Parallelization in CosmoLattice

September 2025, Daejeon

Franz R. Sattler

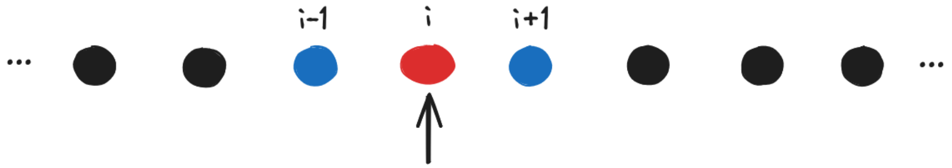
Bielefeld University

Why can we parallelize?

And why would we want to?

Example: Solving massless Klein-Gordon equation, $d = 1$

$$\partial_t^2 \phi(i) = \Delta_i^+ \Delta_i^- \phi(i)$$



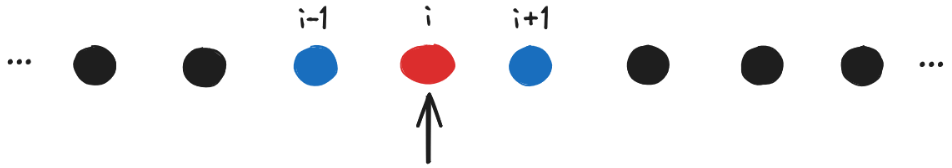
Why can we parallelize?

And why would we want to?

Example: Solving massless Klein-Gordon equation, $d = 1$

$$\partial_t \pi(i) = \Delta_i^+ \Delta_i^- \phi(i)\,,$$
$$\partial_t \phi(i) = \pi(i)\,.$$

(Leapfrog scheme.)



Why can we parallelize?

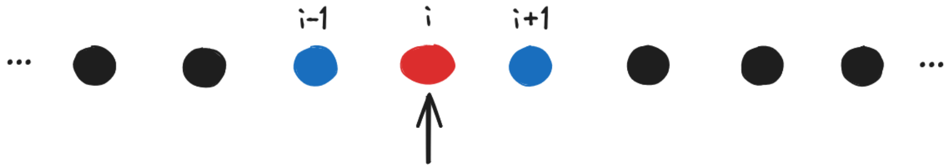
And why would we want to?

Example: Solving massless Klein-Gordon equation, $d = 1$

$$\partial_t \pi(i) = \Delta_i^+ \Delta_i^- \phi(i)\,,$$
$$\partial_t \phi(i) = \pi(i)\,.$$

(Leapfrog scheme.)

Stencil of update for **one** lattice site is $s = 1$



Why can we parallelize?

And why would we want to?

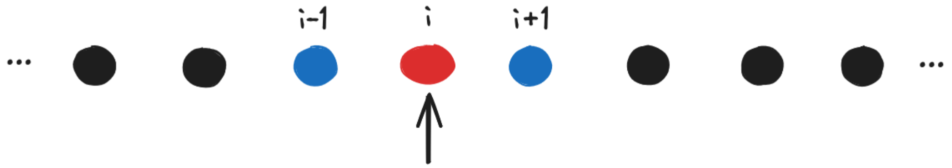
Example: Solving massless Klein-Gordon equation, $d = 1$

$$\partial_t \pi(i) = \Delta_i^+ \Delta_i^- \phi(i)\,,$$
$$\partial_t \phi(i) = \pi(i)\,.$$

(Leapfrog scheme.)

Stencil of update for **one** lattice site is $s = 1$

→ Only local information needed (neighbours).



Why can we parallelize?

And why would we want to?

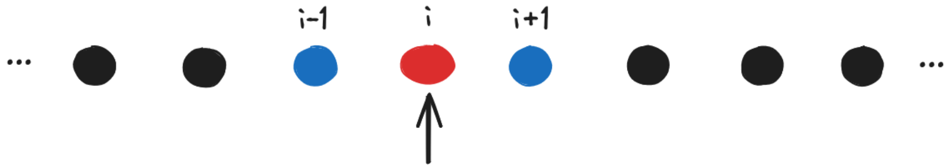
Example: Solving massless Klein-Gordon equation, $d = 1$

$$\partial_t \pi(i) = \Delta_i^+ \Delta_i^- \phi(i)\,,$$
$$\partial_t \phi(i) = \pi(i)\,.$$

(Leapfrog scheme.)

Stencil of update for **one** lattice site is $s = 1$

→ Only local information needed (neighbours).



We could compute all lattice sites independently!

See lecture on Friday!

Why can we parallelize?

And why would we want to?

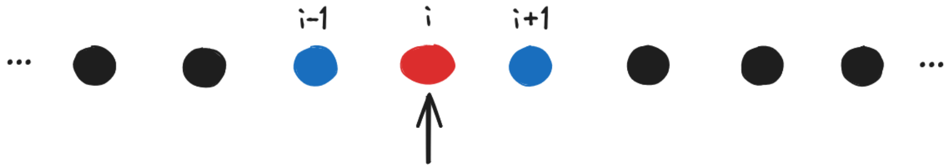
Example: Solving massless Klein-Gordon equation, $d = 1$

$$\partial_t \pi(i) = \Delta_i^+ \Delta_i^- \phi(i)\,,$$
$$\partial_t \phi(i) = \pi(i)\,.$$

(Leapfrog scheme.)

Stencil of update for **one** lattice site is $s = 1$

→ Only local information needed (neighbours).



We could compute all lattice sites independently!

See lecture on Friday!

Why can we parallelize?

And why would we want to?

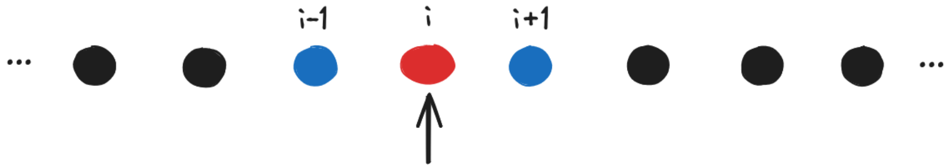
Example: Solving massless Klein-Gordon equation, $d = 1$

$$\partial_t \pi(i) = \Delta_i^+ \Delta_i^- \phi(i)\,,$$
$$\partial_t \phi(i) = \pi(i)\,.$$

(Leapfrog scheme.)

Stencil of update for **one** lattice site is $s = 1$

→ Only local information needed (neighbours).



We could compute all lattice sites independently!

See lecture on Friday!

Less granular: split sub-regions of lattice across many computers (nodes)

Parallelization of CosmoLattice

simulations requires to split both computation and data across cores.

Parallelization of CosmoLattice

simulations requires to split both computation and data across cores.

Cores: **Nodes** (distributed) and **Threads** (shared).

Parallelization of CosmoLattice

simulations requires to split both computation and data across cores.

Cores: **Nodes** (distributed) and **Threads** (shared).

Type	distributed	shared
Data	split between **nodes**	shared by all **threads**
Computation	split between **nodes**	split between **threads**

Parallelization of CosmoLattice

simulations requires to split both computation and data across cores.

$$\text{speedup} = \frac{1}{(1-\alpha) + \frac{\alpha}{n_{\text{cores}}}}$$

α = part of the code which runs in parallel

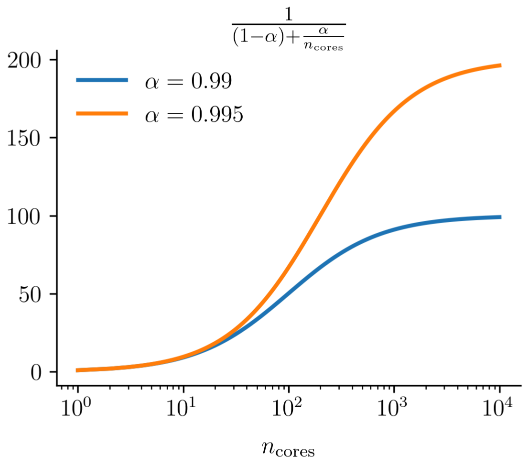
n_{cores} = speedup of the parallel part

CosmoLattice: $\alpha \gtrsim 0.99$

(see manual)

Parallelization of CosmoLattice

simulations requires to split both computation and data across cores.



$$\text{speedup} = \frac{1}{(1-\alpha) + \frac{\alpha}{n_{\rm cores}}}$$

α = part of the code which runs in parallel

$n_{\rm cores}$ = speedup of the parallel part

CosmoLattice: $\alpha \gtrsim 0.99$

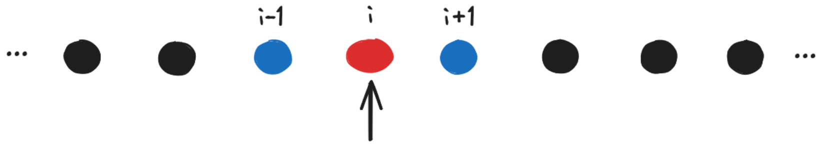
(see manual)

Why can we parallelize?

And why would we want to?

Example: Solving massless Klein-Gordon equation, $d = 1$

$$\partial_t \pi(i) = \Delta_i^+ \Delta_i^- \phi(i)\,,$$
$$\partial_t \phi(i) = \pi(i)\,.$$



(Leapfrog scheme.)

Stencil of update for **one** lattice site is $s = 1$

→ Only local information needed (neighbours).

→ Each site can be updated independently.

Why can we parallelize?

And why would we want to?

Example: Solving massless Klein-Gordon equation, $d = 1$

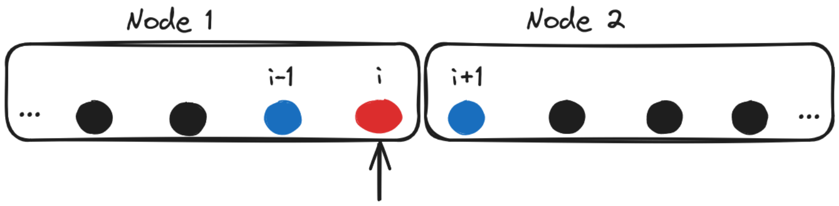
$$\partial_t \pi(i) = \Delta_i^+ \Delta_i^- \phi(i)\,,$$
$$\partial_t \phi(i) = \pi(i)\,.$$

(Leapfrog scheme.)

Stencil of update for **one** lattice site is $s = 1$

→ Only local information needed (neighbours).

→ Each site can be updated independently.



Why can we parallelize?

And why would we want to?

Example: Solving massless Klein-Gordon equation, $d = 1$

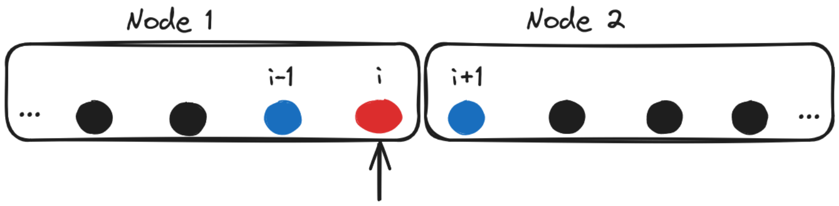
$$\partial_t \pi(i) = \Delta_i^+ \Delta_i^- \phi(i)\,,$$
$$\partial_t \phi(i) = \pi(i)\,.$$

(Leapfrog scheme.)

Stencil of update for **one** lattice site is $s = 1$

→ Only local information needed (neighbours).

→ Each site can be updated independently.



Problem: Data is missing on node 1

Why can we parallelize?

And why would we want to?

Example: Solving massless Klein-Gordon equation, $d = 1$

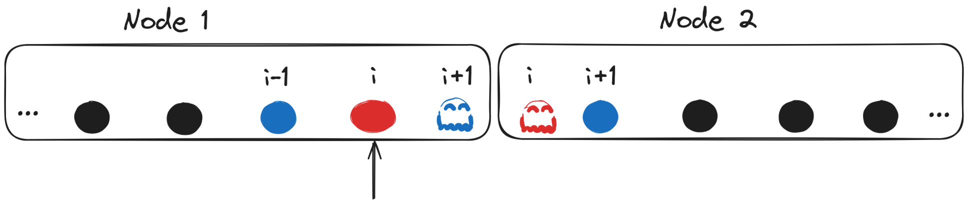
$$\partial_t \pi(i) = \Delta_i^+ \Delta_i^- \phi(i)\,,$$
$$\partial_t \phi(i) = \pi(i)\,.$$

(Leapfrog scheme.)

Stencil of update for **one** lattice site is $s = 1$

→ Only local information needed (neighbours).

→ Each site can be updated independently.



