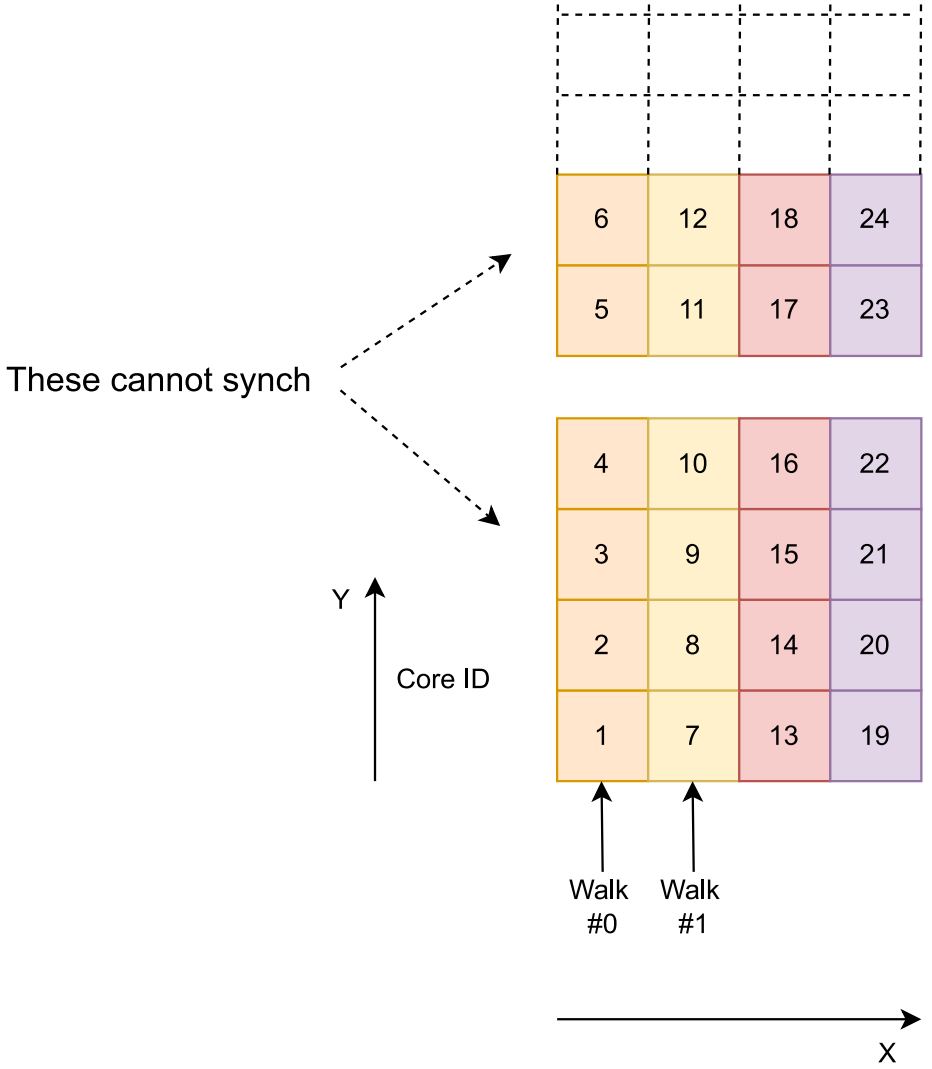
# Generic Grid:



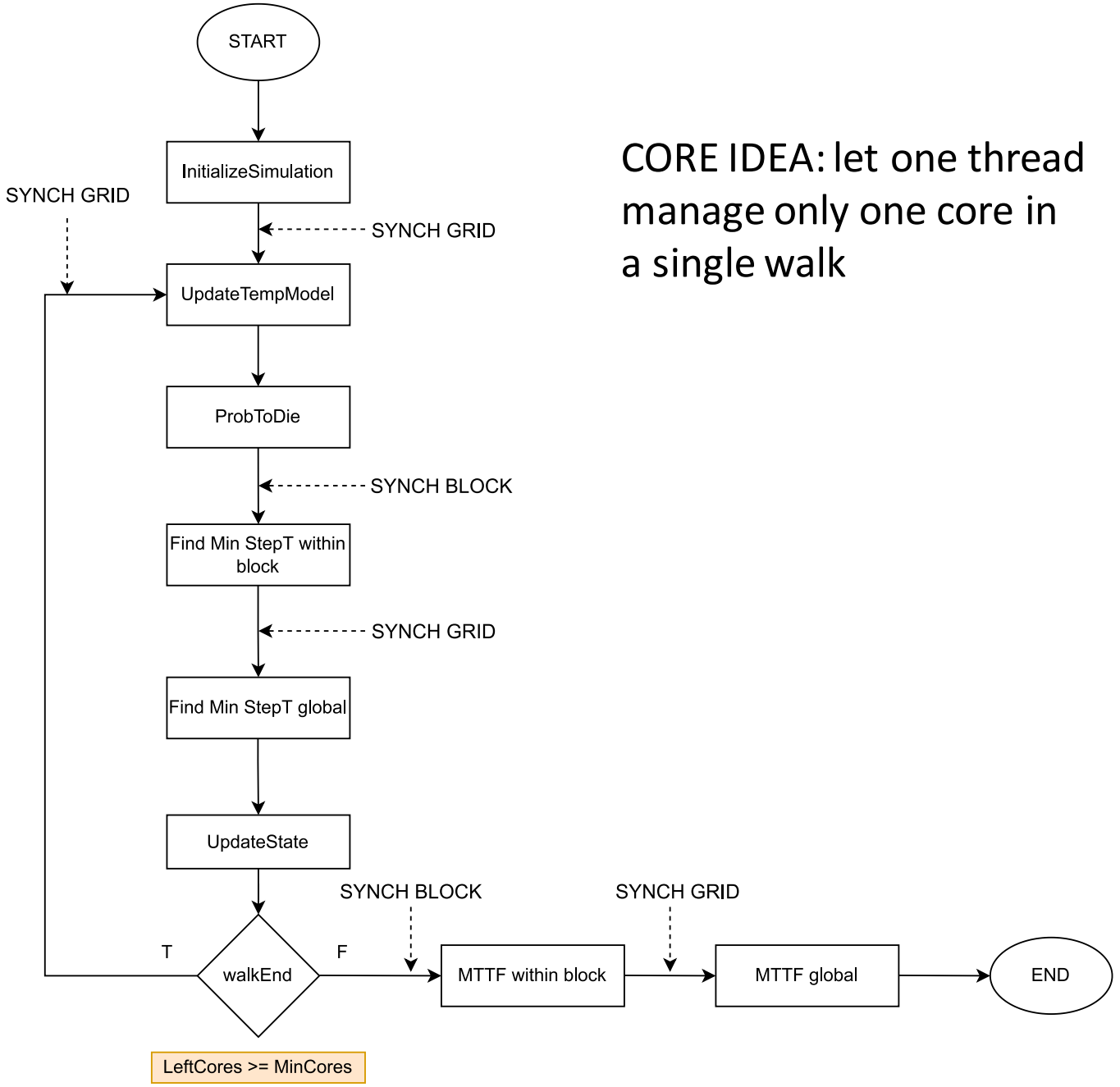

Most generic approach: blocks are defined with square shape and different walks lie into different columns of the grid.

# One new issue to deal with

Rise the need of grid level synchronization when determining min stepT for walks which cores are spread on different blocks.

