Porting a scientific application to GPU using C++ standard parallelism

Presentation · November 2021

DOI: 10.13140/RG.2.2.27117.92647

UNIVERSITÉ DE GENÈVE
FACULTY OF SCIENCE

NVIDIA GTC 21
November 2021

Porting a scientific application to GPU using C++ standard parallelism

Presenter: ***Jonas Latt*** *- Université de Genève (UNIGE), Switzerland*

Contributors: *Christophe Coreixas (UNIGE)*
Francesco Marson (UNIGE)
Karthik Thyagarajan (UNIGE)
Jose Pedro de Santana Neto (UNIGE)
Sriharan BS (IITM)
Gonzalo Brito (NVIDIA)

Outline

Part I. Introduction

Everything you need to know to follow this presentation: C++ parallel algorithms, lattice Boltzmann method, the Palabos project.

Part II. Design: From Object-Oriented to Data-Oriented

For the most part, the code can be executed on GPU as is. But for best performance, the overall architecture must be migrated from object-oriented to data-oriented.

Part III. In-depth discussion: performance improvements

Some neat tricks to get even better performance with little effort.

UNIVERSITÉ DE GENÈVE
FACULTY OF SCIENCE

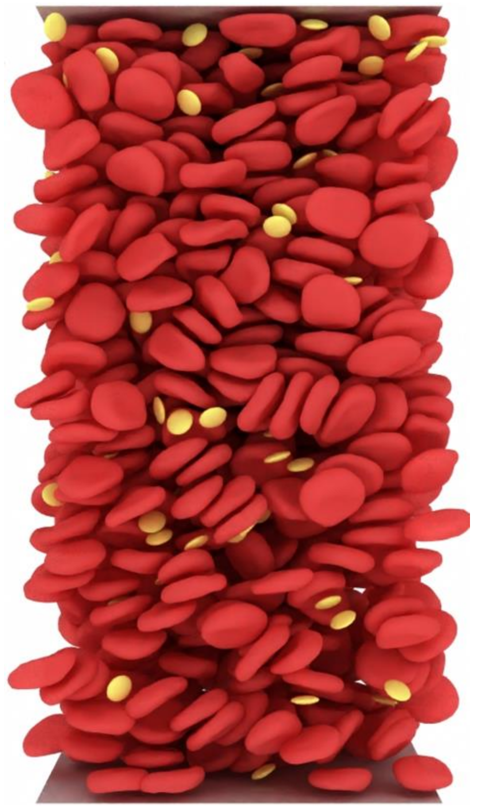
Part I

INTRODUCTION

The Palabos software library

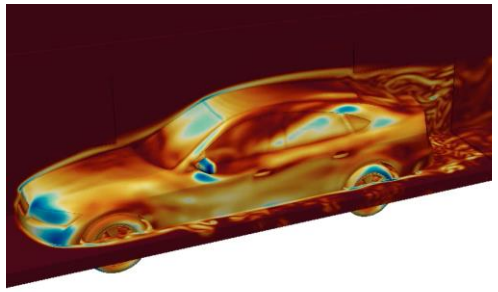
UNIVERSITÉ DE GENÈVE
FACULTY OF SCIENCE

Fluid-Structure

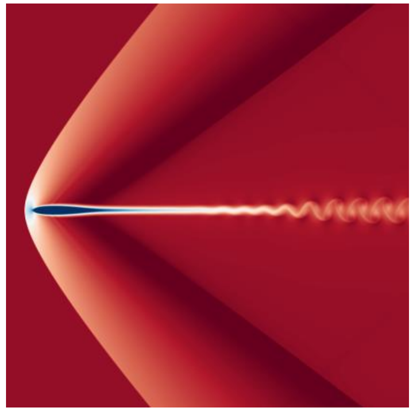


Multi-Phase

Aerodynamics



Supersonic



A software library for complex fluid flow simulations

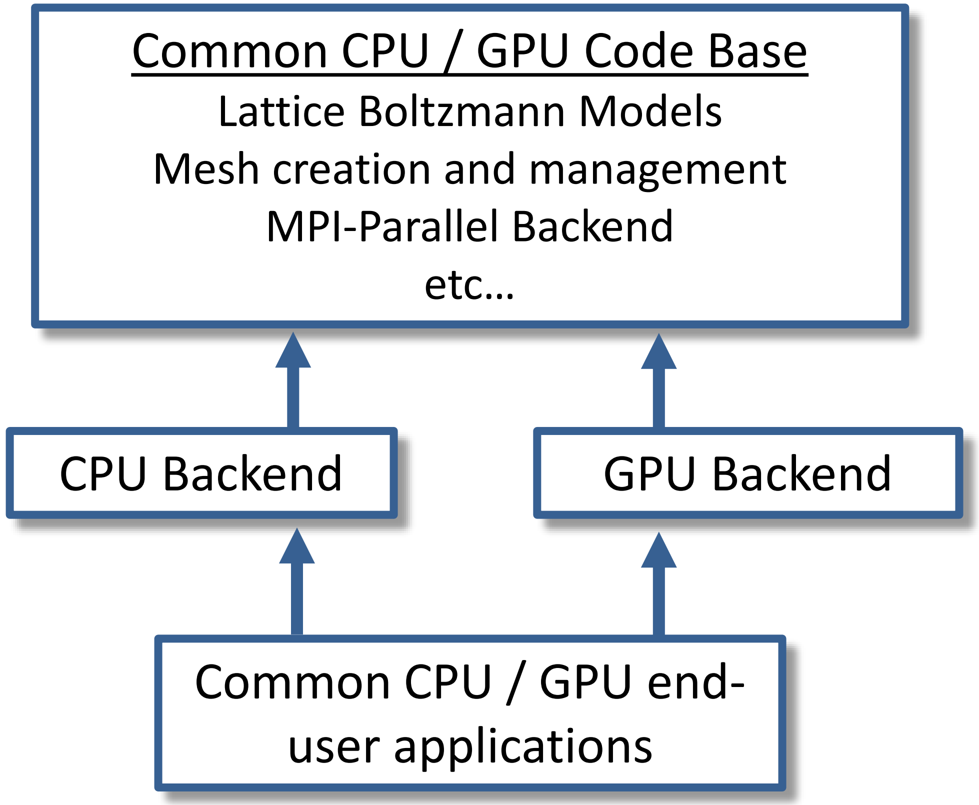
Numerical method: Lattice Boltzmann

Massively parallel (originally CPU)

UNIVERSITÉ DE GENÈVE
FACULTY OF SCIENCE

A GPU backend for Palabos

Modifications to the existing CPU code base should be minimal


Common CPU / GPU Code Base
Lattice Boltzmann Models
Mesh creation and management
MPI-Parallel Backend
etc…
CPU Backend
GPU Backend
Common CPU / GPU end-user applications

Original end-user application

```
MultiBlockLattice3D<T, DESCRIPTOR> lattice(geometry);
defineDynamics(lattice, boundingBox,
               new TRTdynamics<T, DESCRIPTOR>(omega));
defineDynamics(lattice, geometry, boundingBox,
               new BounceBack<T, DESCRIPTOR>, FLAG);
…
lattice.collideAndStream();
```

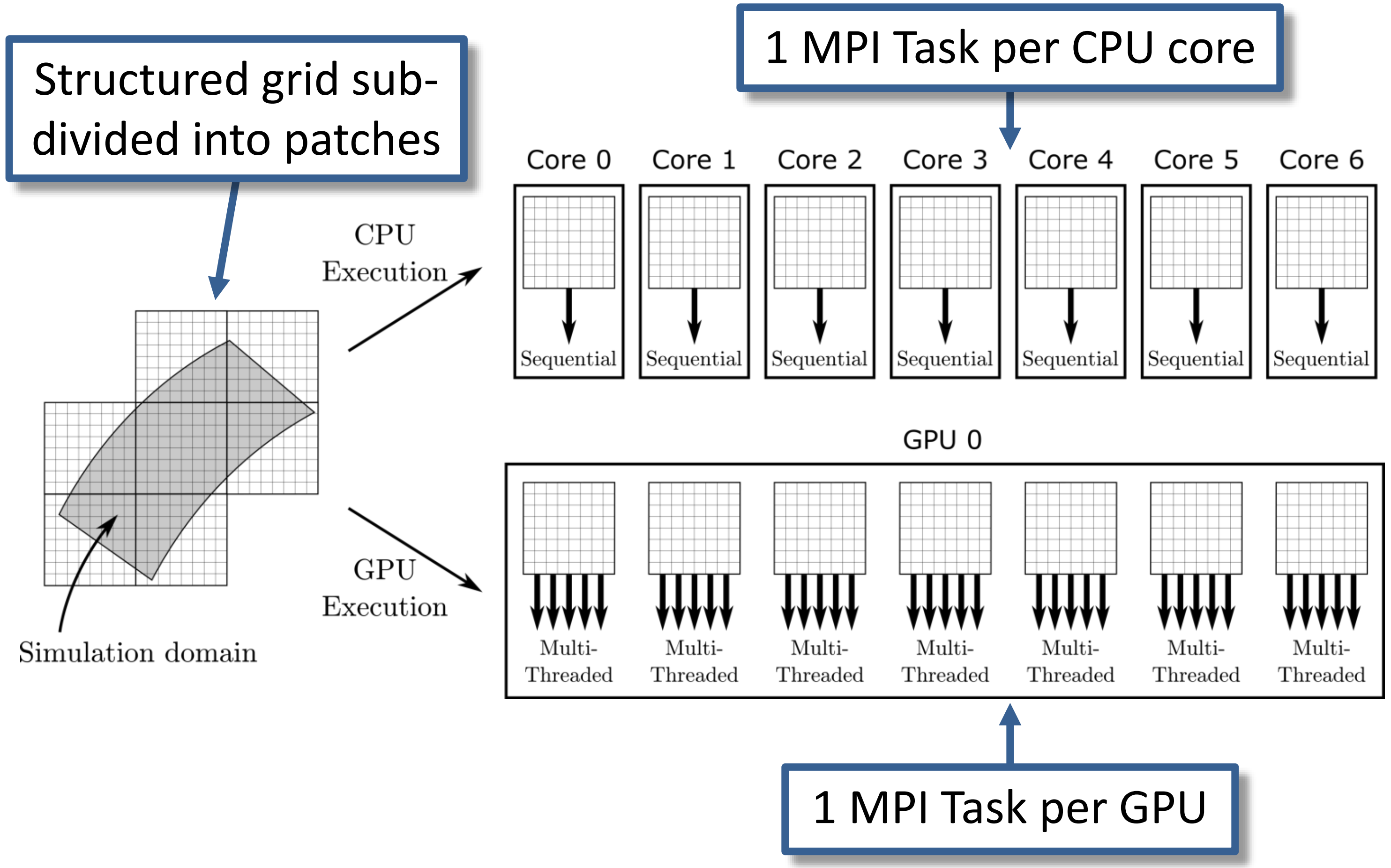
Application with GPU backend

```
MultiBlockLattice3D<T, DESCRIPTOR> lattice(geometry);
defineDynamics(lattice, boundingBox,
               new TRTdynamics<T, DESCRIPTOR>(omega));
defineDynamics(lattice, geometry, boundingBox,
               new BounceBack<T, DESCRIPTOR>, FLAG);
AcceleratedLattice3D<T, DESCRIPTOR> accLattice(lattice);
…
// Lattice.collideAndStream();
accLattice.collideAndStream();
```