Solution: Use ghosts.

Ghosts are local copies of data on other nodes.

Why can we parallelize?

And why would we want to?

Example: Solving massless Klein-Gordon equation, $d = 1$

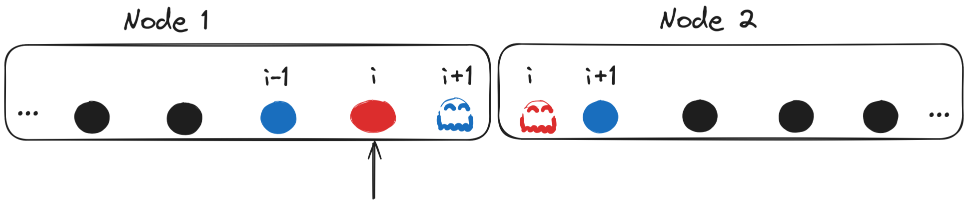
$$\partial_t \pi(i) = \Delta_i^+ \Delta_i^- \phi(i)\,,$$
$$\partial_t \phi(i) = \pi(i)\,.$$

(Leapfrog scheme.)

Stencil of update for **one** lattice site is $s = 1$

→ Only local information needed (neighbours).

→ Each site can be updated independently.



Solution: Use ghosts.

Ghosts are local copies of data on other nodes.

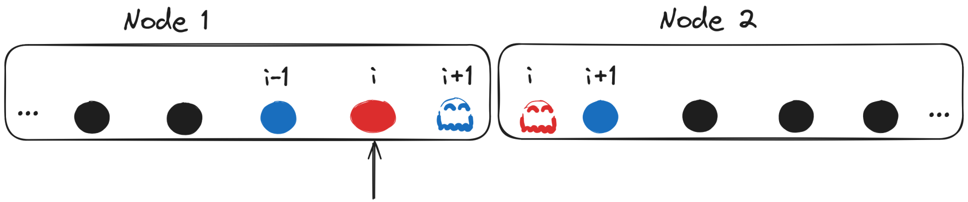
Need to update ghosts after every time-step.

Data communication

The standard for communication in distributed-memory applications:

Message **P**assing **I**nterface (MPI)

Exchange ghost data between **nodes** over the **network** automatically if anything changes.

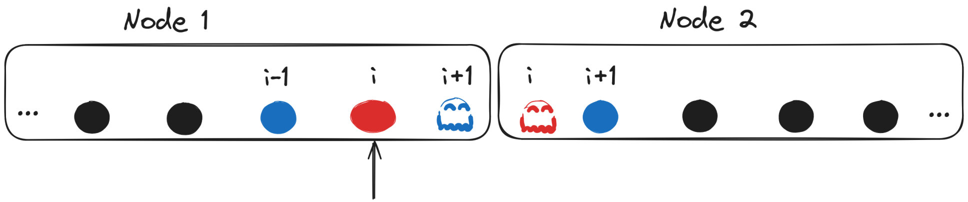


Data communication

The standard for communication in distributed-memory applications:

Message **P**assing **I**nterface (MPI)

Exchange ghost data between **nodes** over the **network** automatically if anything changes.



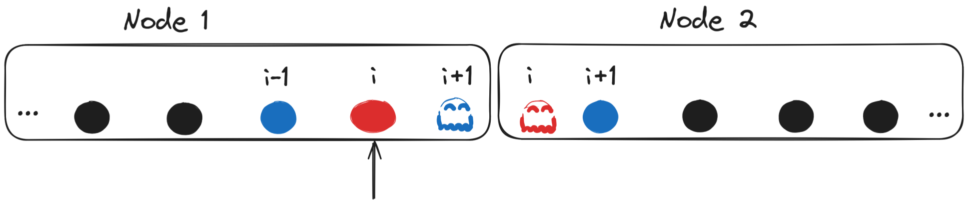
CosmoLattice does this automatically under the hood!

Data communication

The standard for communication in distributed-memory applications:

Message **P**assing **I**nterface (MPI)

Exchange ghost data between **nodes** over the **network** automatically if anything changes.



CosmoLattice does this automatically under the hood!

How to turn it on?

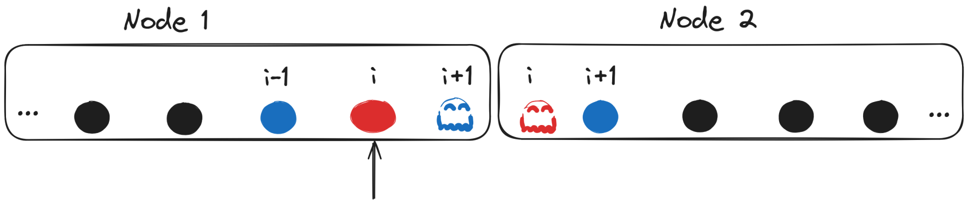
```
$ cd build/
$ cmake -DMPI=ON ..
$ make
$ mpirun -n 16 ./lphi4 input=./lphi4.in
```

Data communication

The standard for communication in distributed-memory applications:

Message **P**assing **I**nterface (MPI)

Exchange ghost data between **nodes** over the **network** automatically if anything changes.



CosmoLattice does this automatically under the hood!

How to turn it on?

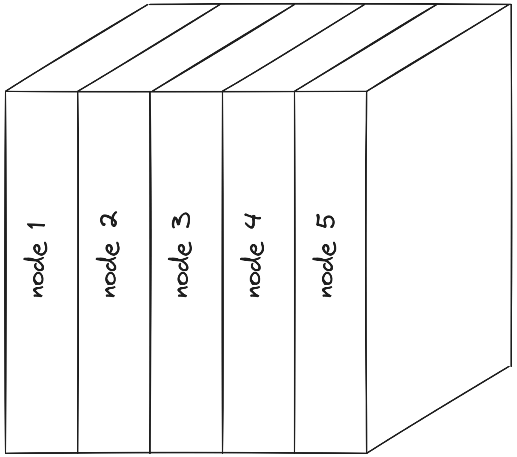
```
$ cd build/
$ cmake -DMPI=ON ..
$ make
$ mpirun -n 16 ./lphi4 input=./lphi4.in
```

You may need to install fftw3 with MPI support.

```
$ sudo apt-get install libfftw3-mpi-dev
```

FFT parallelization

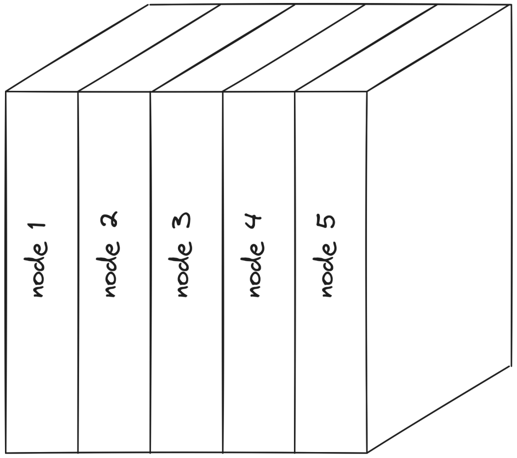
- **FFTW** supports parallelization along 1 direction.

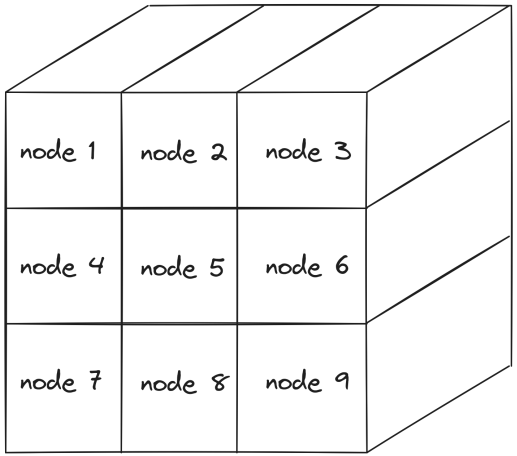


FFT parallelization

- **FFTW** supports parallelization along 1 direction.

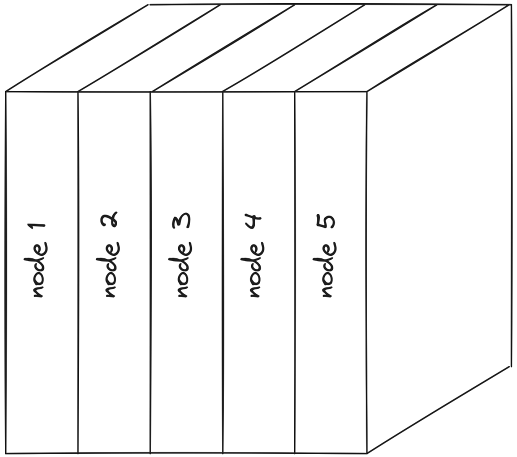
- Improved scaling for many nodes: **PFFT** allows for parallelization in any dimensions.

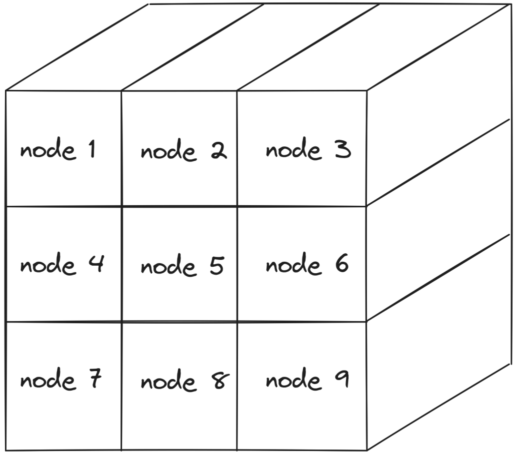




FFT parallelization

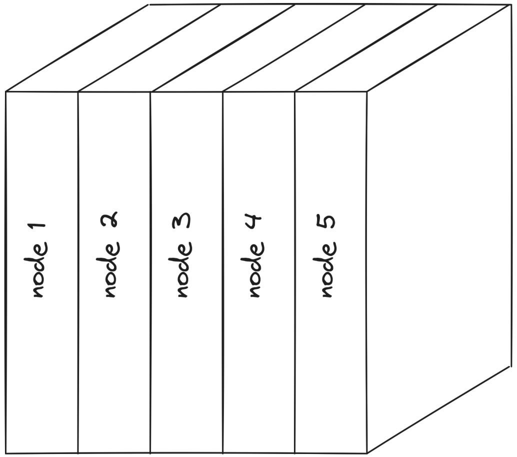
- **FFTW** supports parallelization along 1 direction.
- Improved scaling for many nodes: **PFFT** allows for parallelization in any dimensions.
- To keep data transfer due to ghost exchange manageable, parallelization along $d - 1$ directions.





FFT parallelization

- **FFTW** supports parallelization along 1 direction.
- Improved scaling for many nodes: **PFFT** allows for parallelization in any dimensions.
- To keep data transfer due to ghost exchange manageable, parallelization along $d-1$ directions.



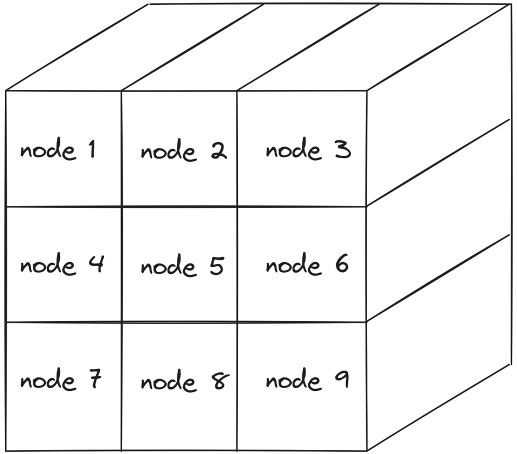
1D

$$N = n_p * m$$

$$N = 50$$

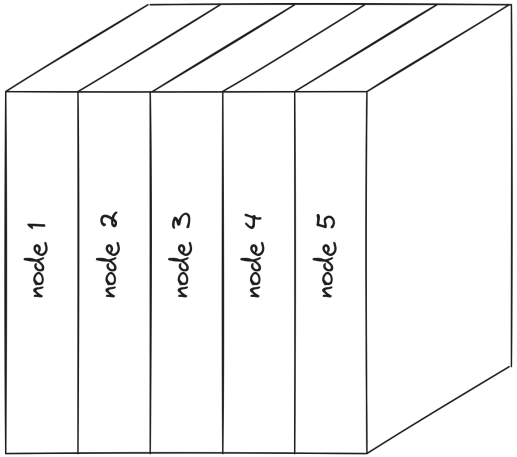
2D

$$N = n_p^{(1)} * m$$
$$= n_p^{(2)} * m$$



FFT parallelization

- **FFTW** supports parallelization along 1 direction.
- Improved scaling for many nodes: **PFFT** allows for parallelization in any dimensions.
- To keep data transfer due to ghost exchange manageable, parallelization along $d-1$ directions.