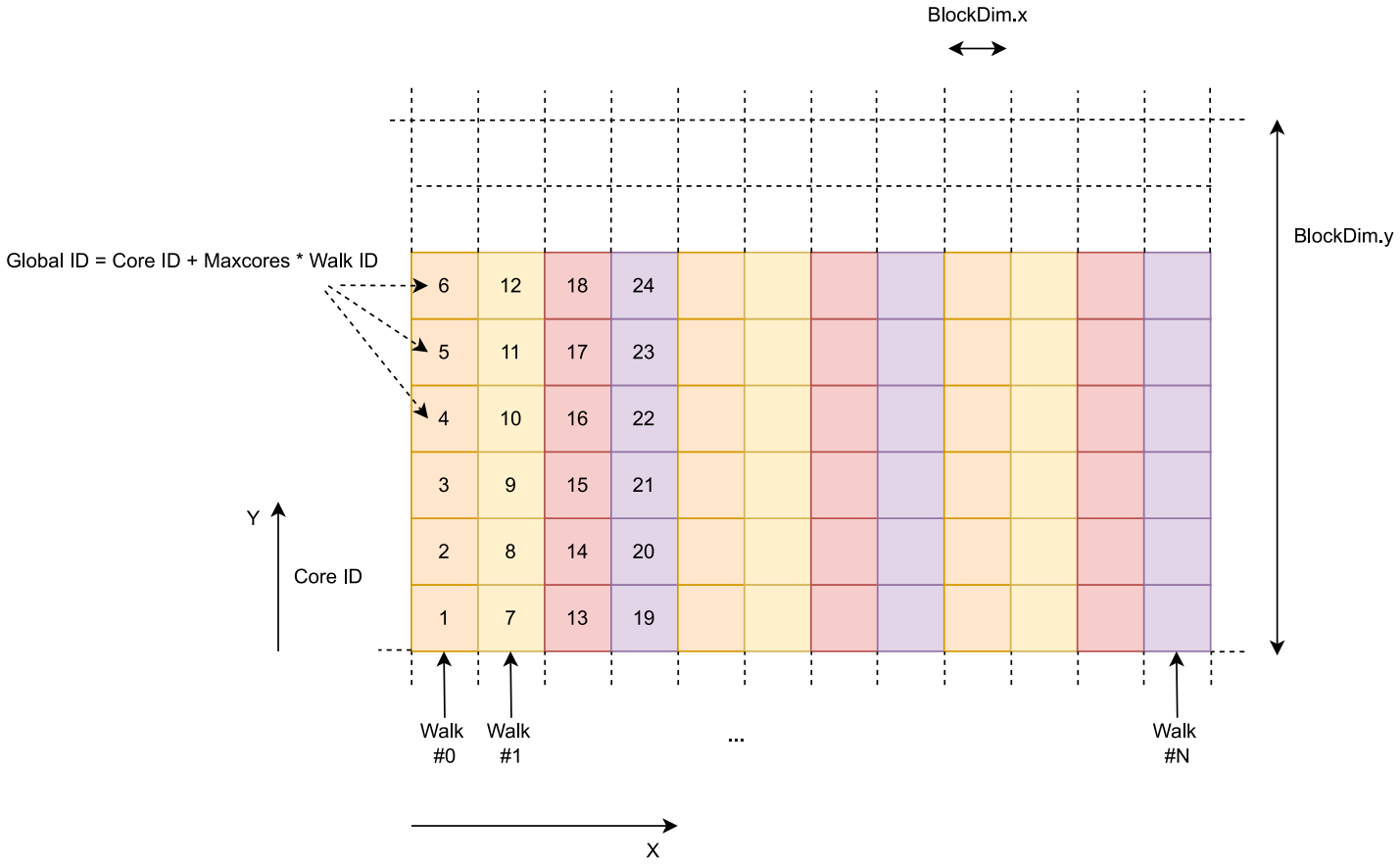


# Grid Versions:




CORE IDEA: let one thread manage only one core in a single walk

# Linearized Grid:

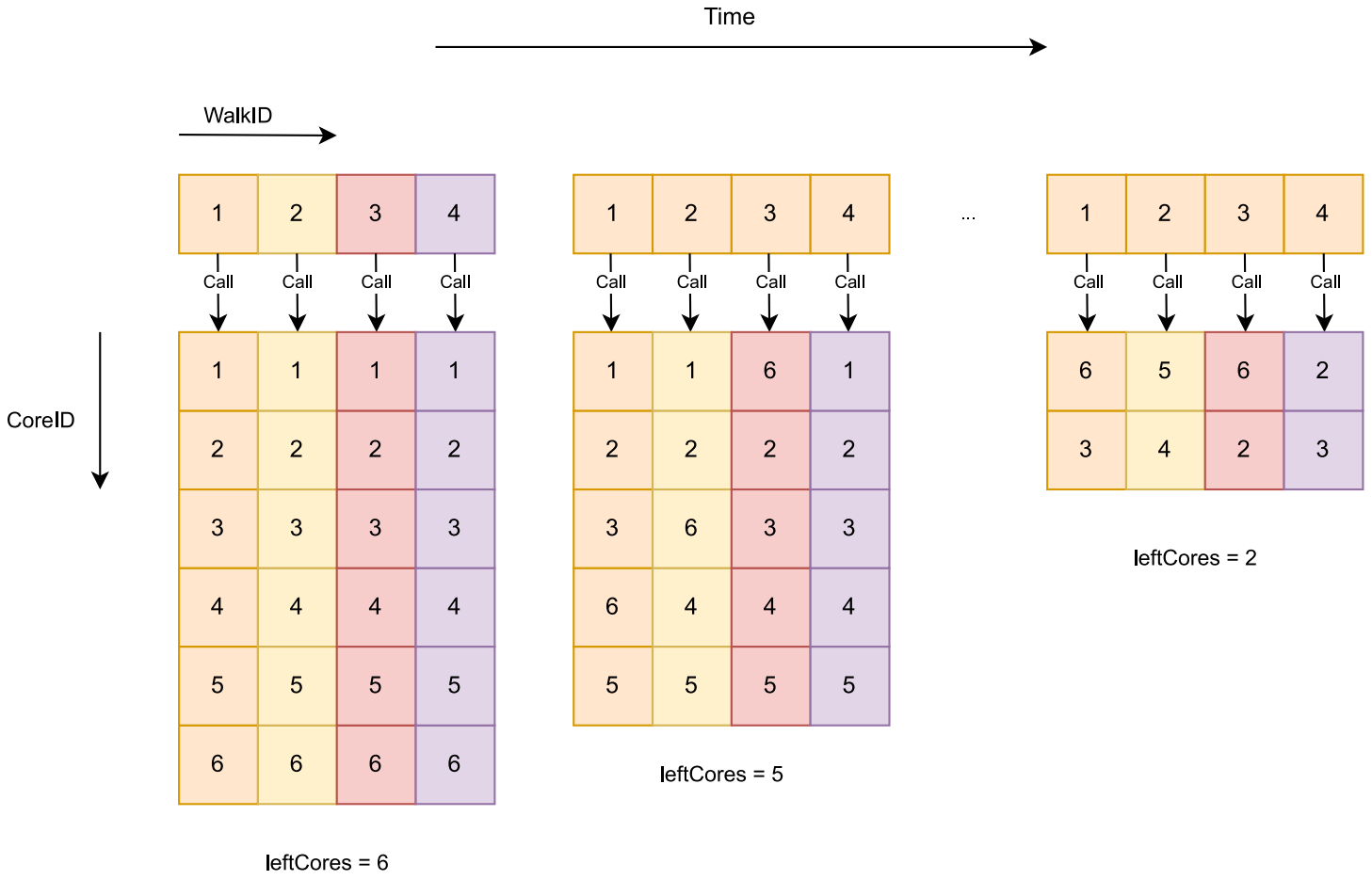



One block accounts for only one walk

Grid synch not needed anymore at Find Min StepT

Implicit removal of Block synch when computing MTTF

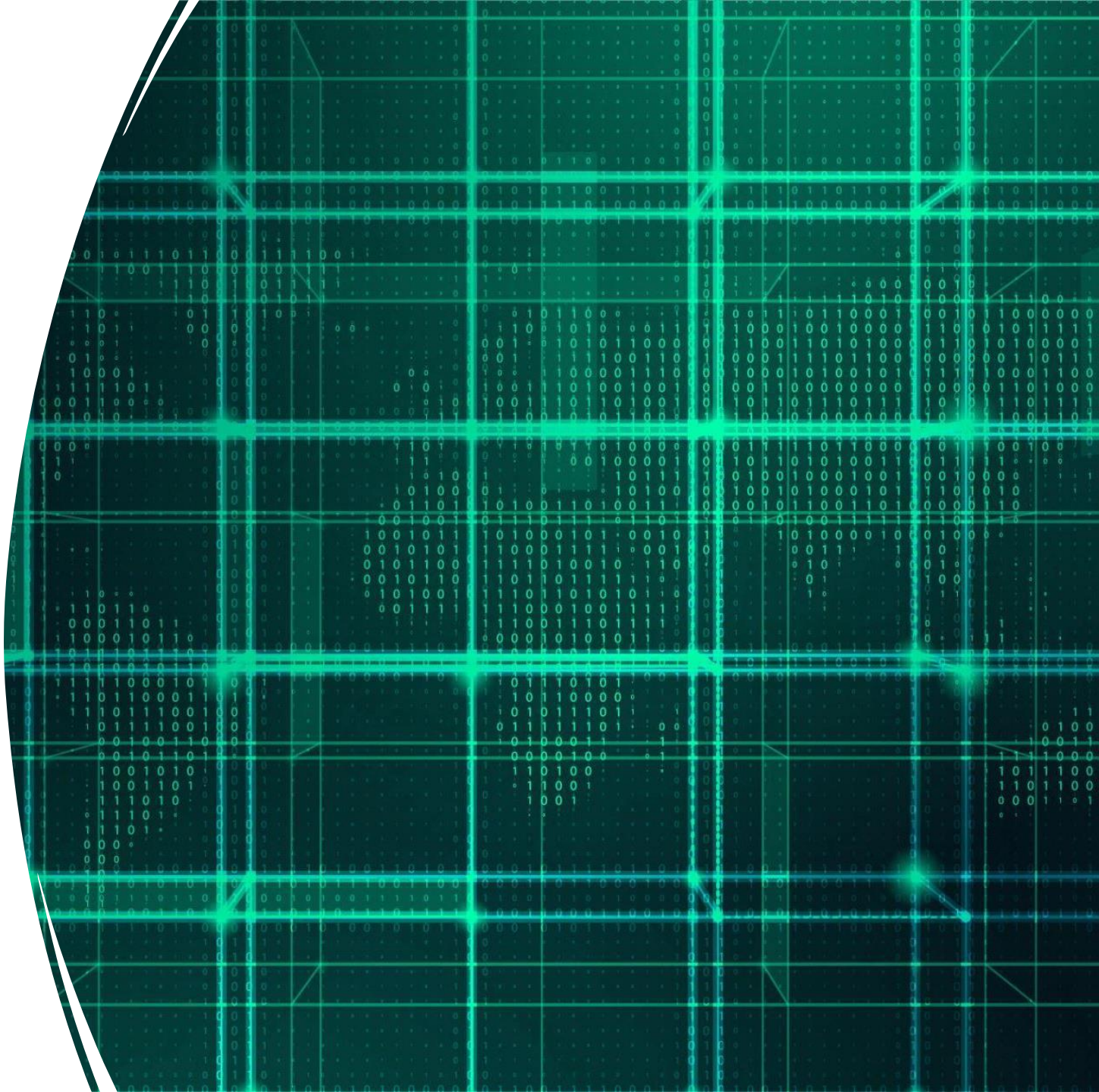
# Dynamic Grid:




At each step, one new kernel is launched for each walk with lower size (account for only alive cores)

# Grid Version Problems:

- Overhead
- Grid Level Synch




Other Optimizations:

# Coalesced Optimization[1]

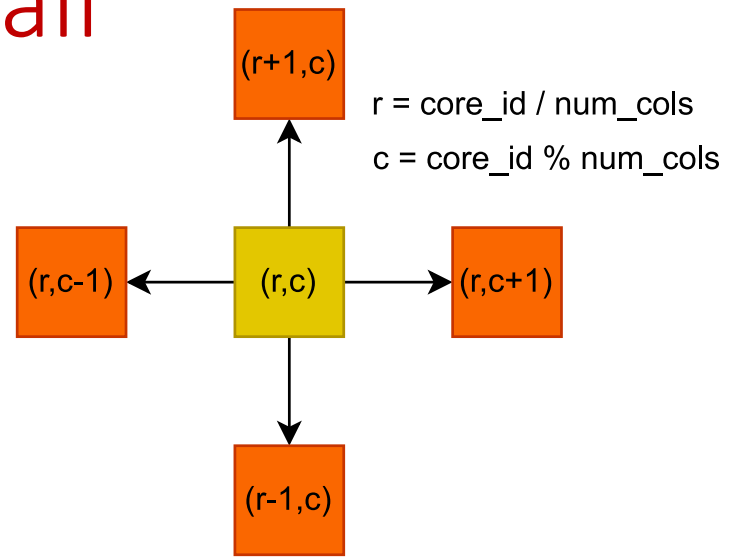
**How can we improve coalesced memory accesses even more?**
**Where was the "bottleneck" of uncoalesceness?**

The Thermal Model is patologically uncoalesced since it

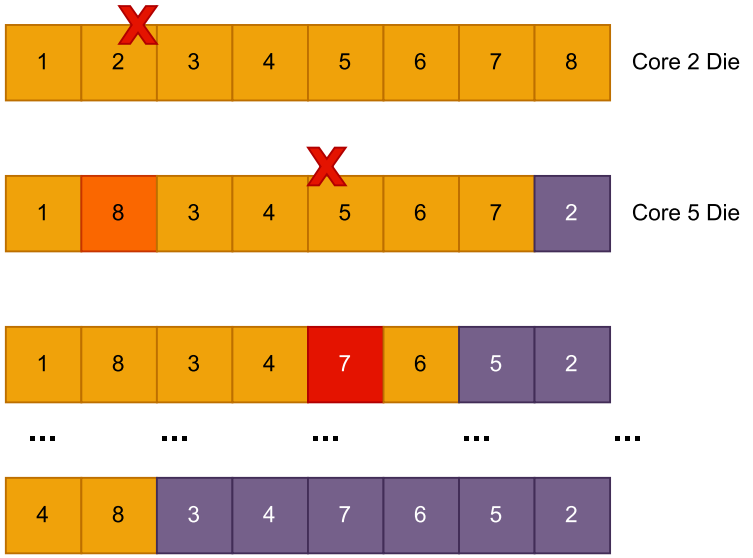
Require information of core neighbours

(Due to the swap of dead cores, those neighbours are NOT in the original positions....)

Recall




Recall



## Coalesced Optimization[2]

**What can we do to solve this uncoalesced accesses?**

We can create a C++ (CoreNeighbours) class to easily manage core neighbourhood.

Then, when a core Die we update it's neighbours neighbourhood!!

In tempModel we use the neighbourhood informations instead of searching them in memory!

(89% of GlobalLoadEfficiency!! -> a 20% improvement)