New structure runs on GPU

A GPU backend for Palabos

On GPU, multi-threading must be more fine-grained



CPU
Patch-level parallelism
Sequential within 1 MPI task

GPU
Node-level parallelism
Multi-threaded

Reminder: C++ Standard Parallelism

UNIVERSITÉ DE GENÈVE
FACULTY OF SCIENCE

Since C++17, C++ includes parallel language features (no extension, no library needed)

```
vector<double> v = { 1, 2, 3, 4 }, w(4);
transform( execution::par_unseq, begin(v), end(v), begin(w),
           [](double const& element) { return 2. * element; } );
// w is {2, 4, 6, 8}
```

Read from v, write into w

Lambda function: defines the element-wise operation applied to v.

Execution policy

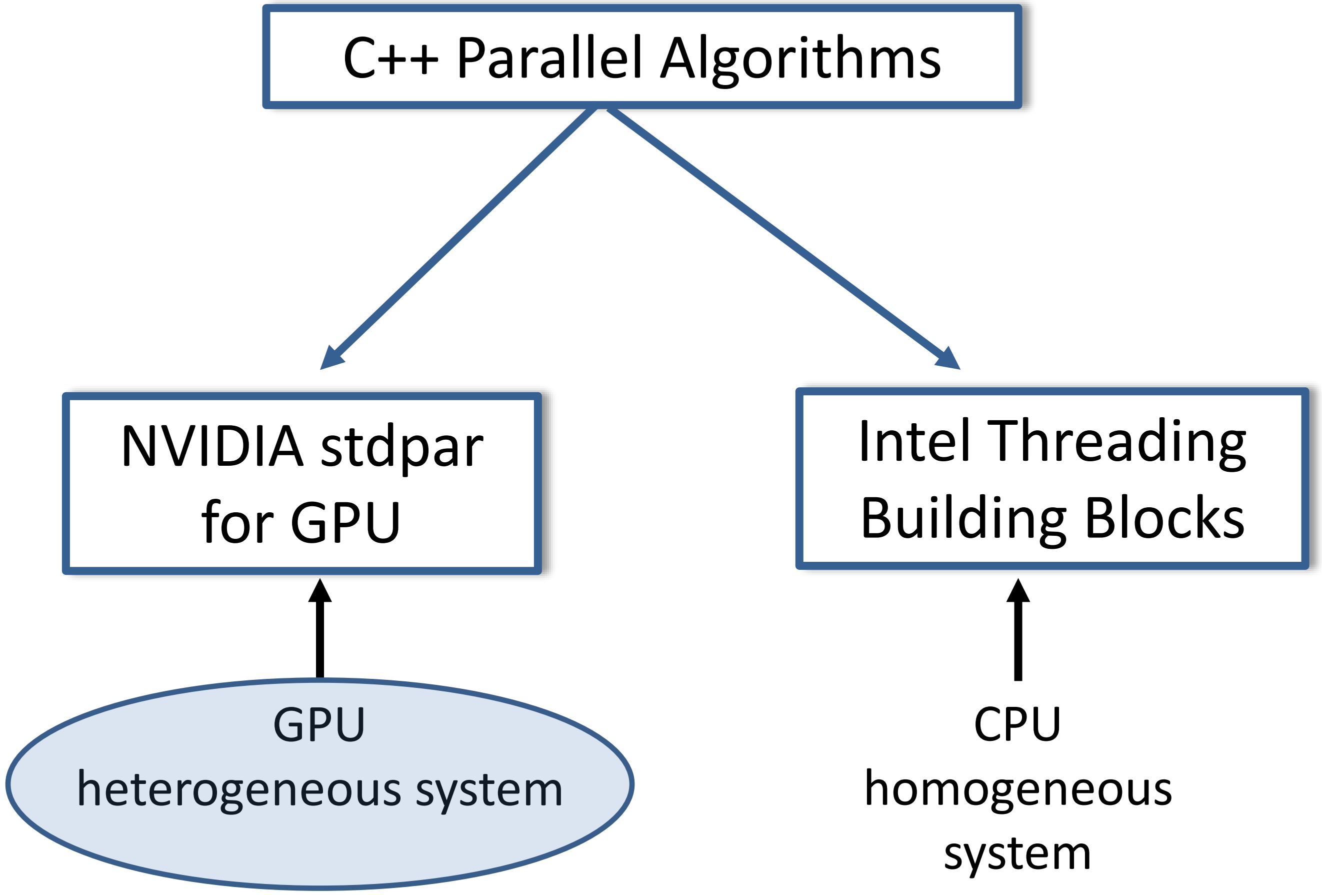
```
nvc++ -stdpar -o program program.cpp
```

Available implementations

The code is portable and gets accelerated for multiple types of parallel platforms

Common formalism

Hardware-specific implementation



CUDA Unified Memory

Parallel STL execution on heterogeneous platform is possible thanks to managed memory model

```
vector<double> v = { 0., 1., 2., 3., 4., 5. };
for_each(begin(v), end(v), [](double& x) { x = sin(x / N * M_PI); });

for_each( execution::par_unseq, begin(v), end(v),
          [](double& x) { x = sqrt(x);  } );
```

Executed on host

Executed on device

Automatic data transfer

This model encourages hybrid CPU / GPU code and porting a code to GPU progressively.

Further Resources

UNIVERSITÉ DE GENÈVE
FACULTY OF SCIENCE

Use of standard language parallelism for GPU programming

GTC21 Spring session:
Fluid Dynamics on GPUs with C++ Parallel Algorithms
https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s32076/

Current GTC21 November session, on Thursday:
Accelerated Computing with Standard C++, Python, and Fortran [A31181]

Use of C++ parallel algorithms for lattice Boltzmann applications

Open-source code STLBM (reusable code-snippets):
https://www.gitlab.com/UnigeHPFS/stlbm

From CPU to GPU in 80 days

Case Study: The Palabos fluid solver was ported to GPU in less than three months this summer

https://palabos.unige.ch/community/cpu-gpu-80-days

Project: a GPU-port of the Palabos library (around half a million lines of code)

Goal: A GPU backend of Palabos with
- Same ease of maintenance as provided by the CPU backend
- Same ease of use as provided by the CPU backend

Replace loops by for_each: is this it?

```
for (int iX = 0; iX < nx; ++iX) {
    for (int iY = 0; iY < ny; ++iY) {
        for (int iZ = 0; iZ < nz; ++iZ) {
            cell(iX, iY, iZ).collideAndStream();
        }
    }
}
```

```
for_each(execution::par_unseq,
         begin(pop), end(pop), [](Cell& cell)
{
    cell.collideAndStream();
} );
```

Is this it? Code runs on GPU after simple substitution ?

Not entirely…

This presentation highlights some points to keep in mind.