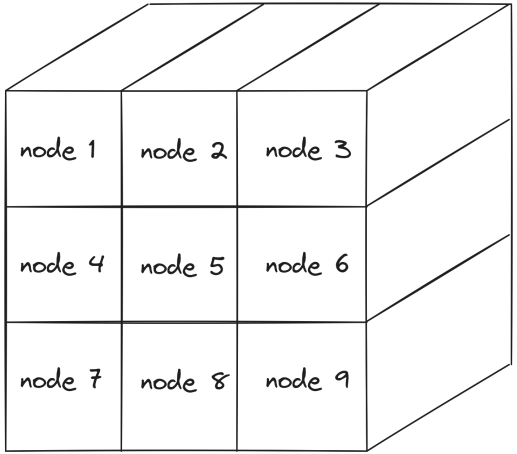
1D

$$N = n_p * m$$

$$N = 50$$

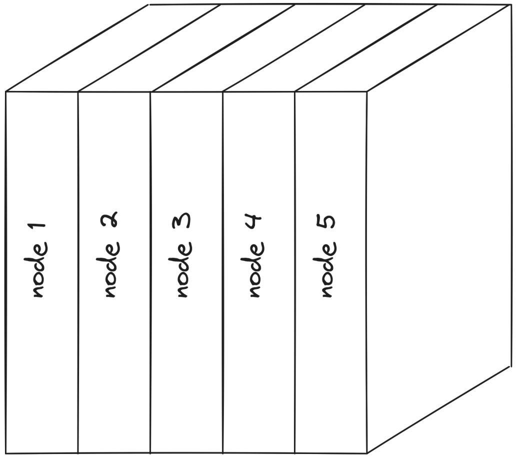
2D

$$N = n_p^{(1)} * m$$
$$= n_p^{(2)} * m$$



FFT parallelization

- **FFTW** supports parallelization along 1 direction.
- Improved scaling for many nodes: **PFFT** allows for parallelization in any dimensions.
- To keep data transfer due to ghost exchange manageable, parallelization along $d-1$ directions.



1D

$$N = n_p * m$$

2D

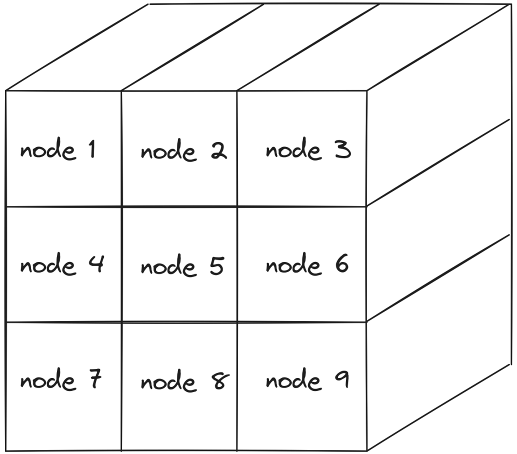
$$N = n_p^{(1)} * m = n_p^{(2)} * m$$

$$N = 50$$

Maximum parallelization

25 nodes.

$25^2 = 625$ nodes.



FFT parallelization

- **FFTW** supports parallelization along 1 direction.
- Improved scaling for many nodes: **PFFT** allows for parallelization in any dimensions.
- To keep data transfer due to ghost exchange manageable, parallelization along $d - 1$ directions.

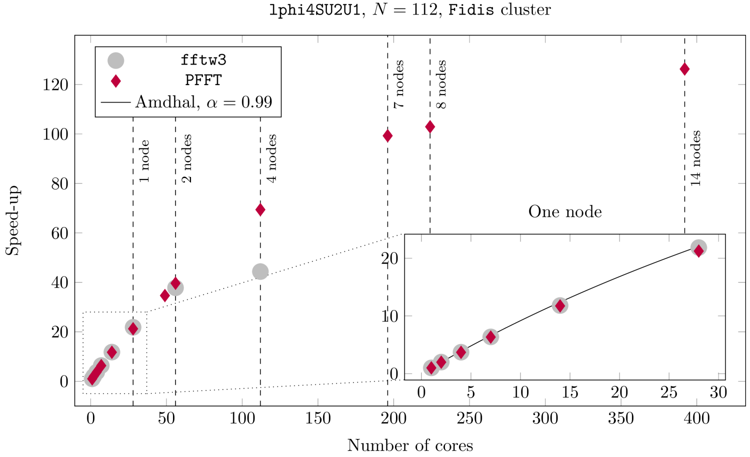
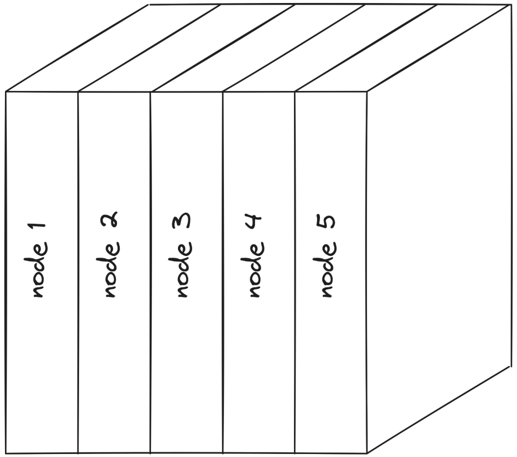
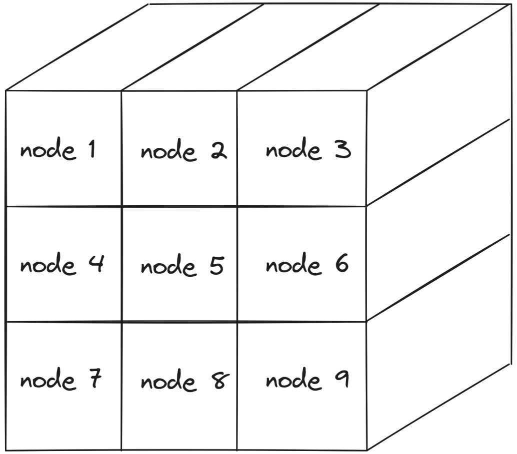


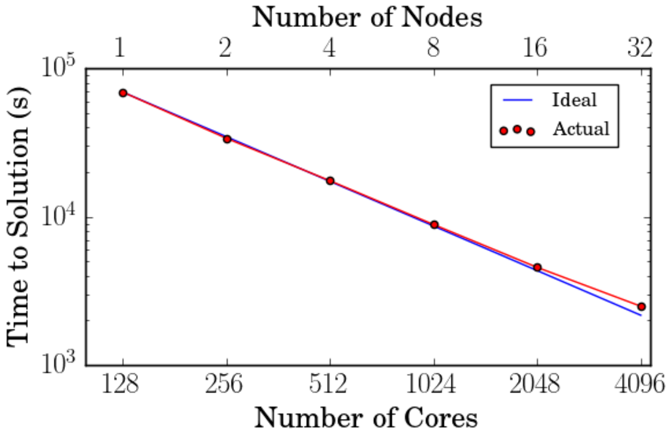
Figure 3: Speed up factor in parallelized simulations as a number of cores (tested on the Gacrux cluster from the EPFL HPC center SCITAS, Switzerland).

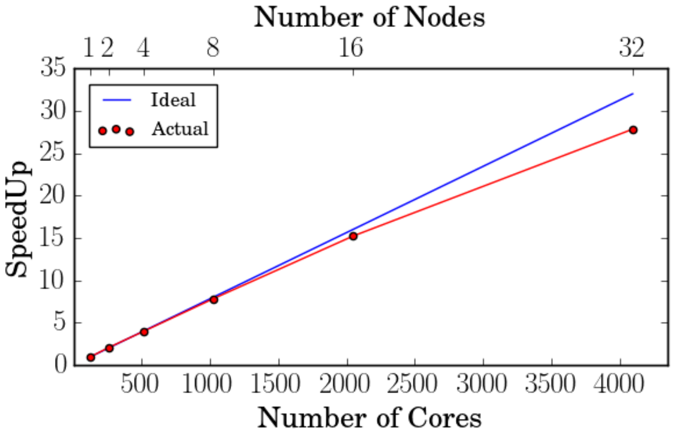




Going to larger clusters

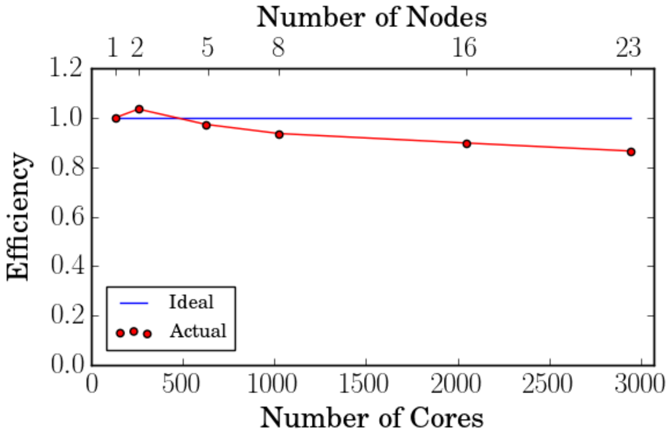
Strong scaling
(same lattice size, more cores)

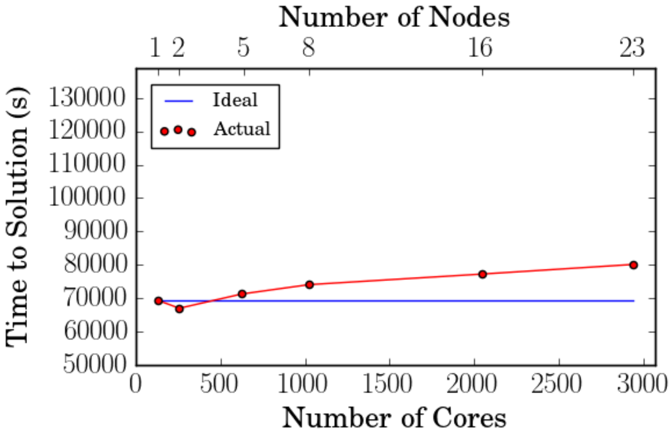




Going to larger clusters

Weak scaling
(lattice size ~ cores)





Questions?

Tomorrow: Shared-memory parallelization with GPUs.

CosmoLattice on GPUs

September 2025, Daejeon

Franz R. Sattler

Bielefeld University

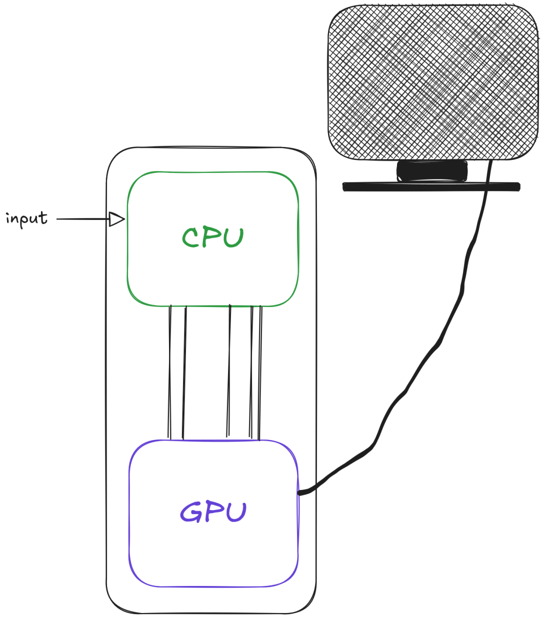
Current computers can be broadly said to have two main processing units:

Current computers can be broadly said to have two main processing units: CPUs (central processing units)

Current computers can be broadly said to have two main processing units: CPUs (central processing units) and GPUs (graphical processing units)

Current computers can be broadly said to have two main processing units: CPUs (central processing units) and GPUs (graphical processing units)

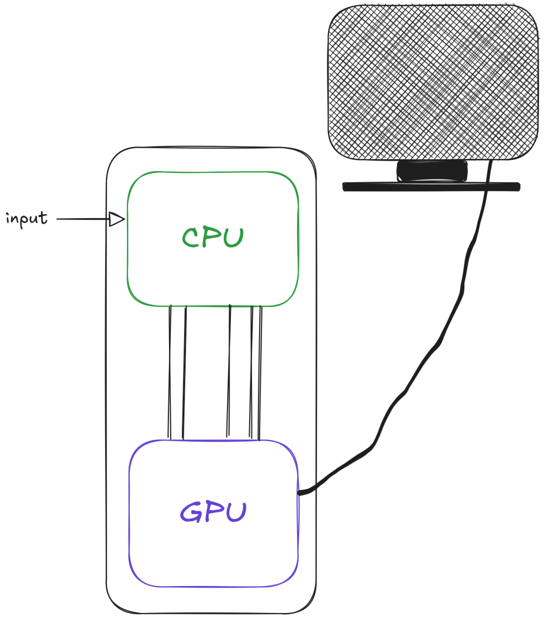
CPUs: for **OS**, **computation**, **general applications**.
GPUs: dedicated just for **video** and **graphics** applications.