# Coalesced Optimization[3]

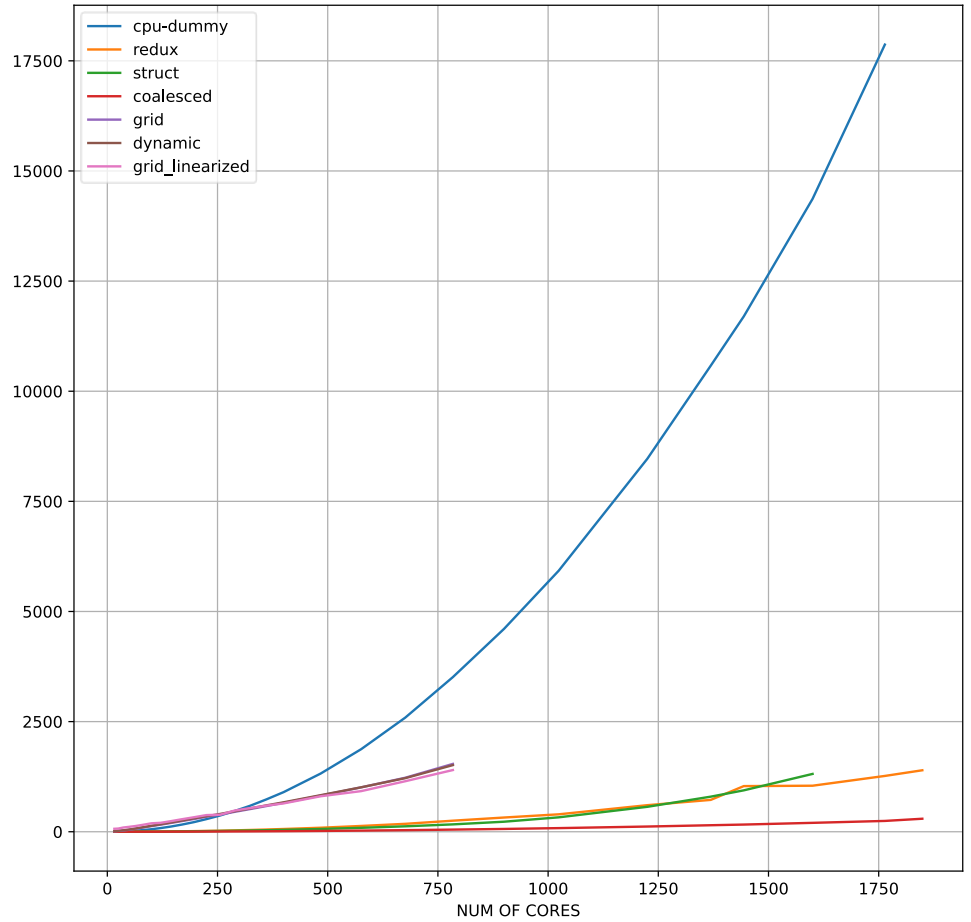
PROBLEM: Excessive usage of memory...

The Simulation go easy out of memory using an array of "Neighbours" class.

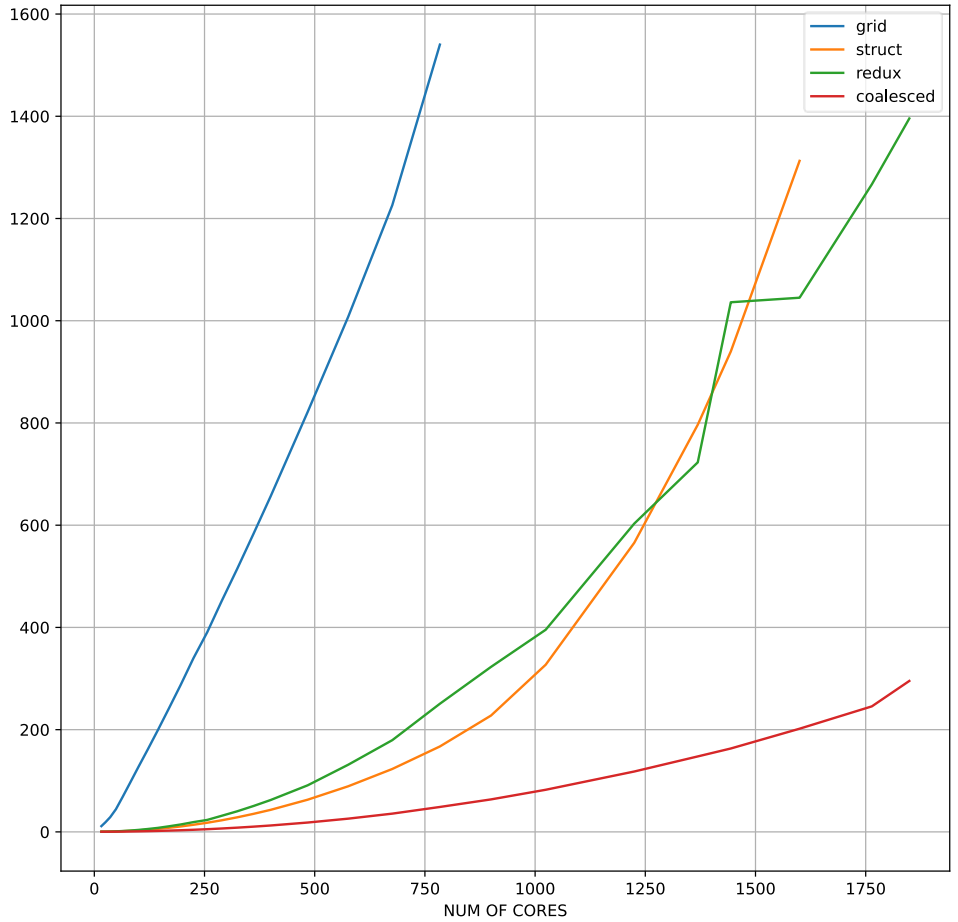
We tried to implement it using a class of arrays but we could not reach same GLD_efficiency results.