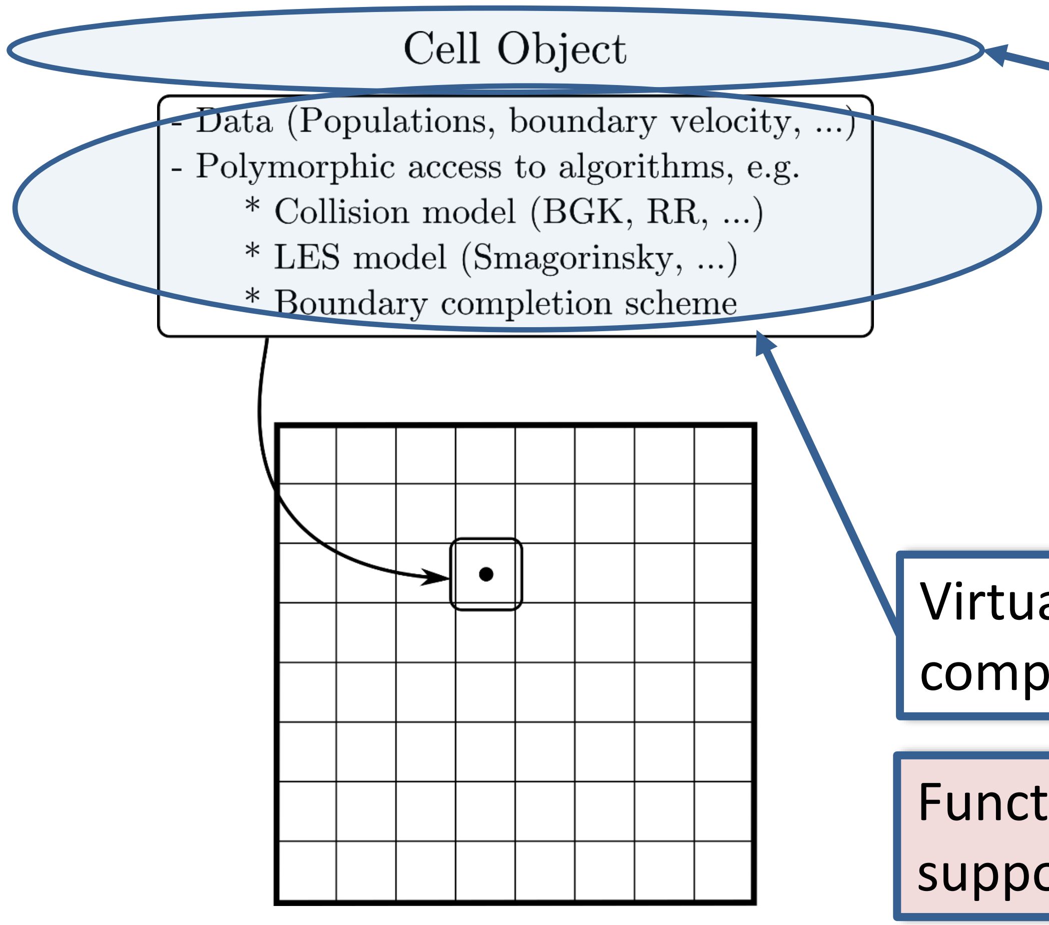
UNIVERSITÉ DE GENÈVE
FACULTY OF SCIENCE

Part II

DESIGN: FROM OBJECT-ORIENTED TO DATA-ORIENTED

Palabos: Object-oriented approach

Polymorphism allows every grid node to implement different physical / numerical model


Cell Object
- Data (Populations, boundary velocity, ...)
- Polymorphic access to algorithms, e.g.
* Collision model (BGK, RR, ...)
* LES model (Smagorinsky, ...)
* Boundary completion scheme

Cell objects contain local data, typically 19 floats, the "populations".

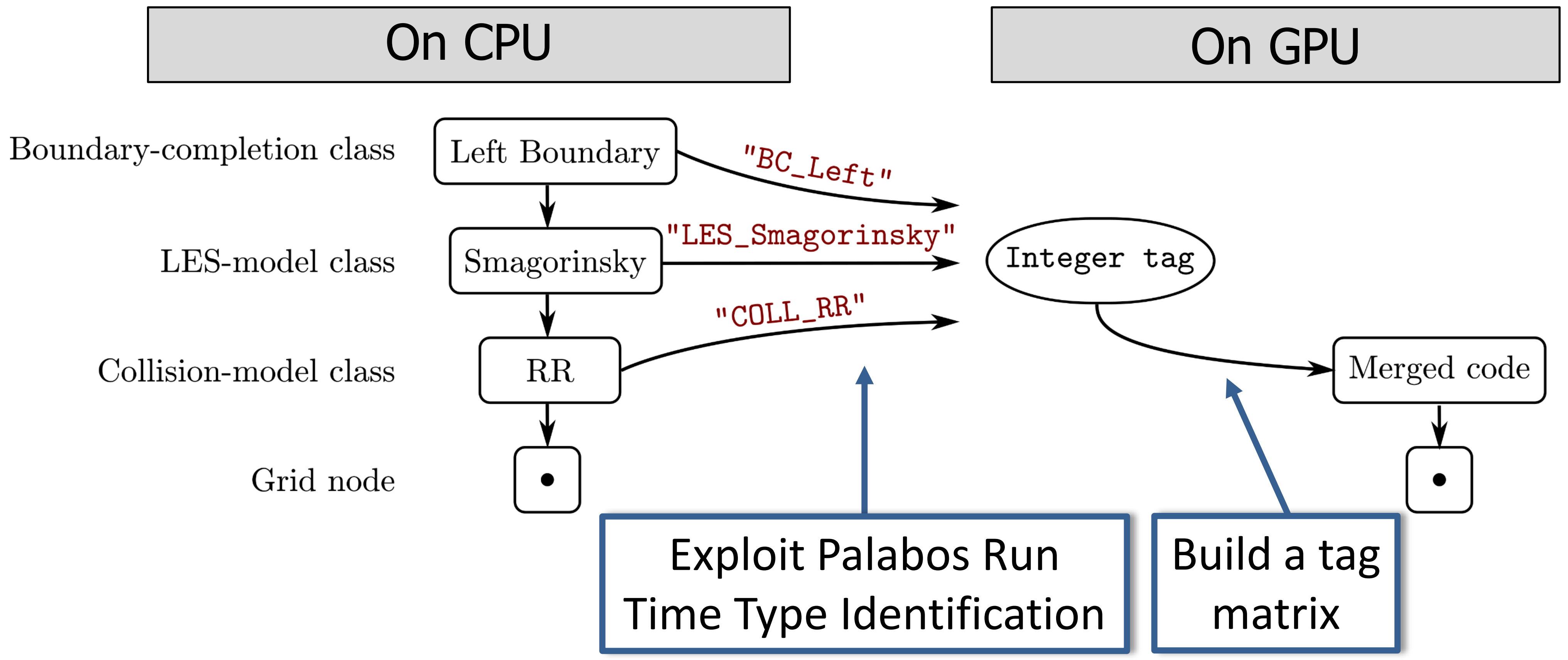
GPUs don't like the resulting memory layout.

Virtual function calls to different model components.

Function-pointer call mechanism is not supported.

Polymorphic objects → Tag matrix

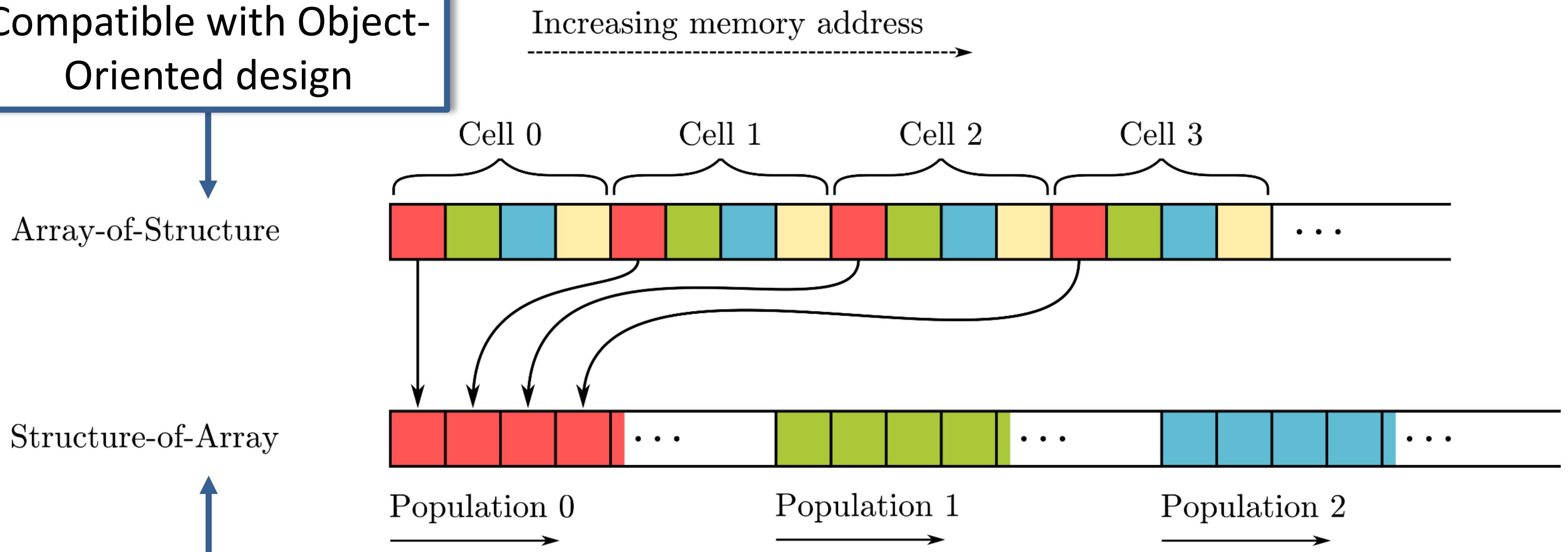
The tag matrix suits the GPU better and can be generated automatically



Array-of-structure → Structure-of-array

Data alignment in memory must be optimized for the way a node interacts with its neighbors