A consumer machine

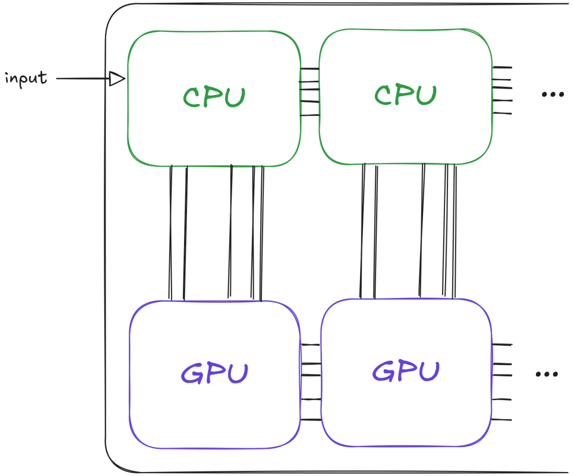
Current computers can be broadly said to have two main processing units: CPUs (central processing units) and GPUs (graphical processing units)

CPUs: for **OS**, **computation**, **general applications**.
GPUs: dedicated just for **video** and **graphics** applications.



A consumer machine

Current (heterogeneous) clusters have both CPUs and GPUs for **computations**.



A typical heterogeneous computing cluster

Why use GPUs for lattice?

Lattice points independently computed & updated → Limit of threads is number of lattice sites!

Why use GPUs for lattice?

Lattice points independently computed & updated → Limit of threads is number of lattice sites!

- AMD EPYC 7763: **64**
- Intel Xeon 6148 (Skylake): **20**
- AMD Ryzen 9 7945HX: **16**

	CPU	GPU
Cores/Node	$\mathcal{O}(10-100)$	$\mathcal{O}(10000)$
Clock speed	~ 3 GHz	~ 1.5 GHz

- Nvidia H100: **~15000**
- Nvidia 4070 RTX mobile: **~5000**

Why use GPUs for lattice?

Lattice points independently computed & updated → Limit of threads is number of lattice sites!

- AMD EPYC 7763: **64**
- Intel Xeon 6148 (Skylake): **20**
- AMD Ryzen 9 7945HX: **16**

	CPU	GPU
Cores/Node	$\mathcal{O}(10-100)$	$\mathcal{O}(10000)$
Clock speed	~ 3 GHz	~ 1.5 GHz

- Nvidia H100: **~15000**
- Nvidia 4070 RTX mobile: **~5000**

CPU: Low parallelization, high clock speed

GPU: High parallelization, moderate clock speed

→ CosmoLattice on GPUs has the potential for *massive parallelism* with $\gg 10^5$ simultaneous operations.

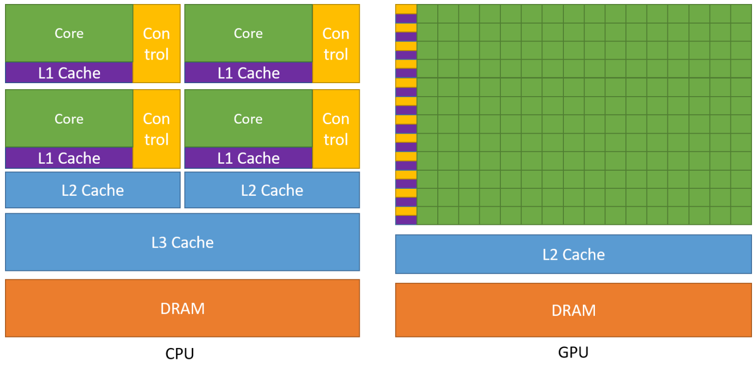
Why use GPUs for lattice?

Lattice points independently computed & updated → Limit of threads is number of lattice sites!

	CPU	GPU
Cache/Thread	64KB / 16MB	1KB
Local Cache	64MB	256KB / 50MB

CPU: **Thread-constrained**

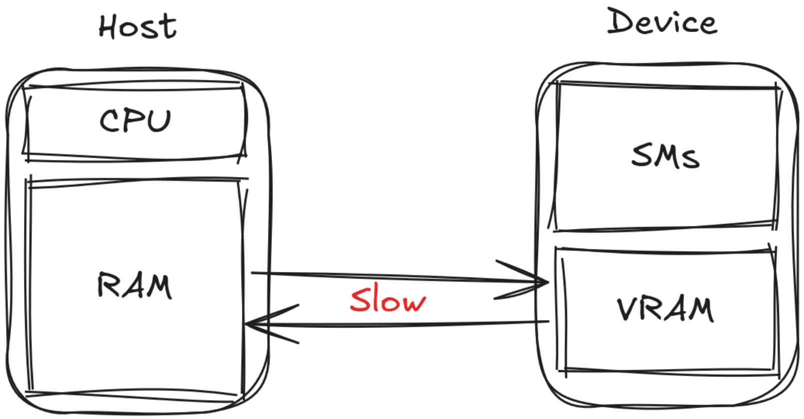
GPU: **Memory-constrained**



Hardware layouts [docs.nvidia.com]

Redesigning TempLat for GPUs

Device-centric programming.

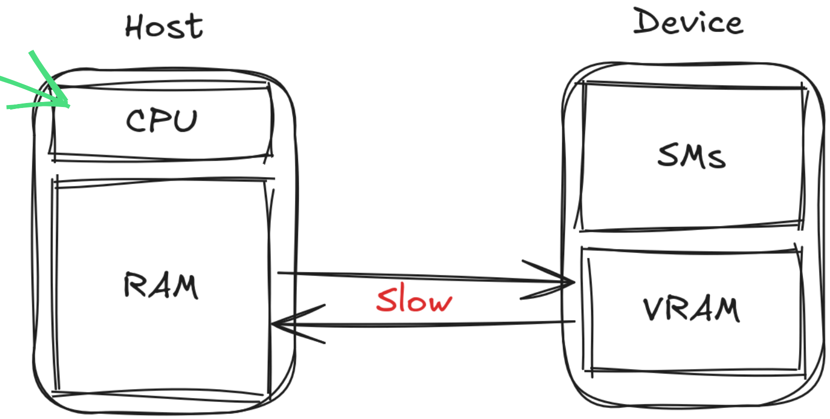


Redesigning TempLat for GPUs

Device-centric programming.

Sequential work,

- expression construction
- program flow logic
- device dispatch.



$$\text{expression} = \pi(t) + dt \cdot \Delta\phi(t)$$

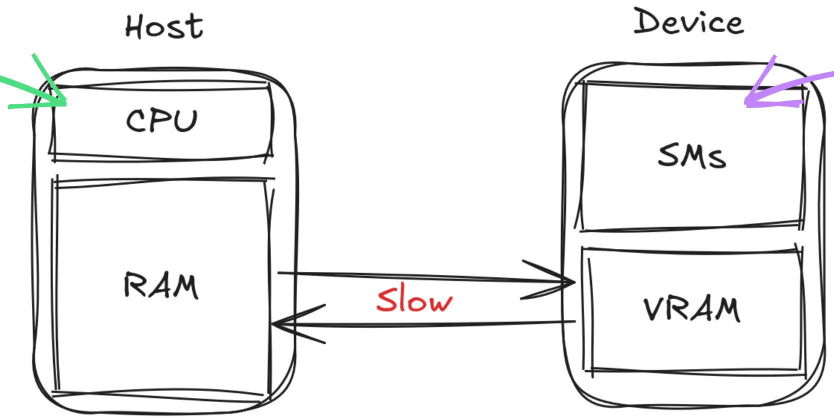
Time evolution

Redesigning TempLat for GPUs

Device-centric programming.

Sequential work,

- expression construction
- program flow logic
- device dispatch.



Parallel (grid-based) work,

- time evolution
- measurements
- memory reordering.

$$\text{expression} = \pi(t) + dt \cdot \Delta\phi(t)$$

Time evolution

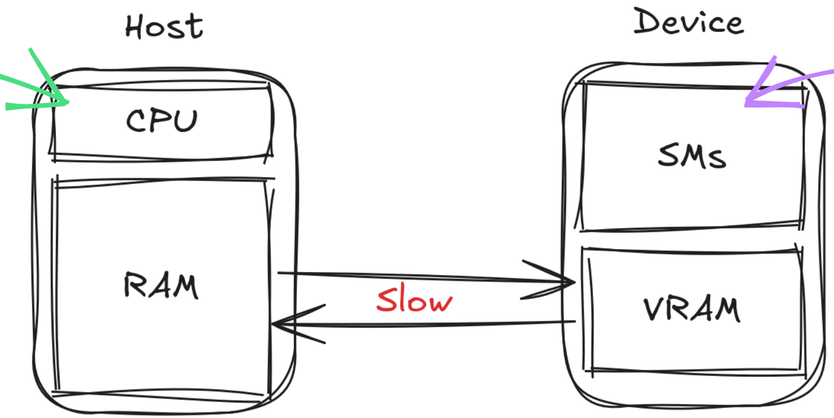
$$\pi(t + dt) = \text{expression}$$

Redesigning TempLat for GPUs

Device-centric programming.

Sequential work,

- expression construction
- program flow logic
- device dispatch.



Parallel (grid-based) work,

- time evolution
- measurements
- memory reordering.

$$\text{expression} = \pi(t) + dt \cdot \Delta\phi(t)$$

Time evolution

$$\pi(t + dt) = \text{expression}$$

```
auto functor = DEVICE_LAMBDA(device::IdxArray<NDim> idx,
                             double& max) {
  // ...
};
double maximum = 0.;
```

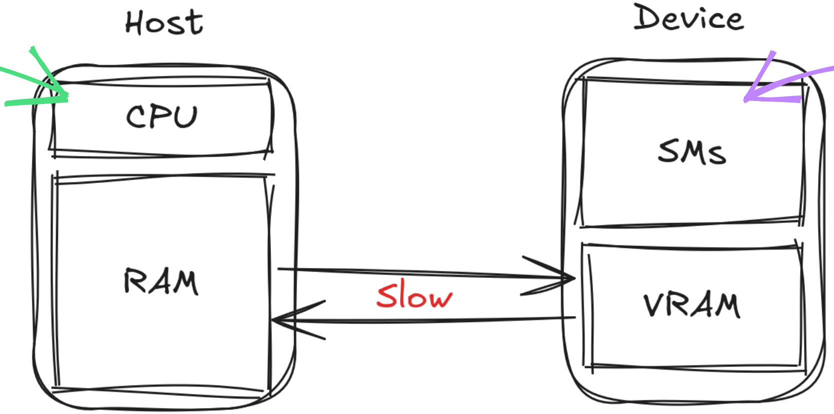
Maximum

Redesigning TempLat for GPUs

Device-centric programming.

Sequential work,

- expression construction
- program flow logic
- device dispatch.