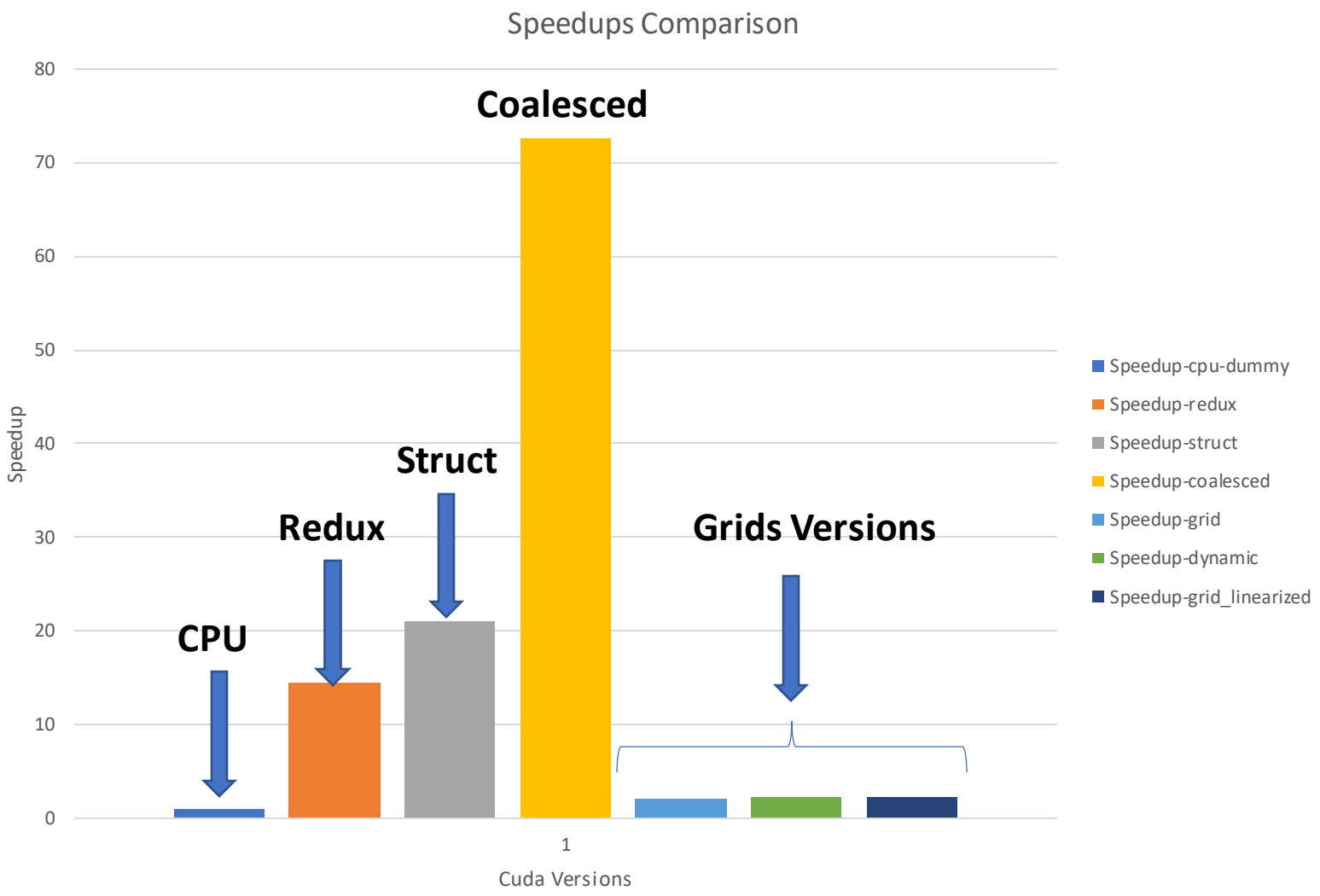

# Benchmarks

# Performances Chart[All]:



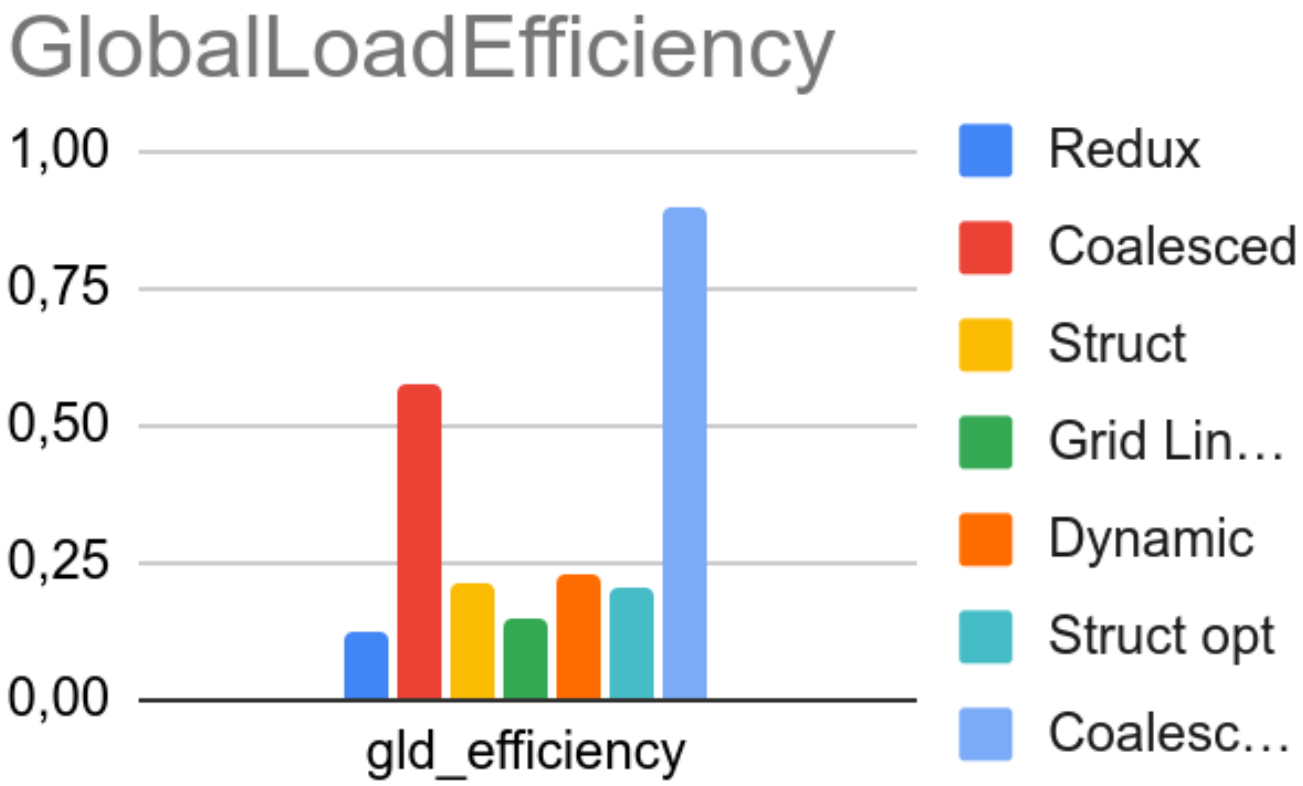
# Performances Chart [Best versions]:





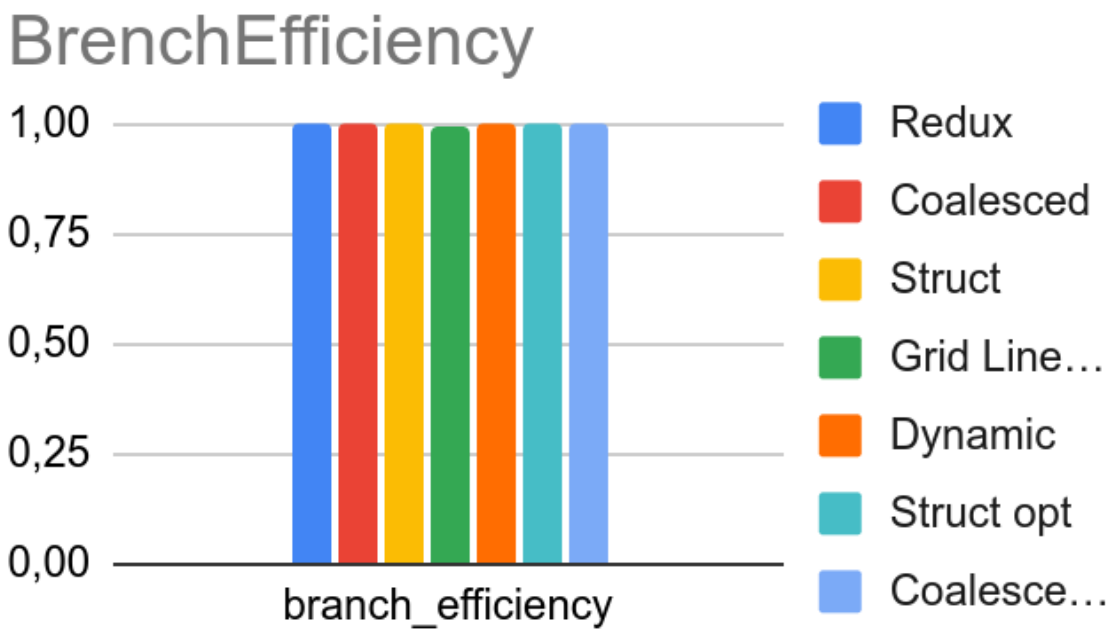

# SpeedUps Table:

# Nvidia Profiling



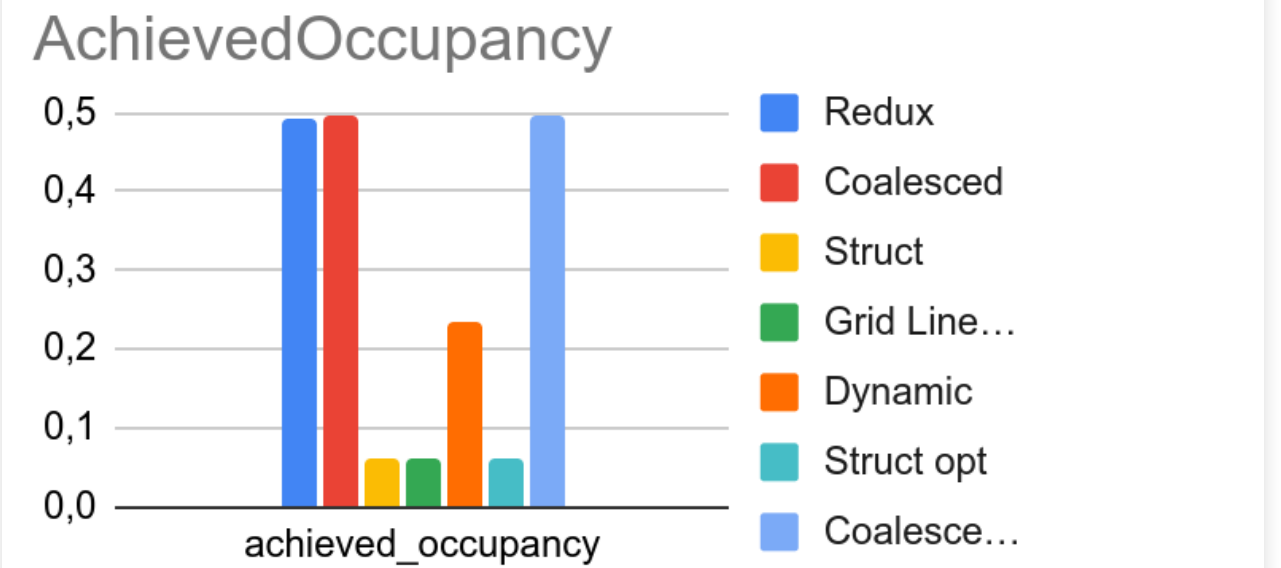

# Global Load Efficiency

# Global Store Efficiency

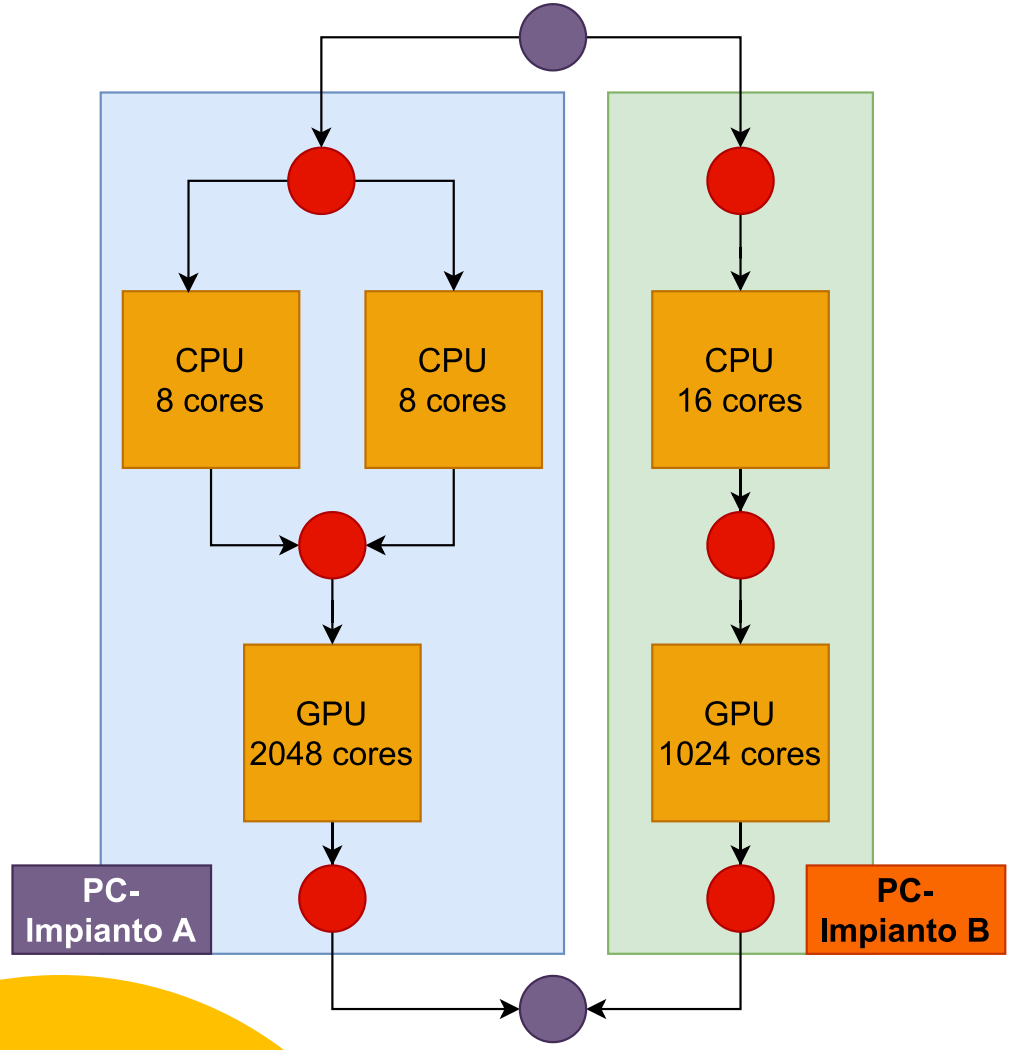



# Branch efficiency

# Achieved Occupancy



# Possible Evolution of the Project




# First Idea:

**What if we convert caliper into a c++ class that represent a single device TTF simulation?**

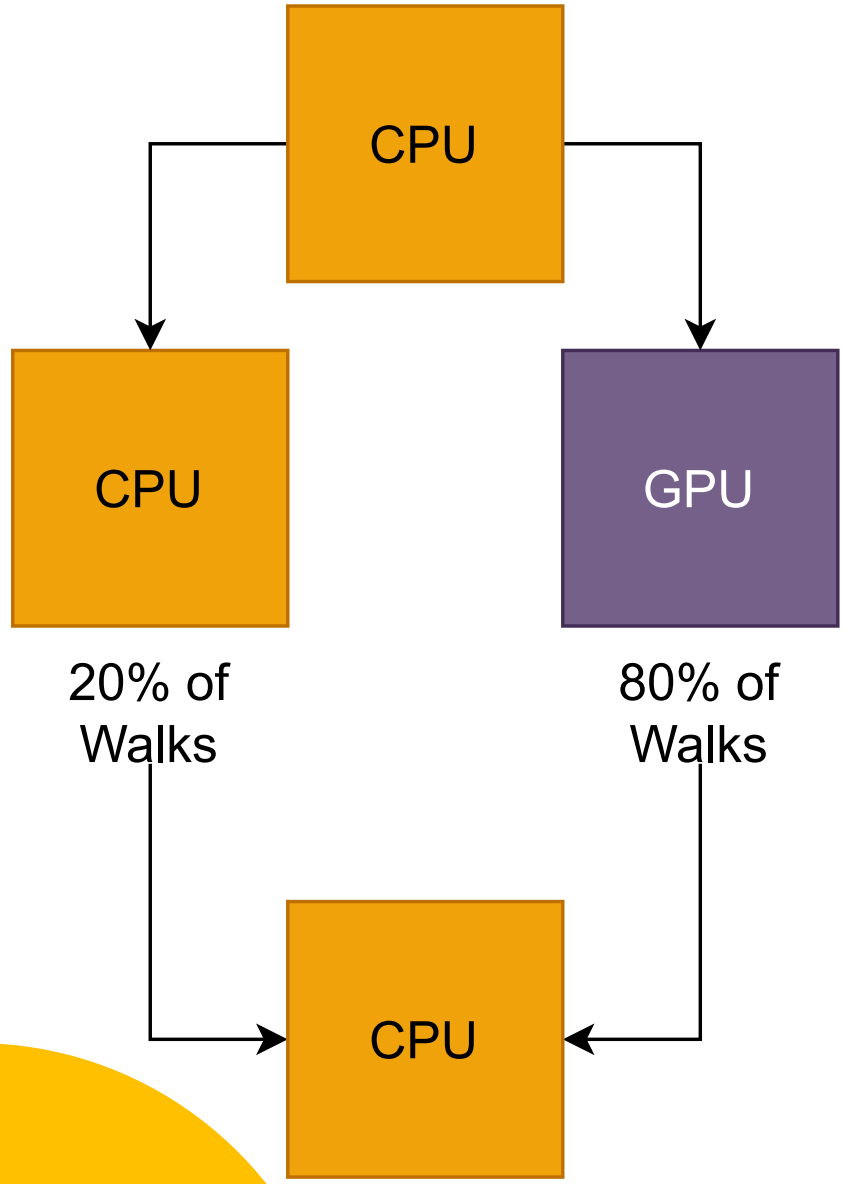
**Example:**

**ImpiantoA** = new CaliperBlock(..configurazione..)

**ImpiantoB** = new CaliperBlock(..configurazione..)

**ImpiantoTotale** = new CaliGraph(ImpiantoA,ImpiantoB);

With this concept caliper would be able to represent heterogeneous systems of serial and parallel components in a graph fashion.




# Second Idea:

**What if we use Cuda streams to parallelize caliper on both CPU and GPU?**

We could for example create a Pthread/MPI/OpenMP version of caliper to run on CPU and then use streams to run it in parallel on CPU and GPU (eg: 20% data on CPU, 80% data on GPU)

# Thanks for the attention