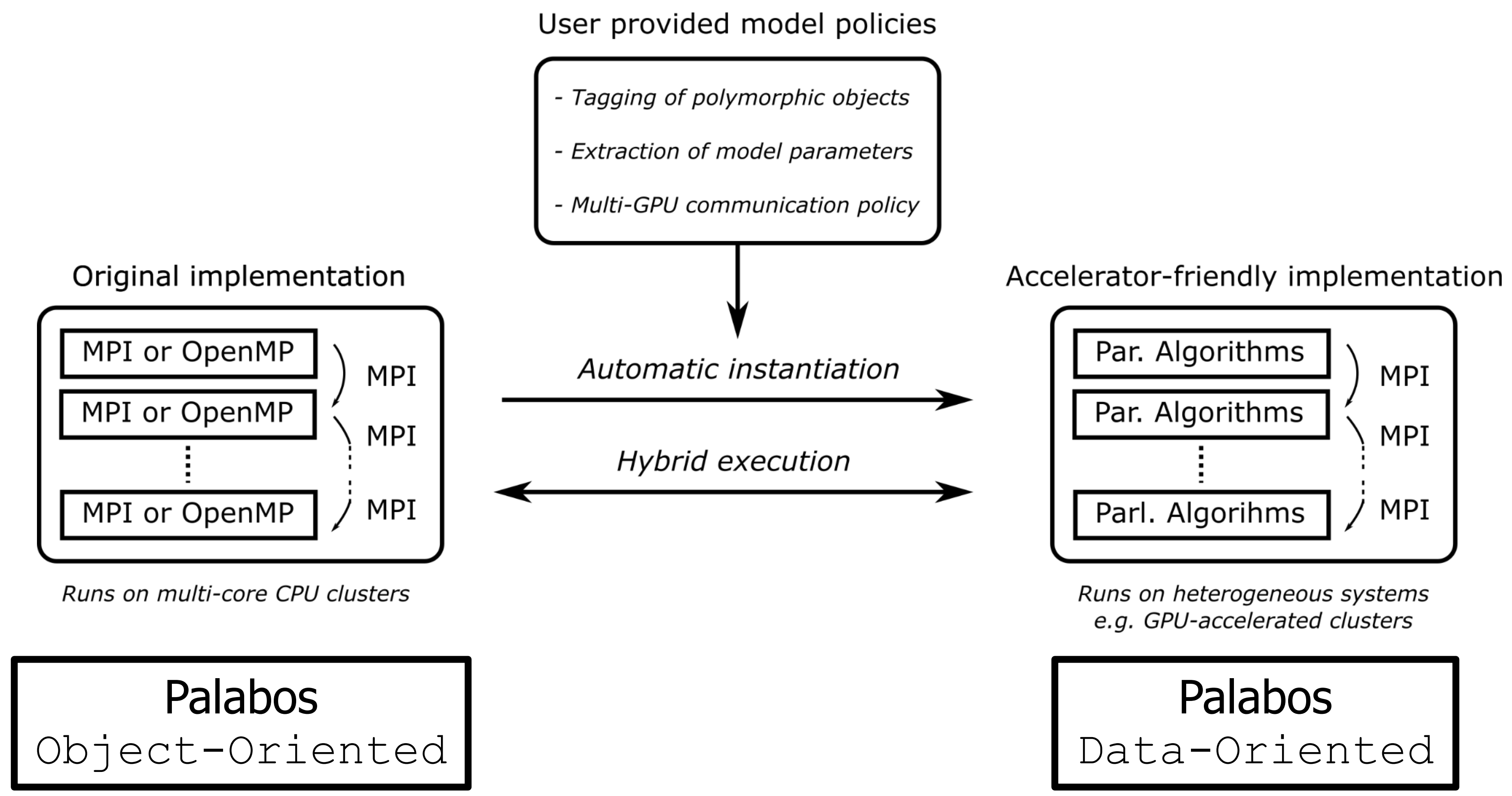
Compatible with Object-Oriented design



Runs efficiently on GPU

Object-oriented → Data-oriented

A policy class, provided for each model, allows automatic instantiation of data-oriented structure


User provided model policies
- Tagging of polymorphic objects
- Extraction of model parameters
- Multi-GPU communication policy
Original implementation
MPI or OpenMP
MPI or OpenMP
MPI or OpenMP
MPI
MPI
MPI
Automatic instantiation
Hybrid execution
Accelerator-friendly implementation
Par. Algorithms
Par. Algorithms
Parl. Algorihms
MPI
MPI
MPI
Runs on multi-core CPU clusters
Runs on heterogeneous systems e.g. GPU-accelerated clusters
Palabos Object-Oriented
Palabos Data-Oriented

Benchmark hardware: 2 desktop computers

Computer 1
(Sits on my desk)
Intel Xeon Gold 6240R CPU
48 cores, 2 sockets

Computer 2
(On a desk next door)
NVIDIA RTX 3090 GPU
Ampere Architecture

All benchmarks executed in single-precision

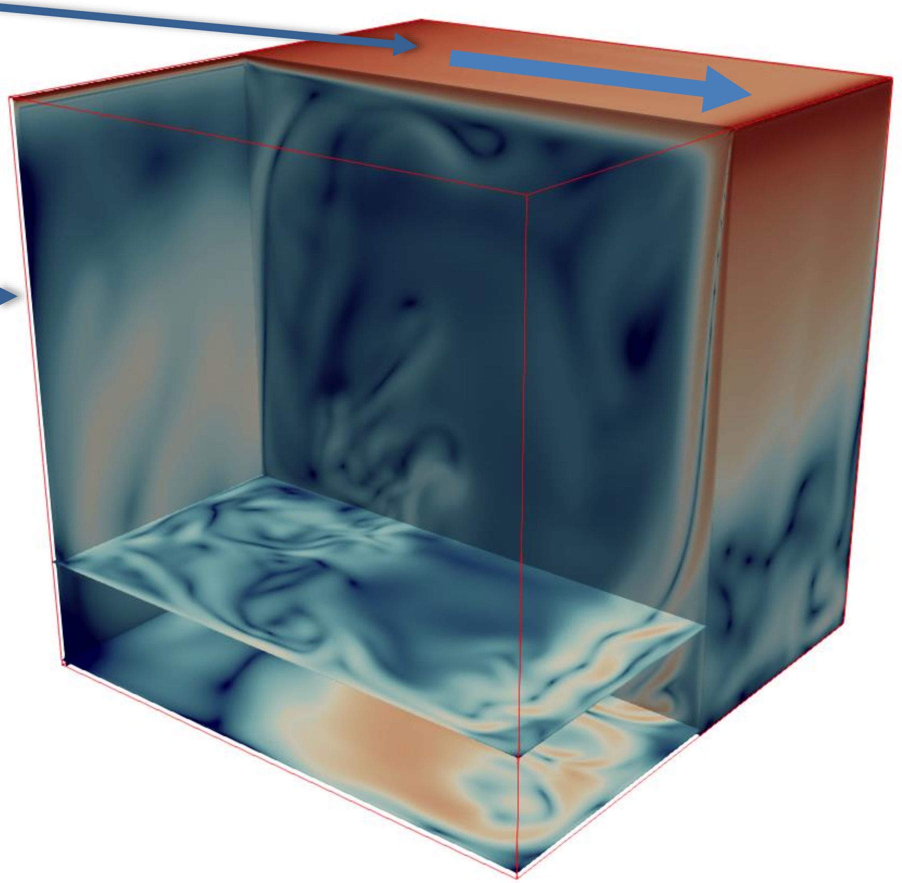
First validation: lid-driven cavity

UNIVERSITÉ DE GENÈVE
FACULTY OF SCIENCE

With a simple geometry, this standard benchmark assesses the raw performance of the code

Moving top lid: constant velocity from left to right

Cubic box with no-slip walls



Collision: Recursive-Regularized
Lattice: D3Q19
Resolution: 420x420x420
Floats: Single precision

Performance: lid-driven cavity

UNIVERSITÉ DE GENÈVE
FACULTY OF SCIENCE

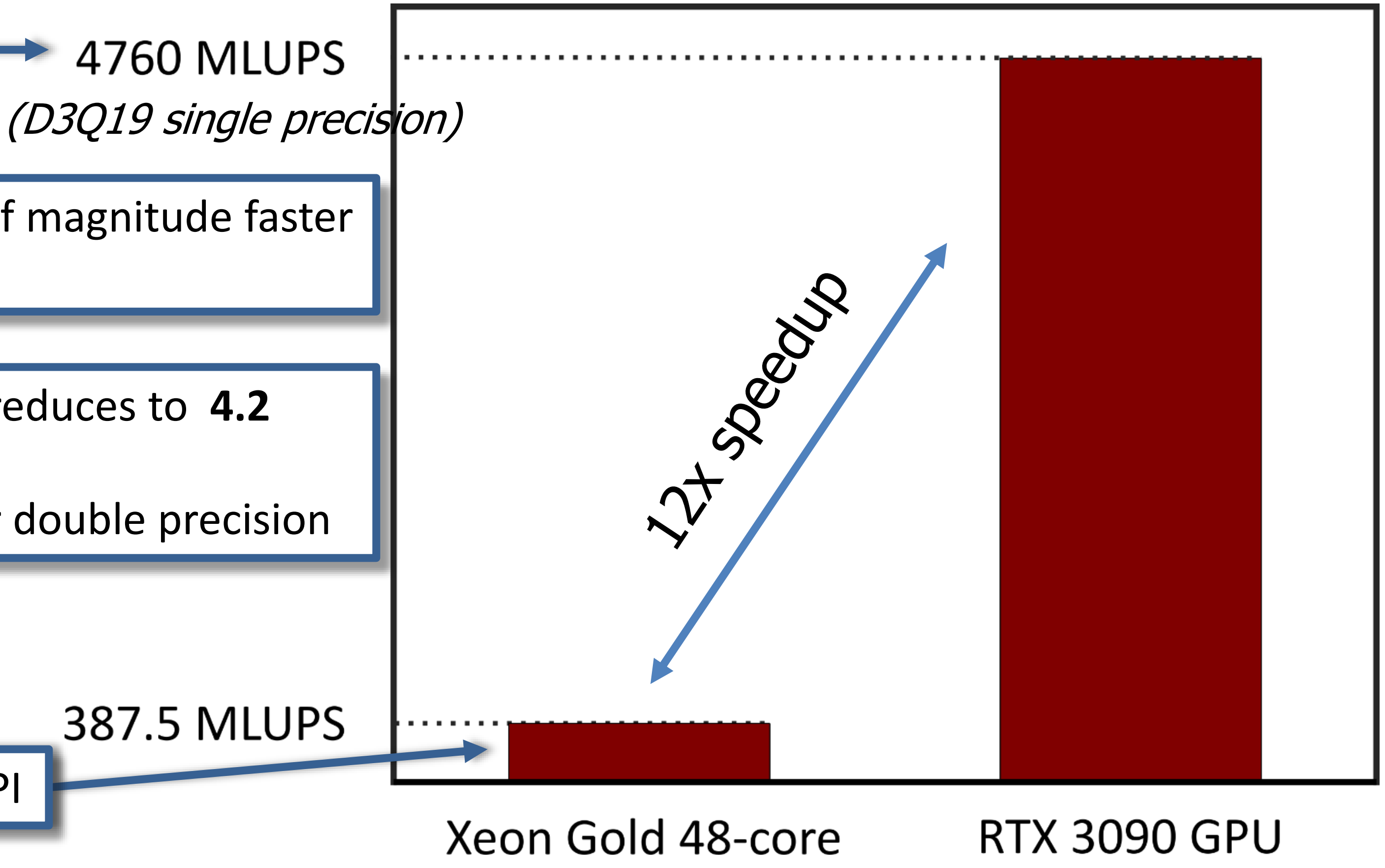
The same end-user application is executed once with the CPU and once with the GPU backend