Parallel (grid-based) work,

- time evolution
- measurements
- memory reordering.

$$\text{expression} = \pi(t) + dt \cdot \Delta\phi(t)$$

Time evolution

$$\pi(t + dt) = \text{expression}$$

```
auto functor = DEVICE_LAMBDA(device::IdxArray<NDim> idx,
                             double& max) {
 // ...
};
double maximum = 0.;
```

Maximum

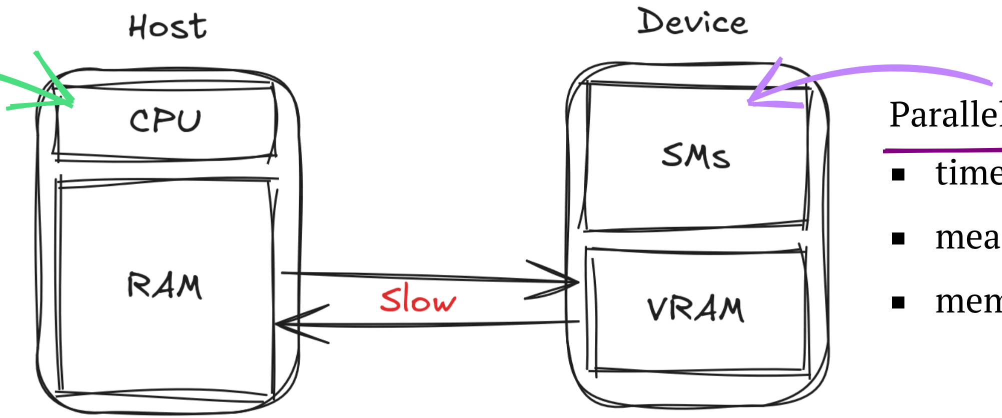
```
device::iteration::reduce("Maximum", functor, maximum);
```

Redesigning TempLat for GPUs

Device-centric programming.

Sequential work,

- expression construction
- program flow logic
- device dispatch.



Parallel (grid-based) work,

- time evolution
- measurements
- memory reordering.

Standard C++ on CPU

Hardware-dependent

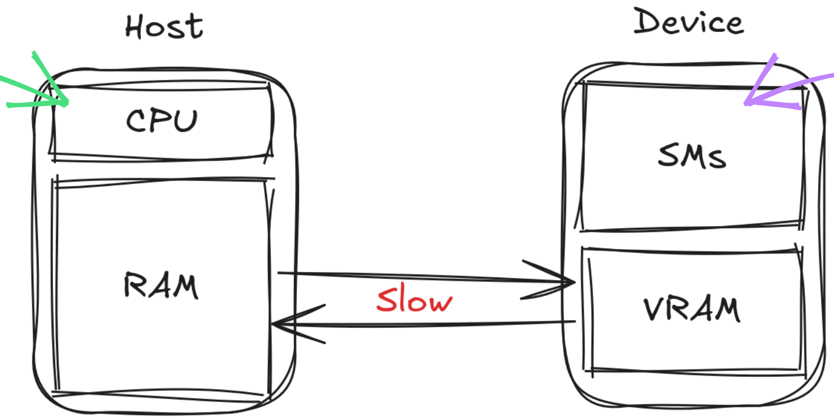
- Nvidia: CUDA
- AMD: ROCM
- Intel: SYCL
- shared-memory CPUs
- FPGAs

Redesigning TempLat for GPUs

Device-centric programming.

Sequential work,

- expression construction
- program flow logic
- device dispatch.



Parallel (grid-based) work,

- time evolution
- measurements
- memory reordering.

Standard C++ on CPU

Backends

- *Kokkos*
- *Sequential STL (2020/2023)*
- …

Abstracted away in TempLat

- `device::iterate::foreach`
- `device::iterate::reduce`
- `device::memory::copyHostToDevice`
- …

Does this make CosmoLattice harder to use?

Does this make CosmoLattice harder to use?

No, but…

Does this make CosmoLattice harder to use?

No, but…

Model file

```
public:

  MODELNAME(ParameterParser &parser, RunParameters<double> &runPar,
      std::shared_ptr<MemoryToolBox> toolBox)
        ...
```

Does this make CosmoLattice harder to use?

No, but...

Model file

```
public:
  static constexpr size_t NDim = Model<MODELNAME>::NDim;

  MODELNAME(ParameterParser &parser, RunParameters<double> &runPar,
      std::shared_ptr<MemoryToolBox<NDim>> toolBox)
    ...
```

Does this make CosmoLattice harder to use?

No, but…

Model file

```
public:
  static constexpr size_t NDim = Model<MODELNAME>::NDim;

  MODELNAME(ParameterParser &parser, RunParameters<double> &runPar,
      std::shared_ptr<MemoryToolBox<NDim>> toolBox)
    ...
```

TempLat

```
vType computeConfigurationSpace() {
  vType localResult{};

  auto& it = mT.getToolBox()->itX();
  for(it.begin();it.end();++it)
  {
    const ptrdiff_t i = it();
    localResult += GetValue::get(mT,i);
  }

  return mWorkspace;
}
```

Does this make CosmoLattice harder to use?

No, but…

Model file

```cpp
public:
  static constexpr size_t NDim = Model<MODELNAME>::NDim;

  MODELNAME(ParameterParser &parser, RunParameters<double> &runPar,
      std::shared_ptr<MemoryToolBox<NDim>> toolBox)
    ...
```

TempLat

```cpp
vType computeConfigurationSpace() {
  auto functor = DEVICE_CLASS_LAMBDA(const device::IdxArray<NDim> &idx,
                                     vType &update) {
    device::apply([&](const auto &...args) {
          update += GetValue::get(mT, args...);
        },
        idx);
  };

  vType localResult{};
  device::iteration::reduce("Averager", cLayout, functor, localResult);
  return localResult;
}
```

Does this make CosmoLattice harder to use?

No, but...

Model file

```cpp
  public:
    static constexpr size_t NDim = Model<MODELNAME>::NDim;

    MODELNAME(ParameterParser &parser, RunParameters<double> &runPar,
        std::shared_ptr<MemoryToolBox<NDim>> toolBox)
      ...
```

Building

Cosmolattice up to now:

```
$ cmake .. -DMODEL=lphi4
...
```

TempLat

```cpp
vType computeConfigurationSpace() {
  auto functor = DEVICE_CLASS_LAMBDA(const device::IdxArray<NDim> &idx,
                                     vType &update) {
    device::apply([&](const auto &...args) {
          update += GetValue::get(mT, args...);
        },
        idx);
  };

  vType localResult{};
  device::iteration::reduce("Averager", cLayout, functor, localResult);
  return localResult;
}
```

Does this make CosmoLattice harder to use?

No, but…

Model file

```
  public:
    static constexpr size_t NDim = Model<MODELNAME>::NDim;

    MODELNAME(ParameterParser &parser, RunParameters<double> &runPar,
        std::shared_ptr<MemoryToolBox<NDim>> toolBox)
      ...
```

Building

New version: CUDA is detected automatically:

```
$ cmake .. -DMODEL=lphi4
...
```

TempLat

```
vType computeConfigurationSpace() {
  auto functor = DEVICE_CLASS_LAMBDA(const device::IdxArray<NDim> &idx,
                                     vType &update) {
    device::apply([&](const auto &...args) {
          update += GetValue::get(mT, args...);
        },
        idx);
  };

  vType localResult{};
  device::iteration::reduce("Averager", cLayout, functor, localResult);
  return localResult;
}
```

Does this make CosmoLattice harder to use?

No, but...

Model file

```
  public:
    static constexpr size_t NDim = Model<MODELNAME>::NDim;

    MODELNAME(ParameterParser &parser, RunParameters<double> &runPar,
        std::shared_ptr<MemoryToolBox<NDim>> toolBox)
      ...
```

Building

Granular control: shared memory OpenMP through Kokkos

```
$ cmake .. -DMODEL=lphi4 -DDEVICE=KOKKOS -DCUDA=OFF
...
```

TempLat

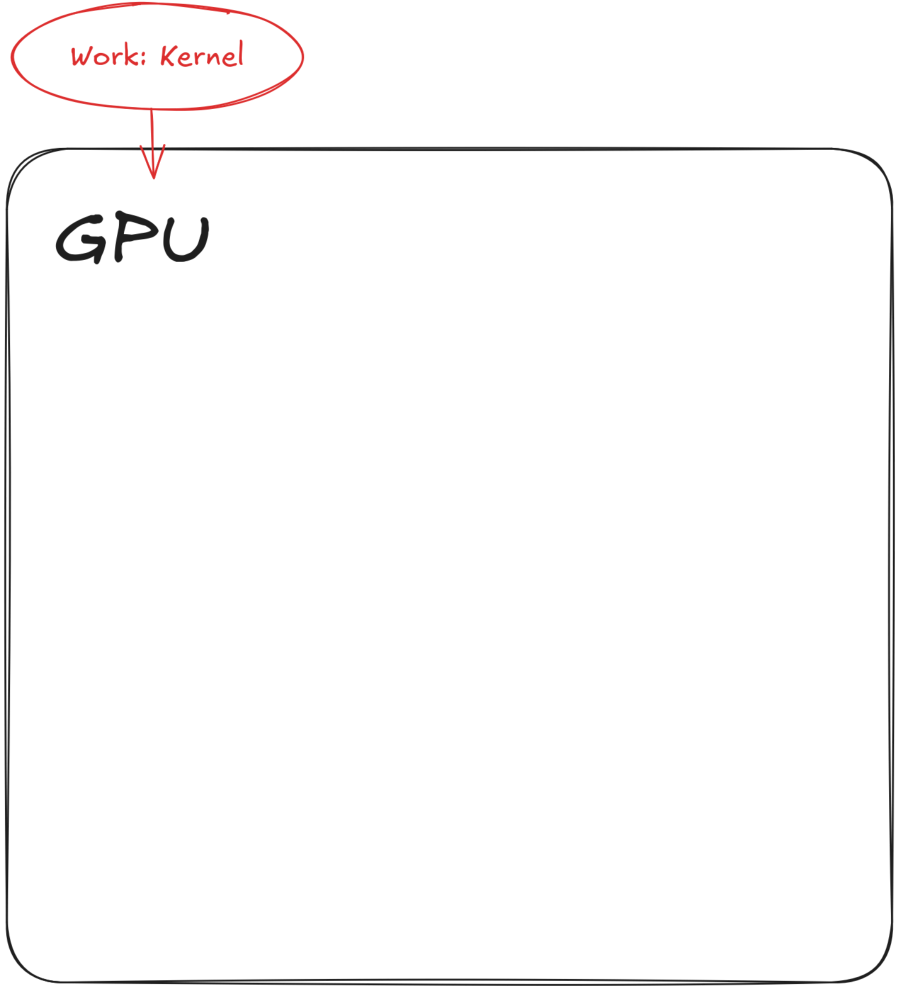
```
vType computeConfigurationSpace() {
  auto functor = DEVICE_CLASS_LAMBDA(const device::IdxArray<NDim> &idx,
                                     vType &update) {
    device::apply([&](const auto &...args) {
          update += GetValue::get(mT, args...);
        },
        idx);
  };

  vType localResult{};
  device::iteration::reduce("Averager", cLayout, functor, localResult);
  return localResult;
}
```

For "average" user:

Only minimal changes

GPU architecture

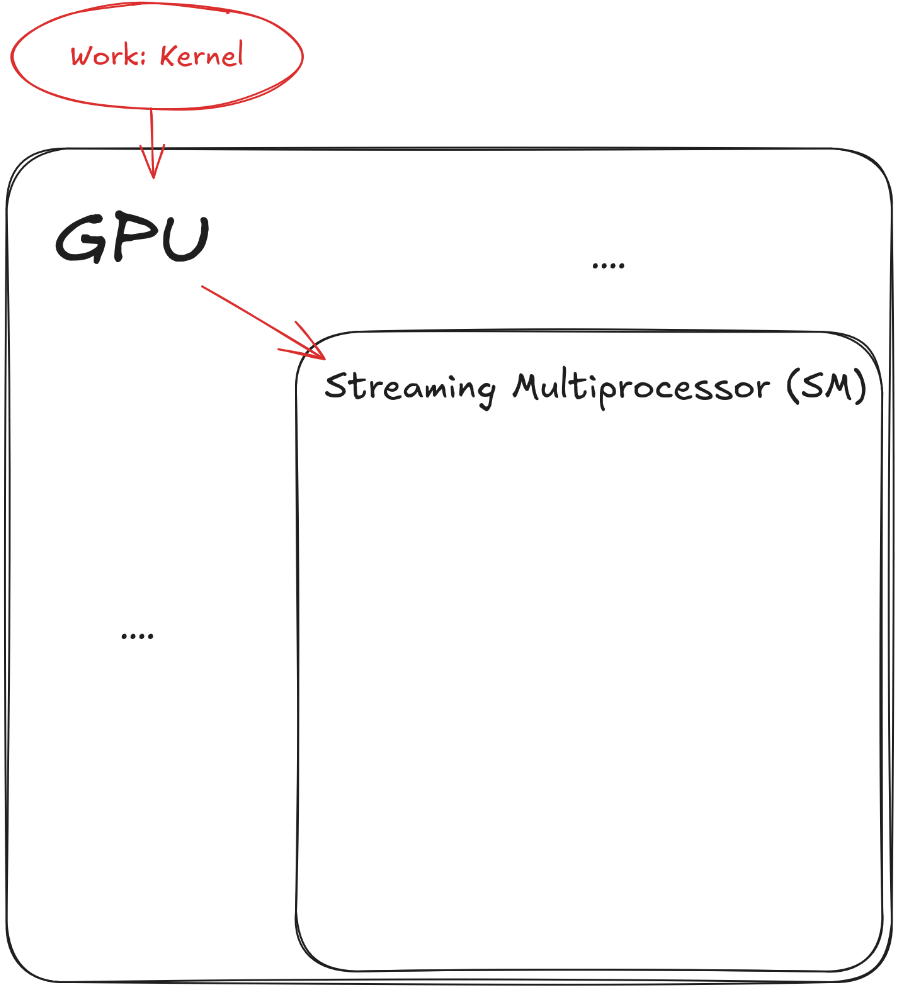
GPU thread hierarchy



GPU architecture

GPU thread hierarchy

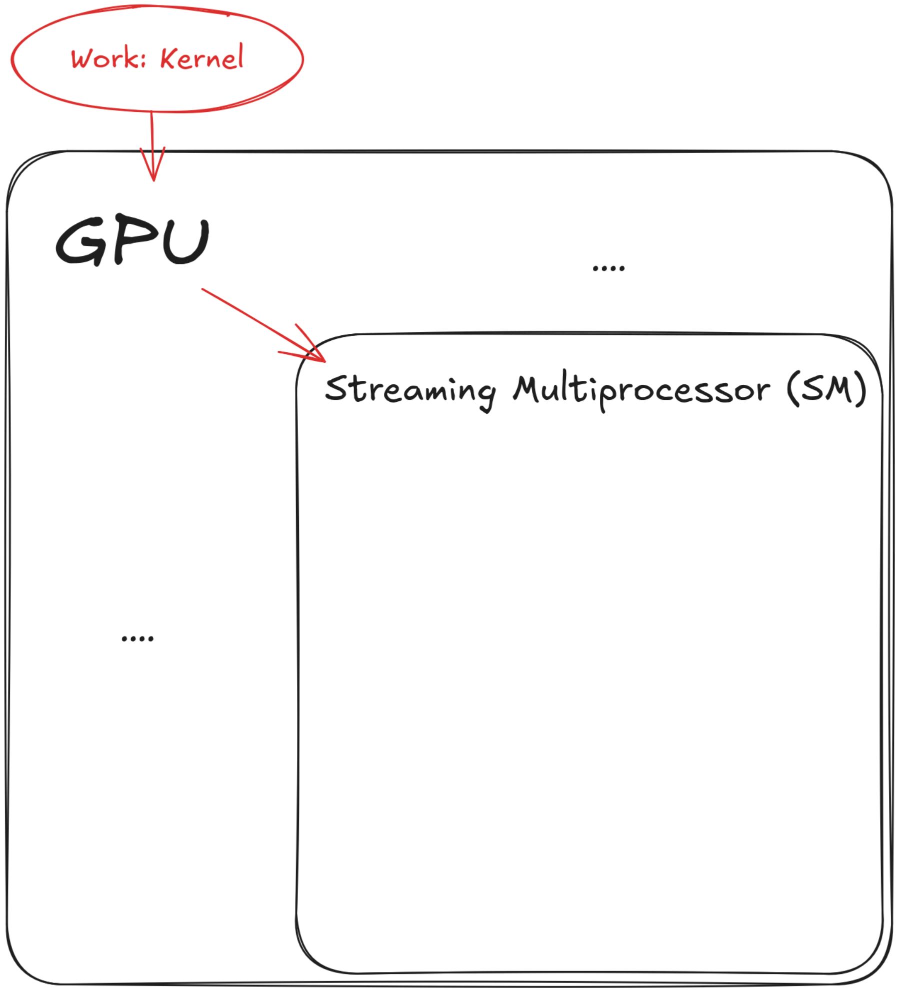
- GPUs have $\mathcal{O}(10)$ *Streaming multiprocessors*.



GPU architecture

GPU thread hierarchy

- GPUs have $\mathcal{O}(10)$ *Streaming multiprocessors*.
  - SMs execute *kernels* with series of parallel instructions.
  - SMs schedule their execution.

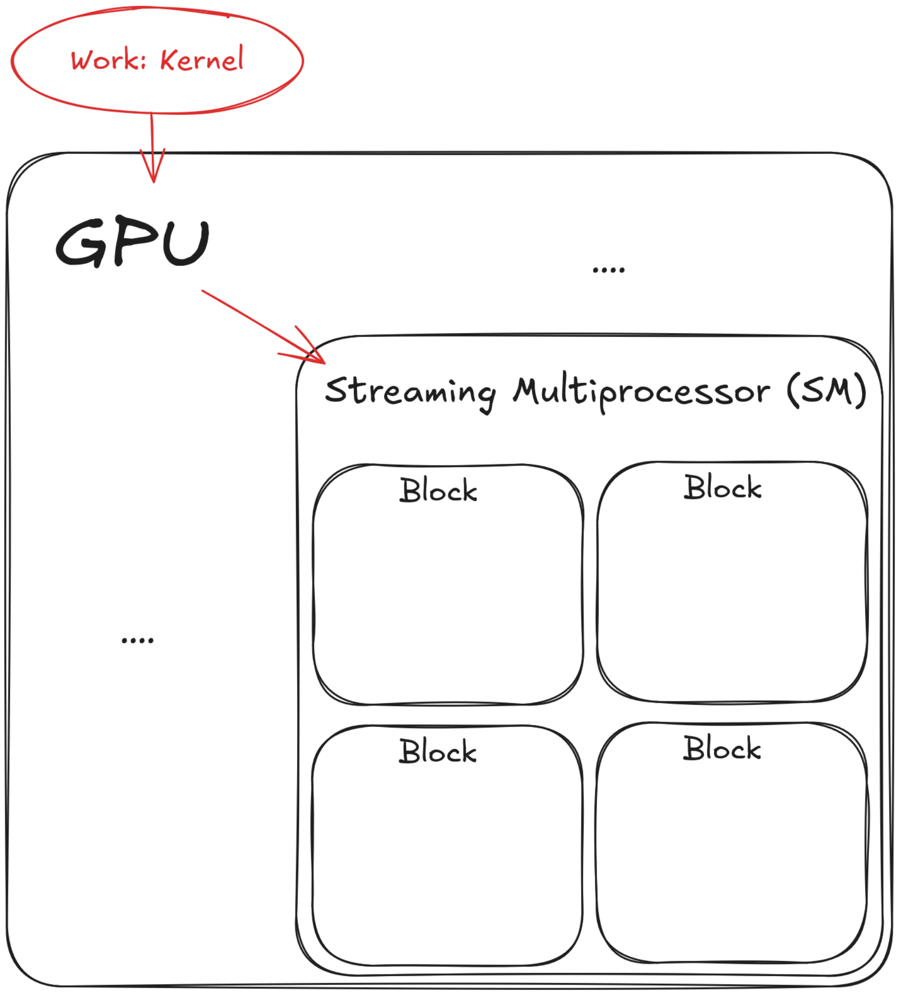


GPU architecture

GPU thread hierarchy

- GPUs have $\mathcal{O}(10)$ *Streaming multiprocessors*.
  - SMs execute *kernels* with series of parallel instructions.
  - SMs schedule their execution.

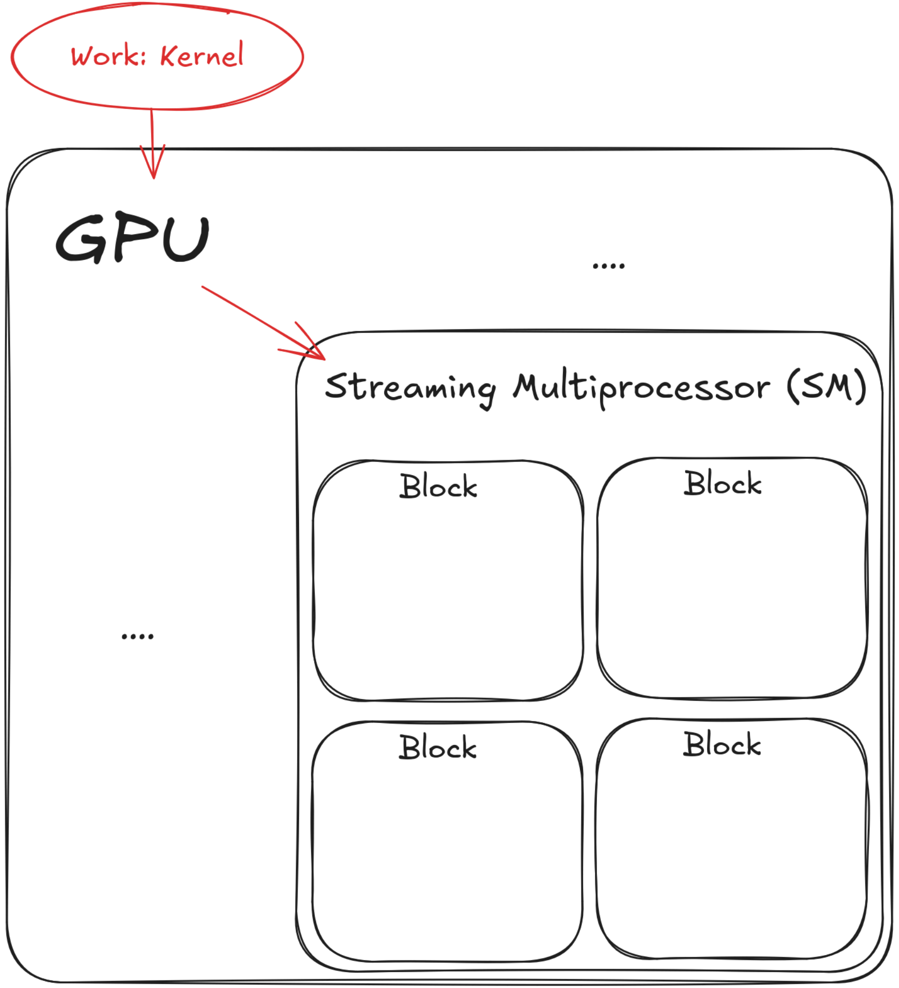
- Work given to SMs in *blocks*.



GPU architecture

GPU thread hierarchy

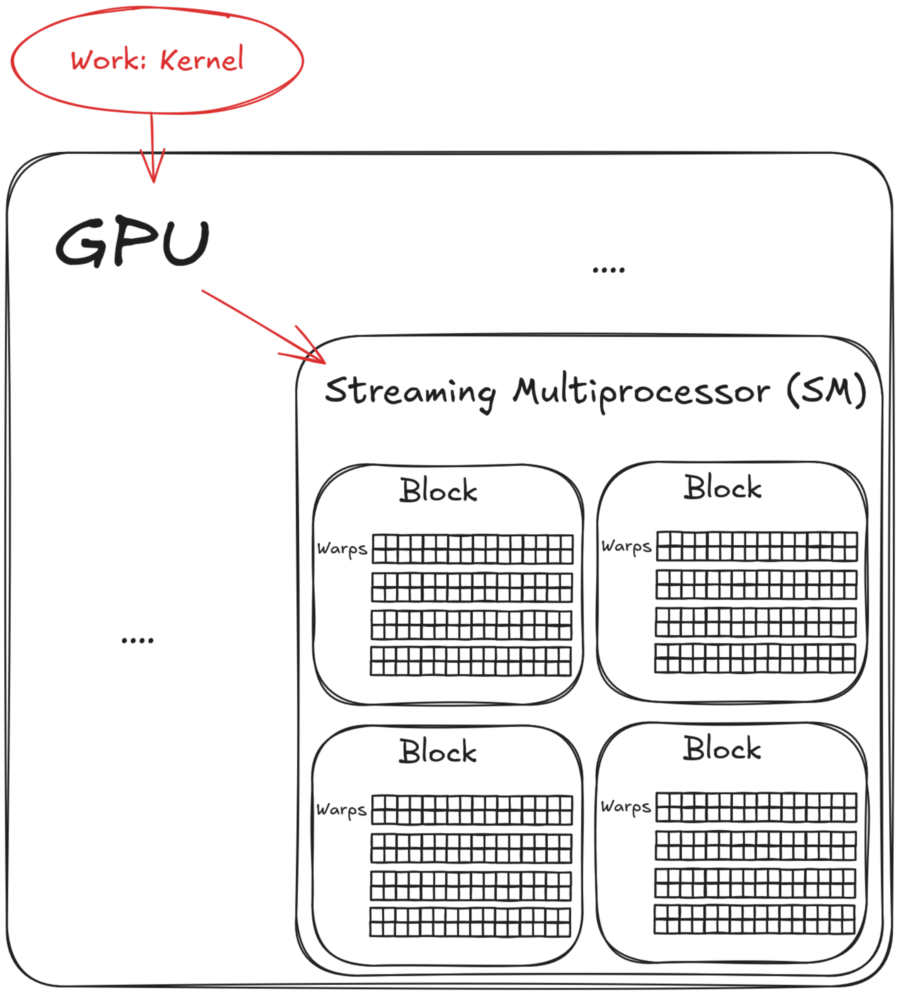
- GPUs have $\mathcal{O}(10)$ *Streaming multiprocessors*.
  - SMs execute *kernels* with series of parallel instructions.
  - SMs schedule their execution.
- Work given to SMs in *blocks*.
  - A block has to fit onto a SM's hardware capabilities (~1024 threads/block).
  - Each block's (sub-)contexts are persistent throughout its execution.