MLUPS: Million grid-node updates per second

GPU is more than an order of magnitude faster (but only in single precision)

Double-precision: speedup reduces to **4.2**
- GPU is memory bound
- RTX 3090 not optimized for double precision

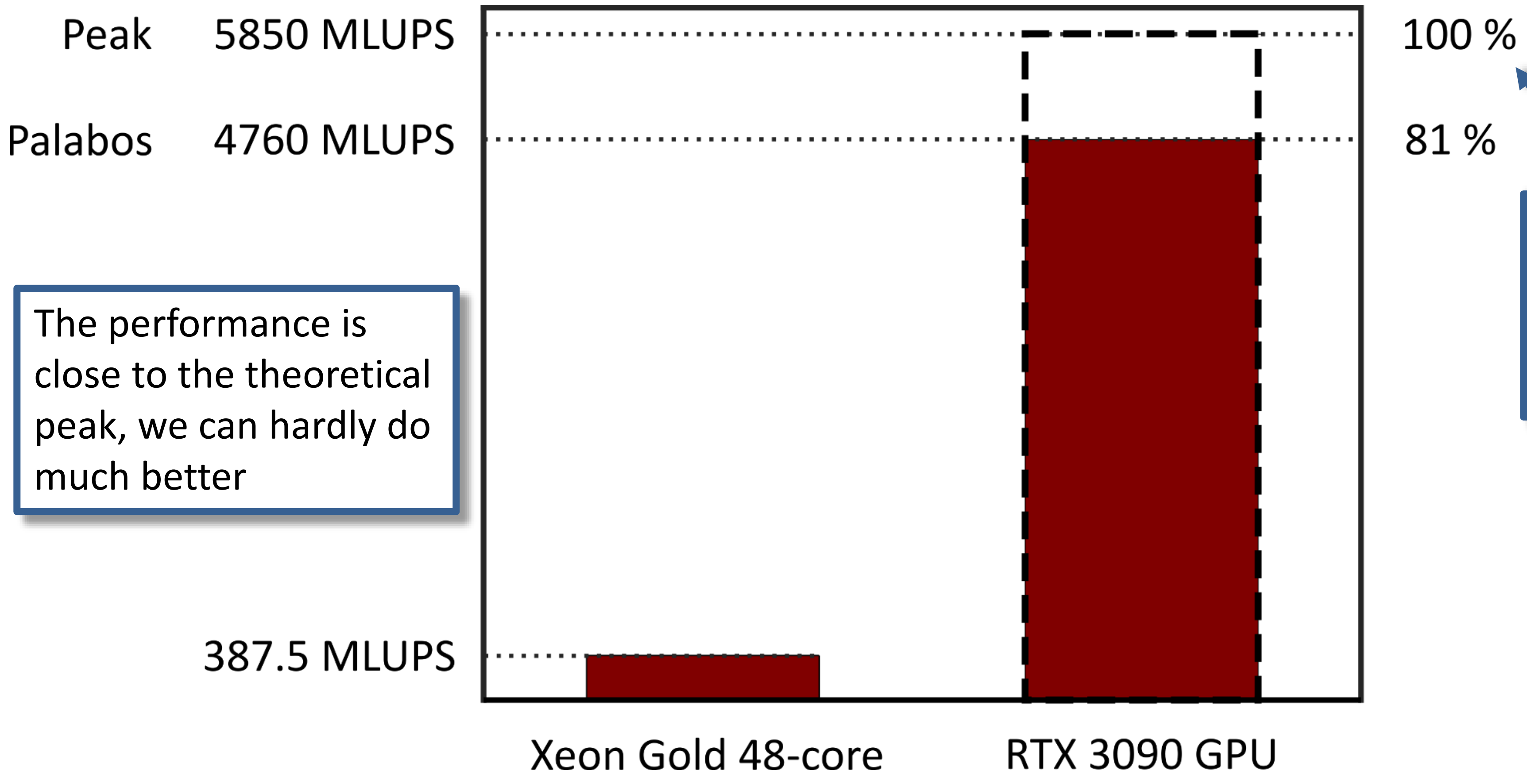
Uses all 48 cores through MPI



Comparison against peak performance

UNIVERSITÉ DE GENÈVE
FACULTY OF SCIENCE

Because the problem is memory bound on GPU, peak performance is dictated by memory bandwidth



The performance is close to the theoretical peak, we can hardly do much better

Maximum performance given the 3090's memory bandwidth, with 20 floats per node.

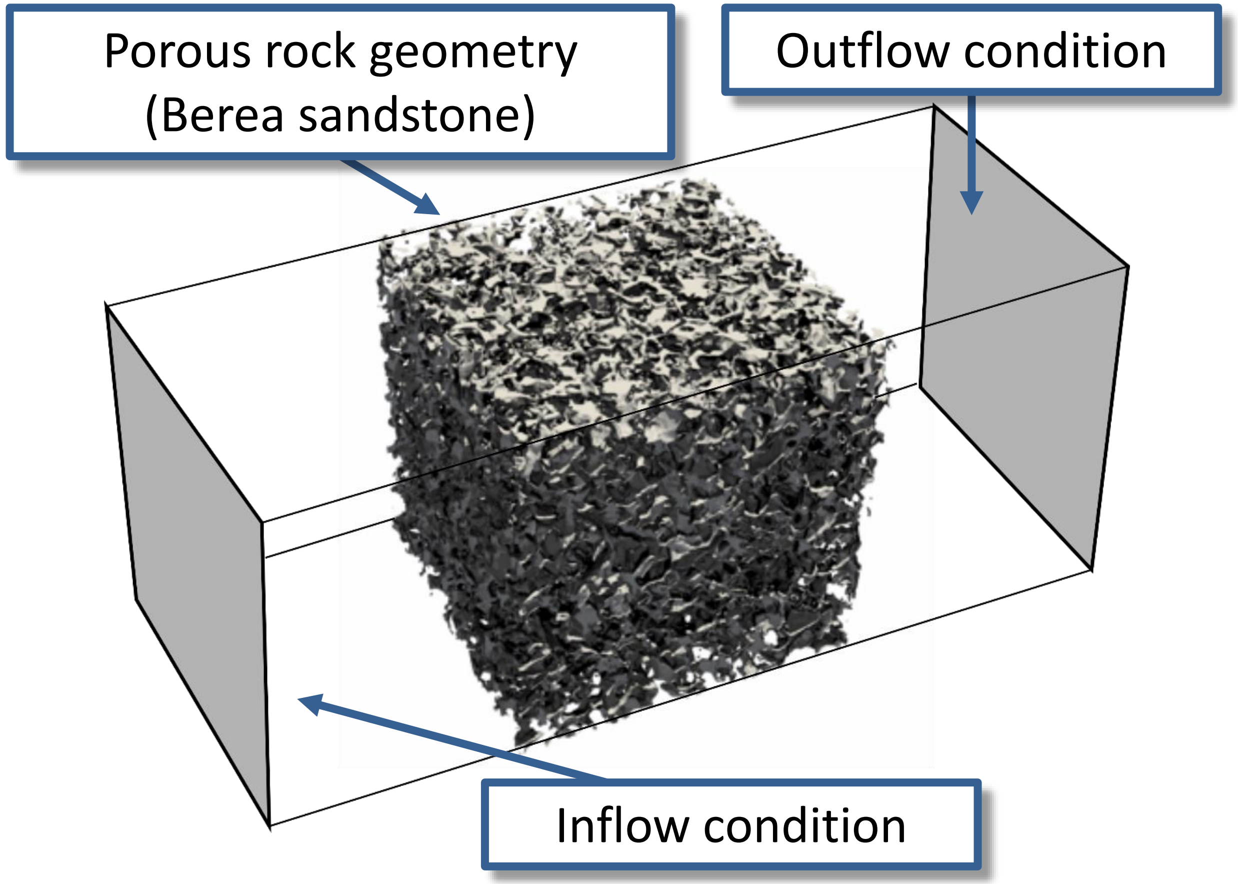
2nd validation: Porous media flow

UNIVERSITÉ DE GENÈVE
FACULTY OF SCIENCE

A fully resolved flow through a porous rock, requires handling of a very complex geometry

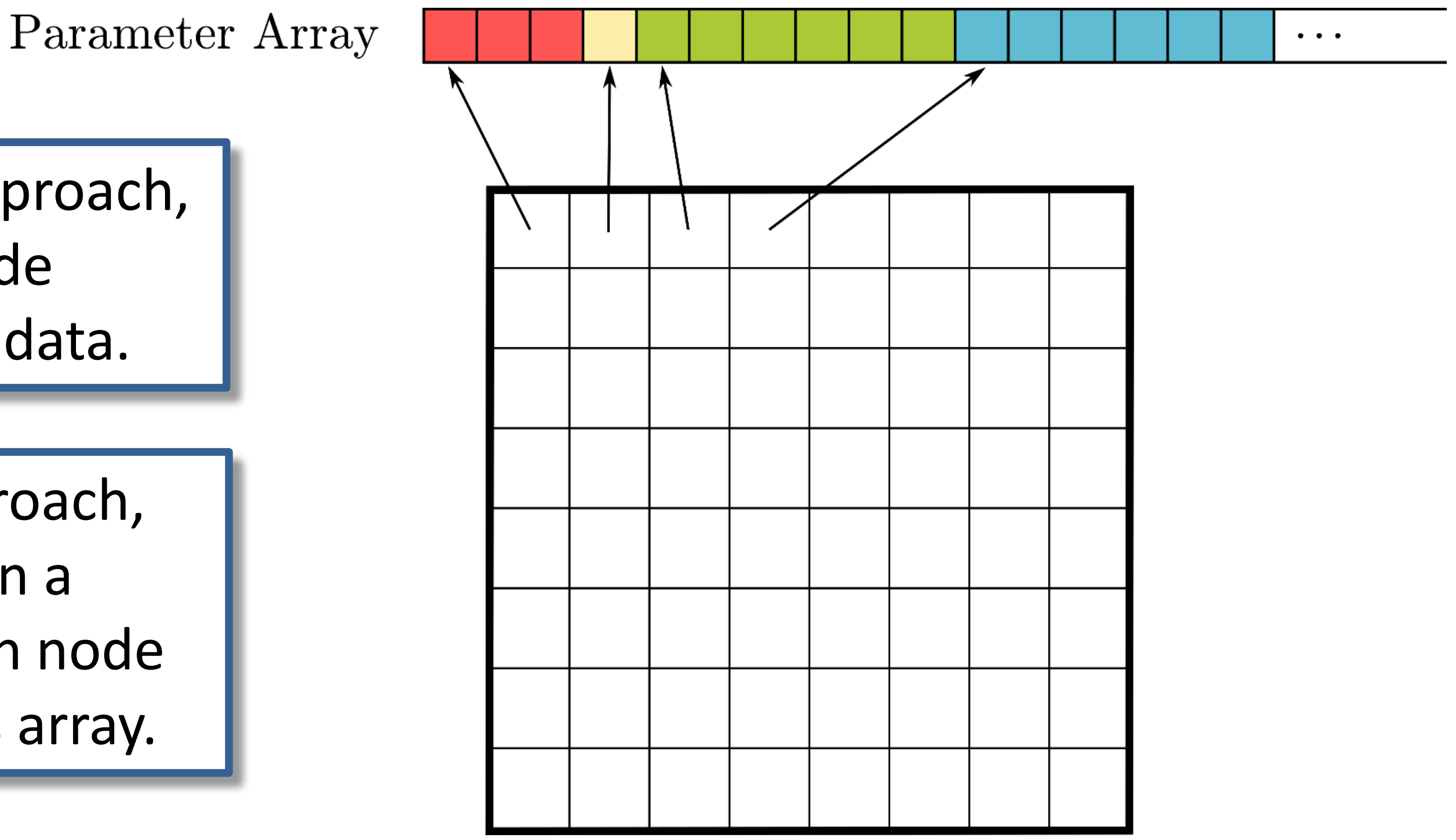
Handling of geometry: use of different collision model on solid and fluid nodes

In the data-oriented code, how do we store extra data required by specific nodes ? Example: flow on inlet condition



Separate array for extra grid-node data

Some nodes require extra data, such as the parameters of a boundary condition



In object-oriented approach, no problem: each node freely allocates extra data.

In data-oriented approach, extra data is packed in a parameter array. Each node has an index into this array.

Performance: Porous media flow

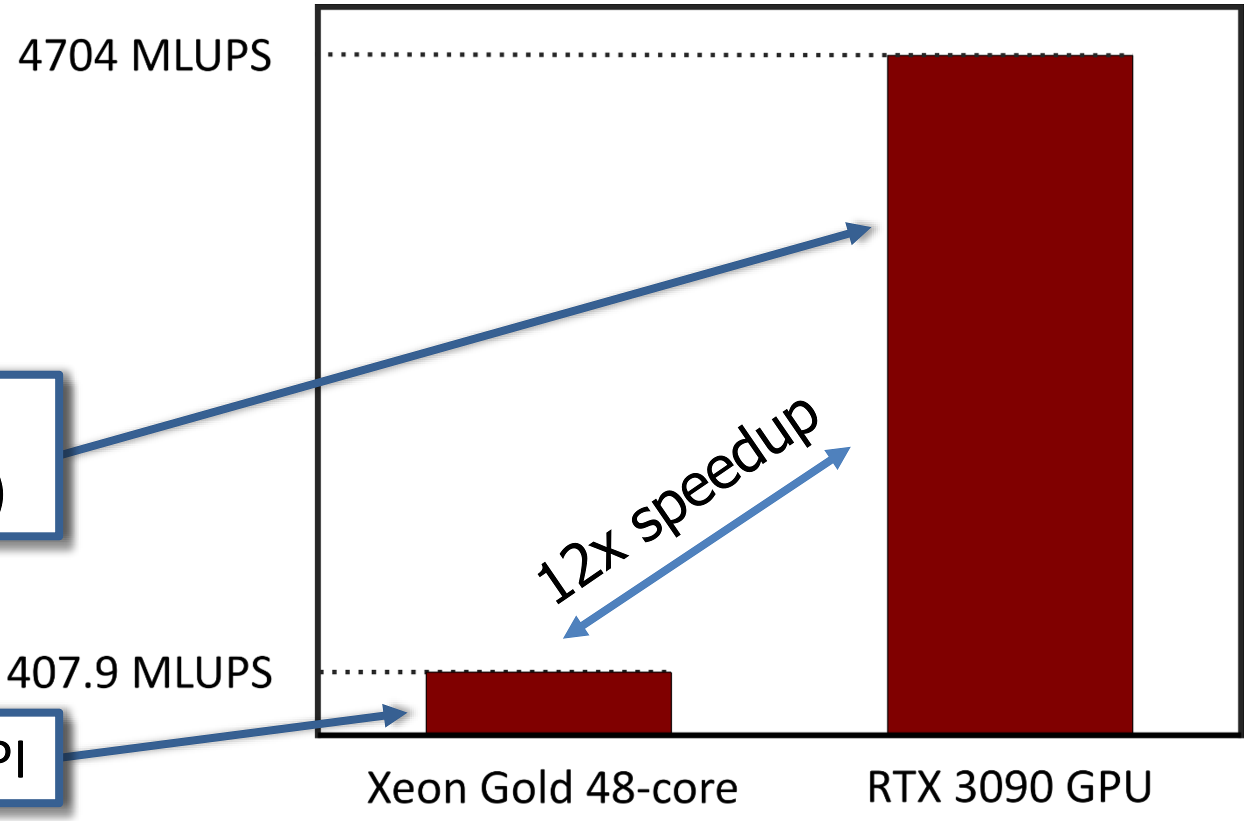
UNIVERSITÉ DE GENÈVE
FACULTY OF SCIENCE

CPU-to-GPU backend switch yields over 10-fold improvement for this complex example as well

Collision: TRT
Lattice: D3Q19
Resolution: 300x300x340
Floats: Single precision

Performance obtained with latest version of nvc++ (21.9)

Uses all 48 cores through MPI



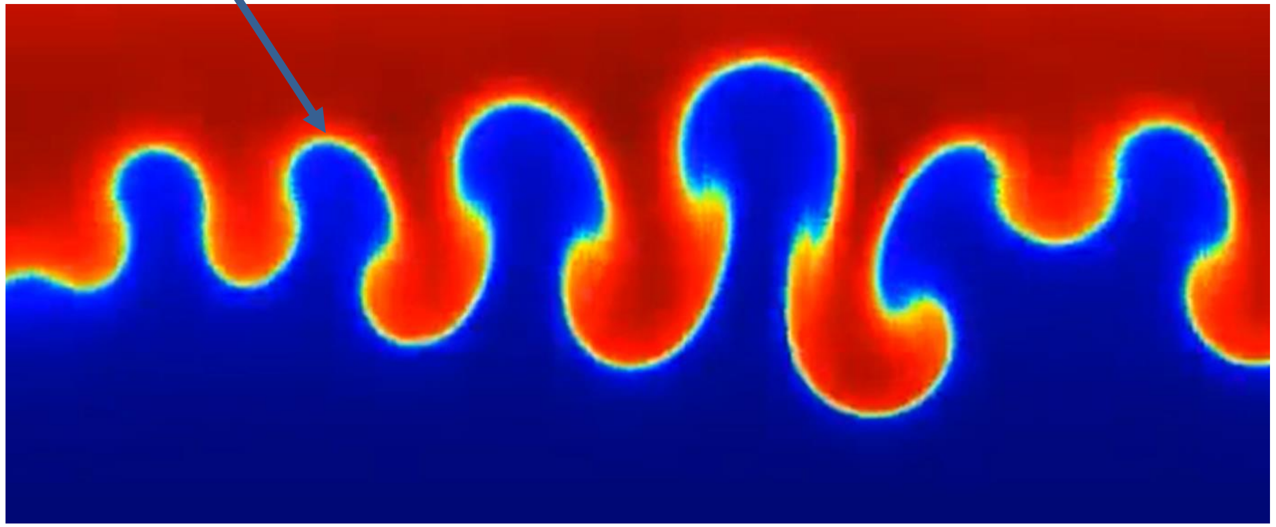
3rd case: 3D multi-component flow

UNIVERSITÉ DE GENÈVE
FACULTY OF SCIENCE

The most complex of the 3 examples, this multi-component flow requires coupling between two grids