GPU architecture

GPU thread hierarchy

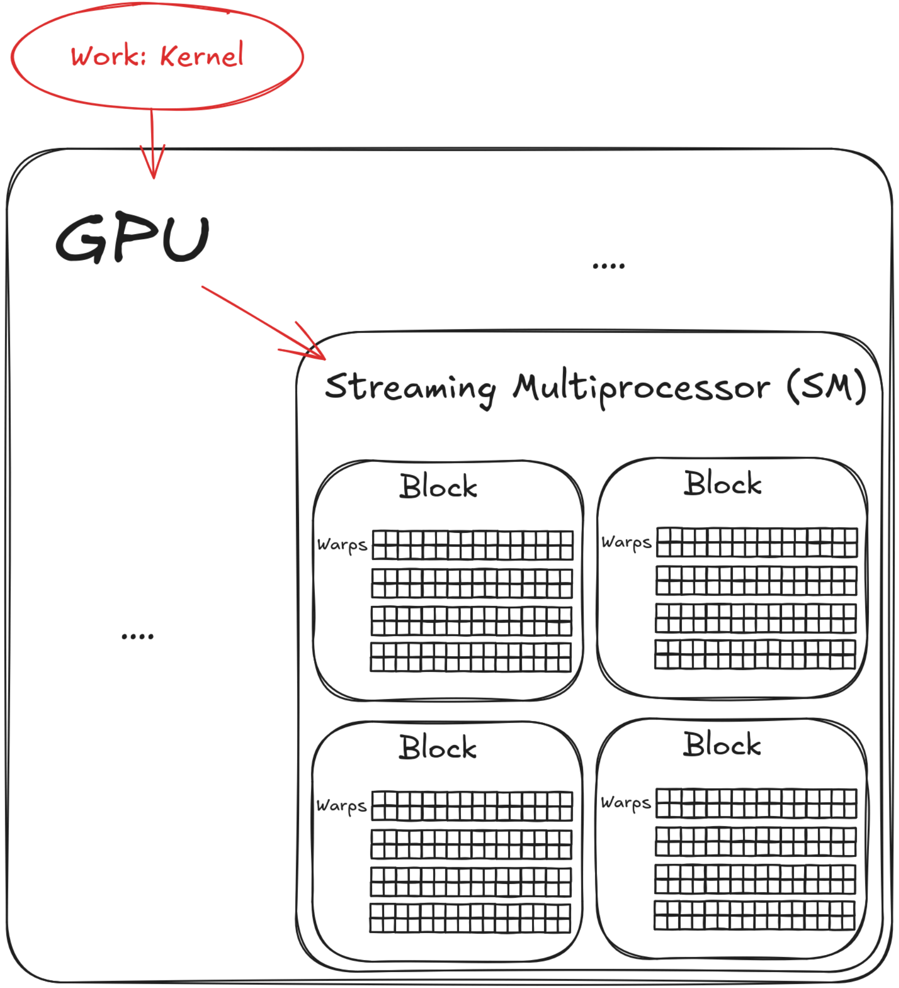
- GPUs have $\mathcal{O}(10)$ *Streaming multiprocessors*.
  - SMs execute *kernels* with series of parallel instructions.
  - SMs schedule their execution.
- Work given to SMs in *blocks*.
  - A block has to fit onto a SM's hardware capabilities (~1024 threads/block).
  - Each block's (sub-)contexts are persistent throughout its execution.
- Internally, blocks are subdivided into *warps*.



GPU architecture

GPU thread hierarchy

- GPUs have $\mathcal{O}(10)$ *Streaming multiprocessors*.
  - SMs execute *kernels* with series of parallel instructions.
  - SMs schedule their execution.
- Work given to SMs in *blocks*.
  - A block has to fit onto a SM's hardware capabilities (~1024 threads/block).
  - Each block's (sub-)contexts are persistent throughout its execution.
- Internally, blocks are subdivided into *warps*.
  - Each warp runs a single instruction in a *kernel* in parallel.
  - Warp size is always 32 for Nvidia, 32 or 64 for AMD.



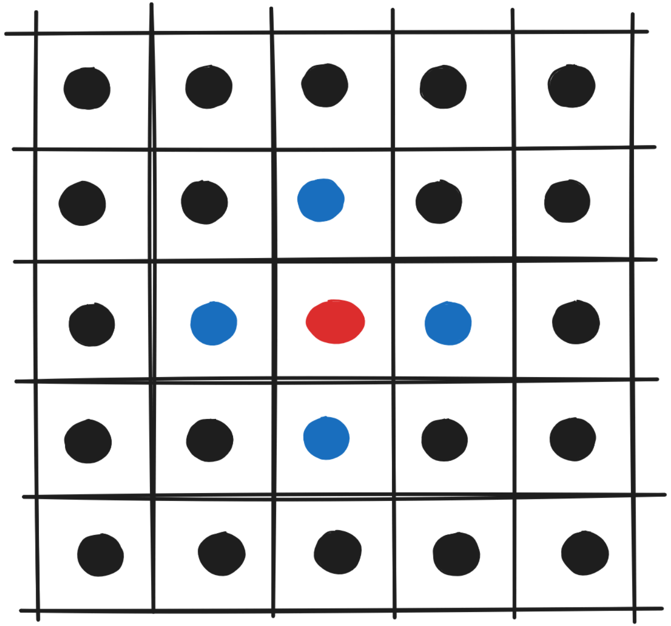
Memory access patterns

Coalescing vs. sequential access

Example: Solving massless Klein-Gordon equation in $d = 3$,

$$\partial_t^2 \phi(t, x) = \Delta \phi(t, x)\,.$$

- Calculation of 1 thread at **red site**.
- **Blue sites** dependents for lattice Laplacian.



Memory access patterns

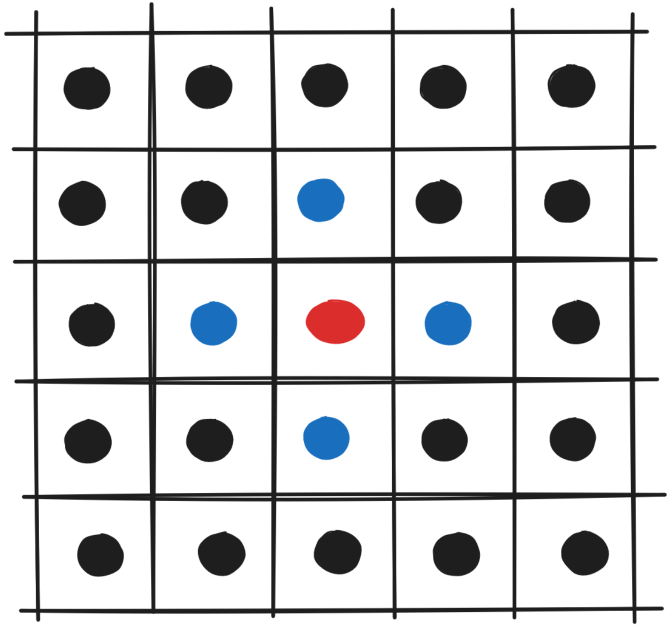
Coalescing vs. sequential access

Example: Solving massless Klein-Gordon equation in $d = 3$,

$$\partial_t^2 \phi(t, x) = \Delta \phi(t, x) \,.$$

- Calculation of 1 thread at **red site**.
- **Blue sites** dependents for lattice Laplacian.

What is the optimal way to iterate over sites?



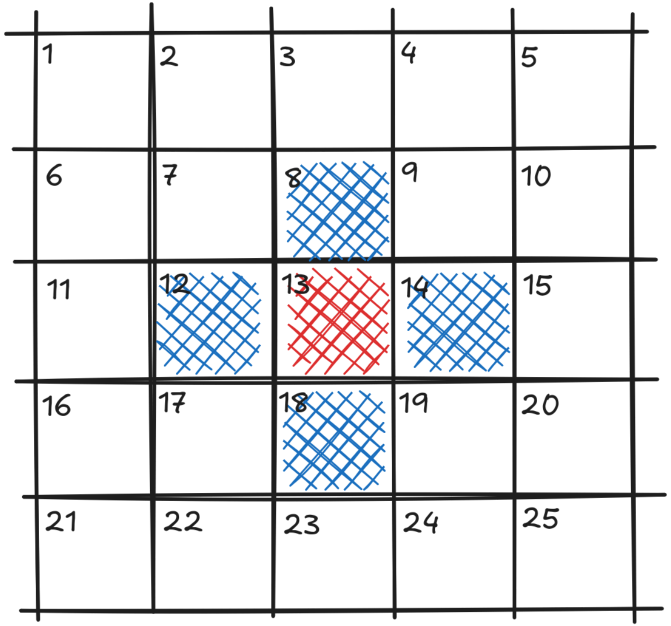
Memory access patterns

Coalescing vs. sequential access

Example: Solving massless Klein-Gordon equation in $d = 3$,

$$\partial_t^2 \phi(t, x) = \Delta \phi(t, x) \, .$$

- Calculation of 1 thread at **red site**.
- **Blue sites** dependents for lattice Laplacian.
- Memory is ordered row-major.



What is the optimal way to iterate over sites?

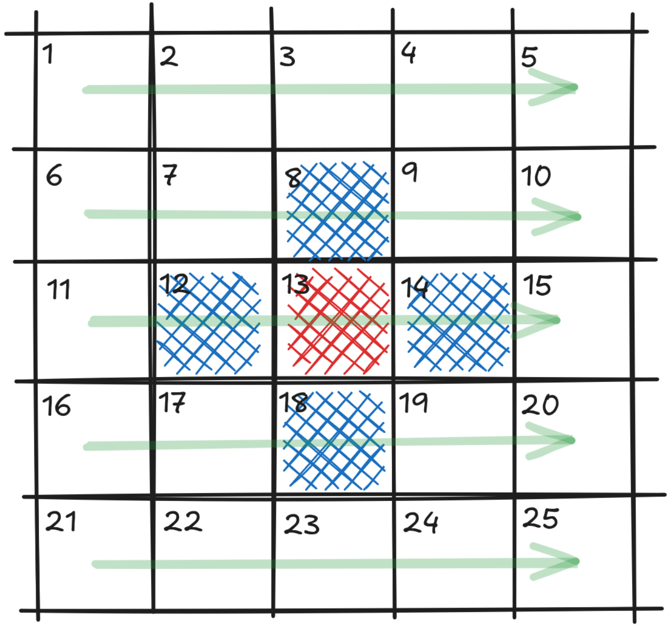
Memory access patterns

Coalescing vs. sequential access

Example: Solving massless Klein-Gordon equation in $d = 3$,

$$\partial_t^2 \phi(t, x) = \Delta \phi(t, x) \, .$$

- Calculation of 1 thread at **red site**.
- **Blue sites** dependents for lattice Laplacian.
- Memory is ordered row-major.



What is the optimal way to iterate over sites?

CPU: **Cashed access pattern** allows for caching of subsequent operations of a single thread.