An interaction term is evaluated to model interface physics

Each fluid component is simulated on a separate grid



Collision: BGK
Model: Pseudo-potential
Lattice: D3Q19
Resolution: 330x330x110
Floats: Single precision

Boundaries: implementation of contact physics (contact angle)

Performance: 3D multi-component flow

UNIVERSITÉ DE GENÈVE
FACULTY OF SCIENCE

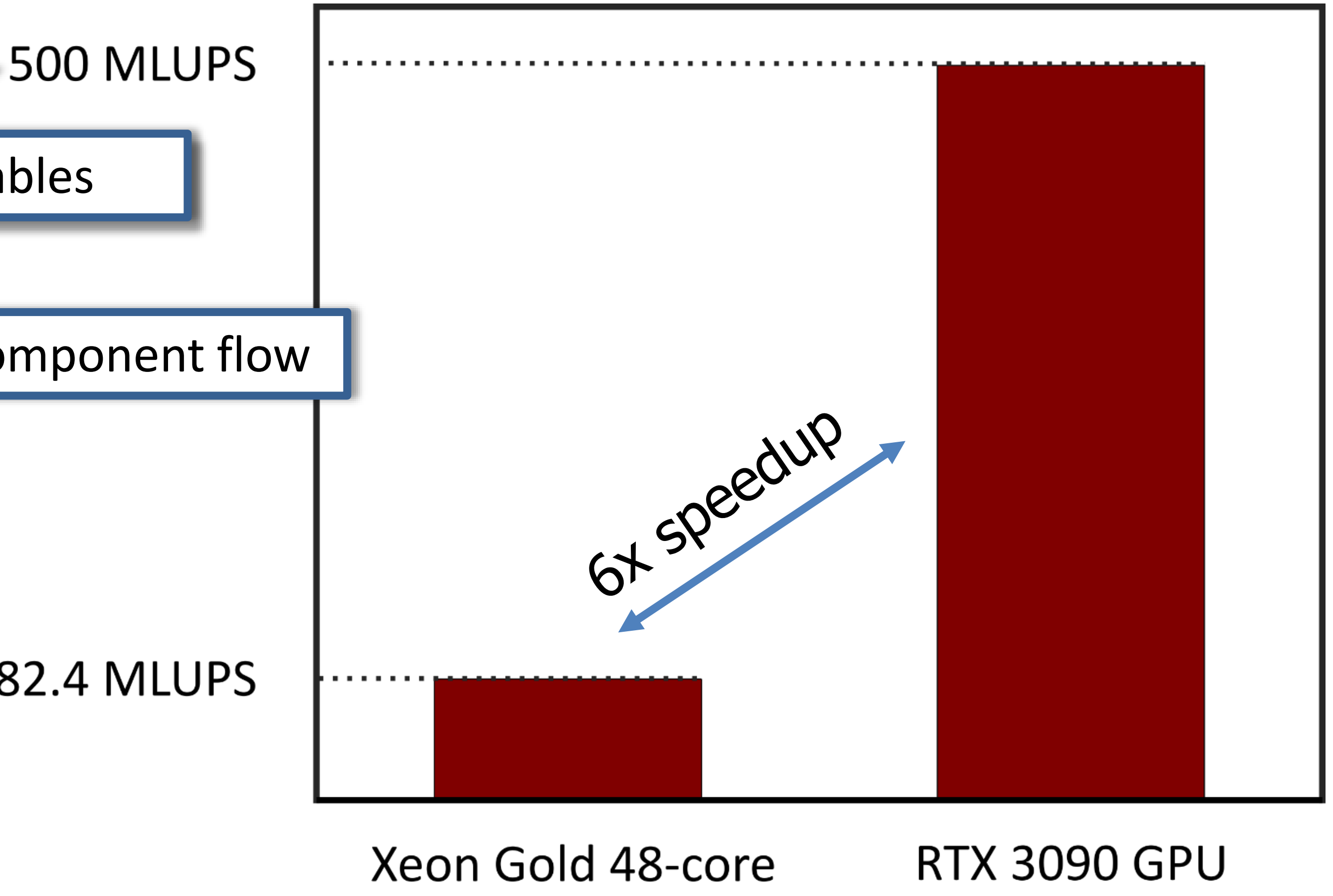
The problem is stressful for memory-bound hardware; GPU backend still yields substantial speedup

MLUPS: 1 Node = 2 x 19 variables

9 times slower than single-component flow

A factor 4 is predicted by memory roofline model

For further improvement, the multi-component interaction term should be optimized for GPU



UNIVERSITÉ
DE GENÈVE
FACULTY OF SCIENCE

Part III

IN-DEPTH: PERFORMANCE IMPROVEMENTS

Multi-GPU for a hybrid CPU/GPU code

Some components of Palabos are not ported to GPU yet.

Some components will not be ported to GPU (in particular pre-processing).

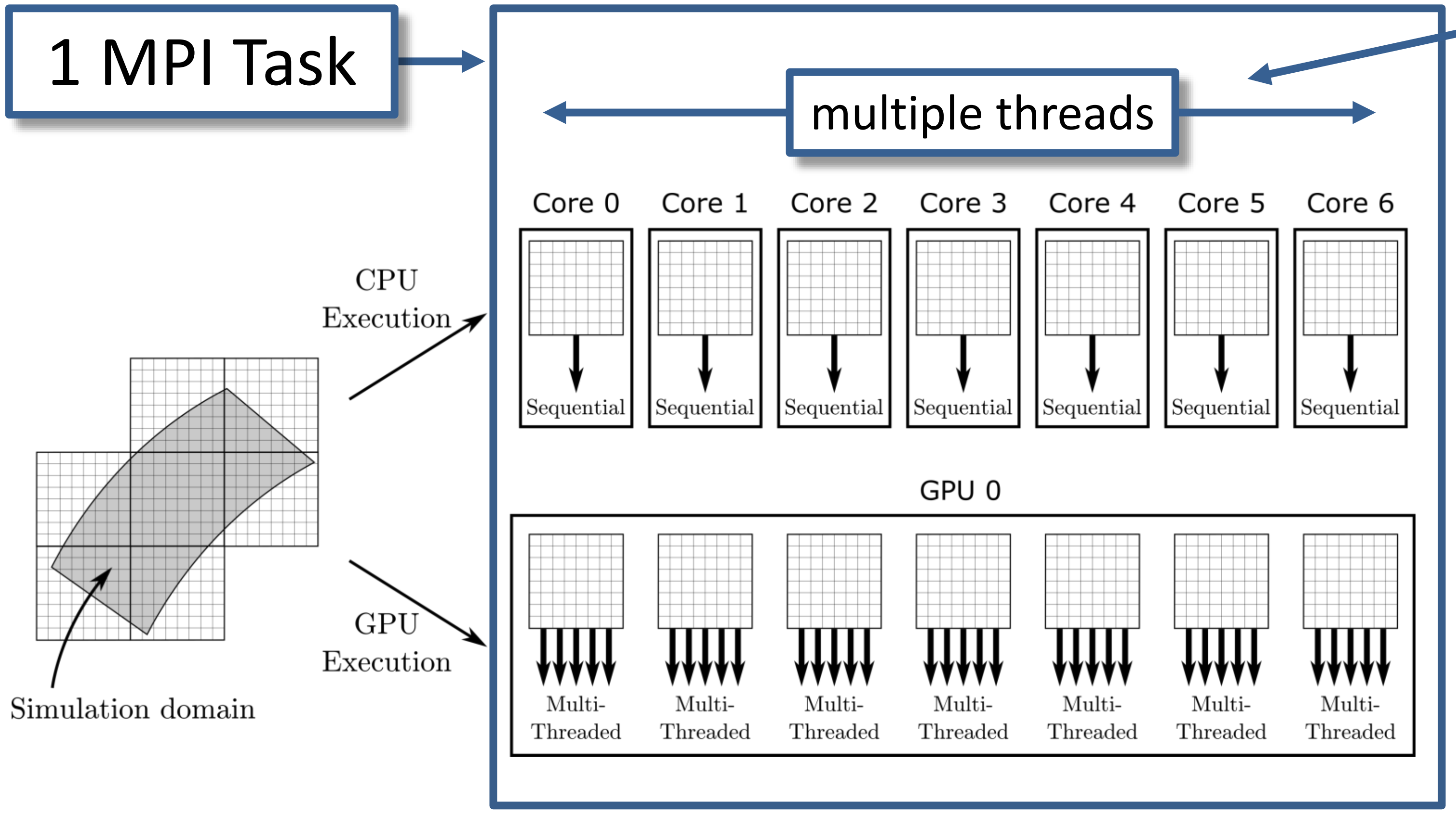
Solution: hybrid CPU / GPU execution.

With standard parallelism: everything is compiled together with nvc++.

For a multi-CPU, multi-GPU code, CPU and GPU are required to have same number of MPI threads. Problem: CPU and GPU have not been parallelized according to the same philosophy.

Solution: MPI + OpenMP on CPU

Both CPU and GPU get one MPI task per cluster node, then the CPU gets some OpenMP threads



C++ standard parallelism is used for GPU already.
We can't re-use it for the CPU, so we use OpenMP instead.

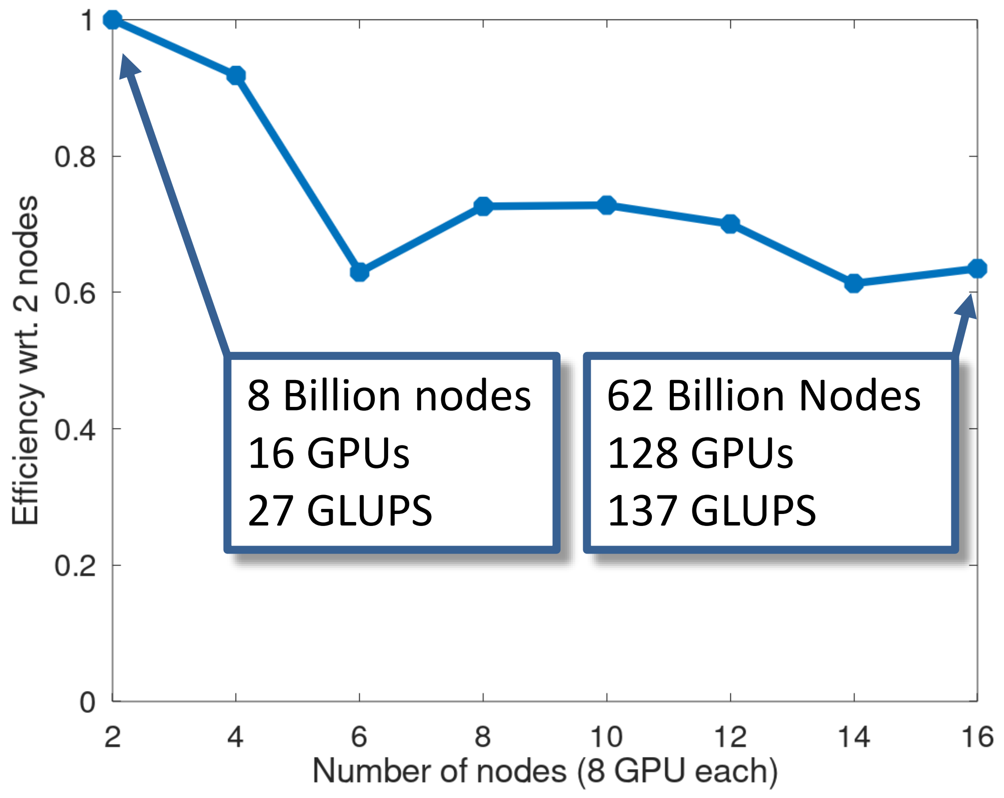
This is a literal one-line change to the code.

Validation: Multi-GPU execution

The existing MPI backend can be reused for multi-GPU as is, although performance is not brilliant

<u>Lattice Boltzmann Model</u>
Problem: Lid-driven cavity
Collision: TRT
Lattice: D3Q19
Floats: Single precision

<u>Cluster: Selene</u>
Nodes: 8 GPUs
GPUs: A100-SXM4-80GB



8 Billion nodes
16 GPUs
27 GLUPS

62 Billion Nodes
128 GPUs
137 GLUPS

Success: Multi-GPU works with existing communication layer (packing / unpacking, MPI, …)

Issue: Scaling below state-of-the-art.
Future improvements:
- Use of pinned memory for communication buffers.
- Overlap communication and computation.

Optimization: collision-kernel size

UNIVERSITÉ DE GENÈVE
FACULTY OF SCIENCE

The issue with the tag-matrix approach: somewhere in your code, you find a huge switch statement

```
template<typename T, template<typename U> class Descriptor>
void collide(int collisionModel, Array<T, Descriptor<T>::numPop>& f,
    Array<T, Descriptor<T>::ExternalField::numScalars>& ext,
    Array<T, GPUconst<T, Descriptor>::maxStaticScalars> staticScalars, T* dynamicScalars, plint index)
{
    switch (collisionModel) {
    case CollisionModel::NoDynamics:
        Collision<T, Descriptor, CollisionModel::NoDynamics>::collide(f, ext, staticScalars, dynamicScalars, index);
        break;
    case CollisionModel::BounceBack:
        Collision<T, Descriptor, CollisionModel::BounceBack>::collide(f, ext, staticScalars, dynamicScalars, index);
        break;
    case CollisionModel::BGK:
        Collision<T, Descriptor, CollisionModel::BGK>::collide(f, ext, staticScalars, dynamicScalars, index);
        break;
    case CollisionModel::BGK_ExternalMoment:
        Collision<T, Descriptor, CollisionModel::BGK_ExternalMoment>::collide(f, ext, staticScalars, dynamicScalars, index);
        break;
    case CollisionModel::TRT:
        Collision<T, Descriptor, CollisionModel::TRT>::collide(f, ext, staticScalars, dynamicScalars, index);
        // And so on, and so on ...
```

Optimization: collision-kernel size

UNIVERSITÉ DE GENÈVE
FACULTY OF SCIENCE

The issue with the tag-matrix approach: somewhere in your code, you find a huge switch statement

Problem 1: Maintenance
It is impossible to propose new models without modifying internal Palabos code.

```cpp
template<typename T, template<typename U> class Descriptor>
void collide(int collisionModel, Array<T, Descriptor<T>::numPop>& f,
    Array<T, Descriptor<T>::ExternalField::numScalars>& ext,
    Array<T, GPUconst<T, Descriptor>::maxStaticScalars> staticScalars, T* dynamicScalars, plint index)
{
    switch (collisionModel) {
    case CollisionModel::NoDynamics:
        Collision<T, Descriptor, CollisionModel::NoDynamics>::collide(f, ext, staticScalars, dynamicScalars, index);
        break;
    case CollisionModel::BounceBack:
        Collision<T, Descriptor, CollisionModel::BounceBack>::collide(f, ext, staticScalars, dynamicScalars, index);
        break;
    case CollisionModel::BGK:
        Collision<T, Descriptor, CollisionModel::BGK>::collide(f, ext, staticScalars, dynamicScalars, index);
        break;
    case CollisionModel::BGK_ExternalMoment:
        Collision<T, Descriptor, CollisionModel::BGK_ExternalMoment>::collide(f, ext, staticScalars, dynamicScalars, index);
        break;
    case CollisionModel::TRT:
        Collision<T, Descriptor, CollisionModel::TRT>::collide(f, ext, staticScalars, dynamicScalars, index);
        // And so on, and so on ...
```

Problem 2: Size of collision kernels
- Potentially hundreds of collision models.
- Everything gets compiled into a single Cuda kernel.
- Size of Cuda kernels have performance impact.

Solution: variadic templates

All models that are actually used are provided to the kernel at compile time using templates

```
lattice -> collideAndStream (
           CollisionKernel<T, DESCRIPTOR,
           CollisionModel::TRT,
           CollisionModel::BounceBack,
           CollisionModel::Boundary_RegularizedVelocity_0_1__TRT,
           CollisionModel::Boundary_RegularizedVelocity_0_M1__TRT>());
```

- All models required in the kernel are provided as template arguments.
- Variadic templates allow a variable number of template arguments.
- “Switch statement” automatically generated in pre-compilation stage.

Solution: variadic templates

```
template<int MODEL>
static void static_switch(IntList<MODEL>, int collisionModel, Array<T, Descriptor<T>::numPop>& f,
    Array<T, Descriptor<T>::ExternalField::numScalars>& ext, Array<T, GPUconst<T, Descriptor>::maxStaticScalars> staticScalars,
    T* dynamicScalars, plint index)
{
    if (collisionModel == MODEL) {
        Collision<T, Descriptor, MODEL>::collide(f, ext, staticScalars, dynamicScalars, index);
    }
    else {
        printf("Collision model not implemented: %d\n", collisionModel);
    }
}
template<int ...N>
static void static_switch(int collisionModel, Array<T, Descriptor<T>::numPop>& f,
    Array<T, Descriptor<T>::ExternalField::numScalars>& ext, Array<T, GPUconst<T, Descriptor>::maxStaticScalars> staticScalars,
    T* dynamicScalars, plint index)
{
    static_switch(IntList<N...>(), collisionModel, f, ext, staticScalars, dynamicScalars, index);
}
```

Template argument list is processed recursively on every node to match the local tag

Performance gain from using variadic templates, on RTX 3090: 12%

Conclusion

UNIVERSITÉ DE GENÈVE
FACULTY OF SCIENCE

- C++ standard parallelism allows porting a CPU code to GPU, conveniently, step-by-step, efficiently.
- Ideal for hybrid CPU / GPU codes, including multi-CPU / multi-GPU.
- The approach is entirely portable: support on different types of GPUs to be expected.
- Some low-level implementation difficulties occur, but they are addressed with reasonable efforts.
- Most problems are solved at the level of the software architecture, without technical GPU-specific knowledge.

Get the GPU-enabled branch of Palabos:
https://palabos.unige.ch/community/cpu-gpu-80-days