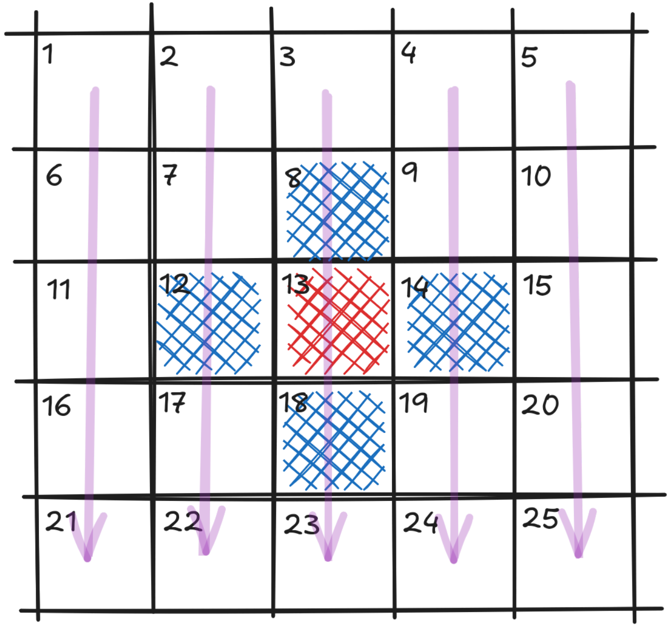
Memory access patterns

Coalescing vs. sequential access

Example: Solving massless Klein-Gordon equation in $d = 3$,

$$\partial_t^2 \phi(t, x) = \Delta\phi(t, x)\,.$$

- Calculation of 1 thread at **red site**.
- **Blue sites** dependents for lattice Laplacian.
- Memory is ordered row-major.



What is the optimal way to iterate over sites?

CPU: **Cashed access pattern** allows for caching of subsequent operations of a single thread.

GPU: **Coalesced access pattern** allows for simultaneous reading of memory for multiple threads.

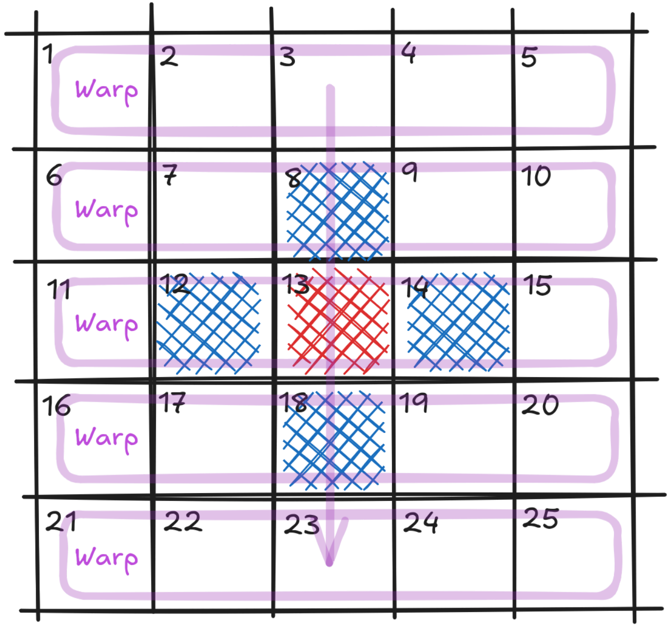
Memory access patterns

Coalescing vs. sequential access

Example: Solving massless Klein-Gordon equation in $d = 3$,

$$\partial_t^2 \phi(t, x) = \Delta \phi(t, x)\,.$$

- Calculation of 1 thread at **red site**.
- **Blue sites** dependents for lattice Laplacian.
- Memory is ordered row-major.



What is the optimal way to iterate over sites?

CPU: **Cashed access pattern** allows for caching of subsequent operations of a single thread.

GPU: **Coalesced access pattern** allows for simultaneous reading of memory for multiple threads.

This is similar to vectorization on a CPU!

SIMD (Single Instruction, Multiple Data) vs **SIMT** (Single Instruction, Multiple Threads)

Memory access patterns

Coalesced vs. cached access

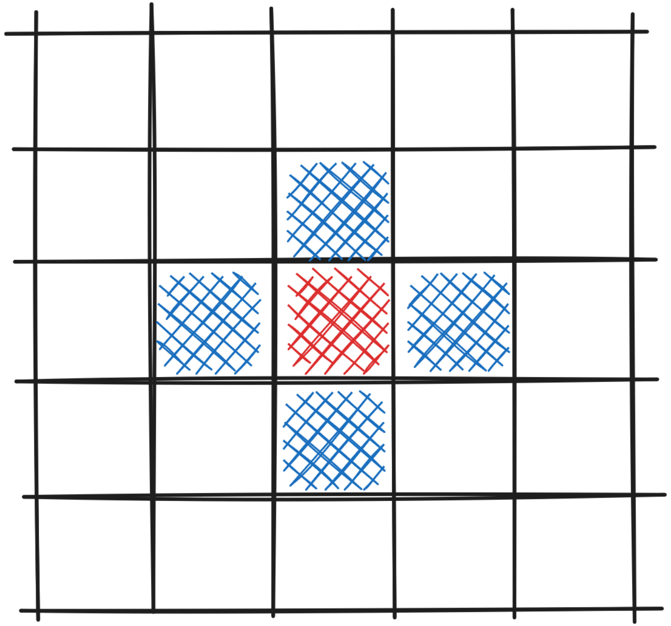
Example: Solving massless Klein-Gordon equation in $d = 3$,

$$\partial_t^2 \phi(t, x) = \Delta \phi(t, x) \, .$$

- Calculation of 1 thread at **red site**.
- **Blue sites** dependents for lattice Laplacian.
- Memory is ordered row-major.

CPU: Prefer **prefer row-major access pattern**.

GPU: Prefer **column-major access pattern**.



How does this perform in vivo?

```cpp
#define FORCE_ACCESS_PATTERN 0 // or 1
...

int main(int argc, char **argv)
{
  constexpr size_t NDim = 3;
  using T = double;
  constexpr size_t nGrid = 512;
  constexpr size_t nGhost = 1;
  constexpr size_t nSteps = 512;
  constexpr T dt = 0.01;
  ...
  Field<NDim, T> phi("phi", toolBox);
  Field<NDim, T> pi("pi", toolBox);

  Benchmark bench([&](Benchmark::Measurer &measurer) {
    phi.inFourierSpace() = RandomGaussianField<NDim, T>("Rand", toolBox);
    pi.inFourierSpace() = RandomGaussianField<NDim, T>("Rand2", toolBox);

    for (size_t i = 0; i < nSteps; ++i) {
      pi.updateGhosts();
      device::iteration::fence();
      measurer.measure("timestepping", [&]() {
        pi = pi + dt * LatticeLaplacian<NDim, decltype(phi)>(phi); // kick
        phi = phi + dt * pi;                                        // drift
        device::iteration::fence();
      });
```

```cpp
...
    for (size_t i = 0; i < nSteps; ++i) {
      pi.updateGhosts();
      device::iteration::fence();
      measurer.measure("timestepping", [&]() {
        pi = pi + dt * LatticeLaplacian<NDim, decltype(phi)>(phi); // kick
        phi = phi + dt * pi;                                       // drift
        device::iteration::fence();
      });
    }
```

Running this on my PC:

GPU: Nvidia 4070RTX mobile - 4788 Cores @ 2.175 GHz

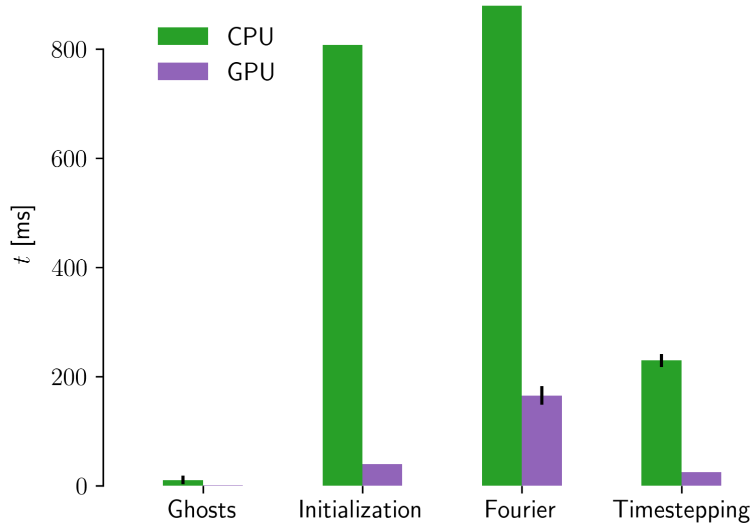
CPU: Ryzen 9 7945HX - 16 Cores @ 5.4GHz

Taking a closer look

```
...
Benchmark bench([&](Benchmark::Measurer &measurer) {
  measurer.measure("x->k fourier", [&]() {
    phi.getMemoryManager()->confirmFourierSpace();
    pi.getMemoryManager()->confirmFourierSpace();
  });

  measurer.measure("initialize field", [&]() {
    phi.inFourierSpace() = RandomGaussianField<NDim, T>("Hoi", toolBox);
    pi.inFourierSpace() = RandomGaussianField<NDim, T>("Hai", toolBox);
  });

  measurer.measure("k->x fourier", [&]() {
    phi.getMemoryManager()->confirmConfigSpace();
    pi.getMemoryManager()->confirmConfigSpace();
  });

  for (size_t i = 0; i < nSteps; ++i) {
    measurer.measure("ghosts", [&]() {
      pi.updateGhosts();
      device::iteration::fence();
    });
    measurer.measure("timestepping", [&]() {
      pi = pi + dt * LatticeLaplacian<NDim, decltype(phi)>(phi);
```

Taking a closer look

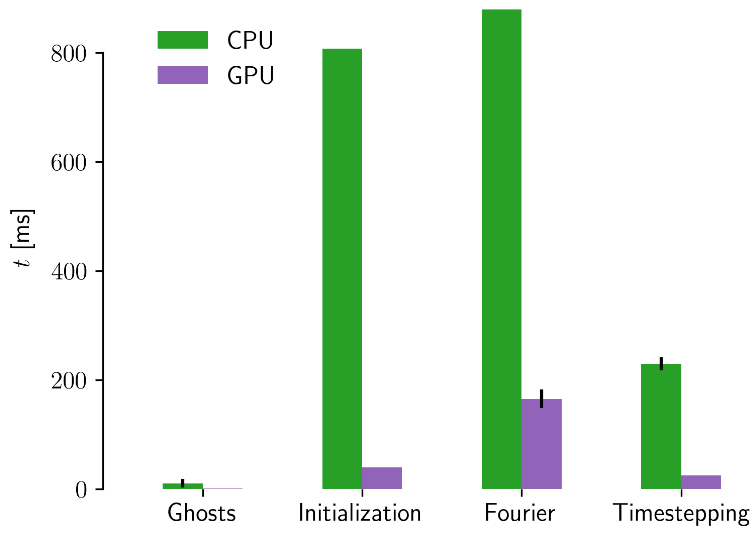
```
...
Benchmark bench([&](Benchmark::Measurer &measurer) {
  measurer.measure("x->k fourier", [&]() {
    phi.getMemoryManager()->confirmFourierSpace();
    pi.getMemoryManager()->confirmFourierSpace();
  });

  measurer.measure("initialize field", [&]() {
    phi.inFourierSpace() = RandomGaussianField<NDim, T>("Hoi", toolBox);
    pi.inFourierSpace() = RandomGaussianField<NDim, T>("Hai", toolBox);
  });

  measurer.measure("k->x fourier", [&]() {
    phi.getMemoryManager()->confirmConfigSpace();
    pi.getMemoryManager()->confirmConfigSpace();
  });

  for (size_t i = 0; i < nSteps; ++i) {
    measurer.measure("ghosts", [&]() {
      pi.updateGhosts();
      device::iteration::fence();
    });
    measurer.measure("timestepping", [&]() {
      pi = pi + dt * LatticeLaplacian<NDim, decltype(phi)>(phi);
```



CPU on MPI, GPU on CUDA

Taking a closer look

```
...
Benchmark bench([&](Benchmark::Measurer &measurer) {
  measurer.measure("x->k fourier", [&]() {
    phi.getMemoryManager()->confirmFourierSpace();
    pi.getMemoryManager()->confirmFourierSpace();
  });

  measurer.measure("initialize field", [&]() {
    phi.inFourierSpace() = RandomGaussianField<NDim, T>("Hoi", toolBox);
    pi.inFourierSpace() = RandomGaussianField<NDim, T>("Hai", toolBox);
  });

  measurer.measure("k->x fourier", [&]() {
    phi.getMemoryManager()->confirmConfigSpace();
    pi.getMemoryManager()->confirmConfigSpace();
  });

  for (size_t i = 0; i < nSteps; ++i) {
    measurer.measure("ghosts", [&]() {
      pi.updateGhosts();
      device::iteration::fence();
    });
    measurer.measure("timestepping", [&]() {
      pi = pi + dt * LatticeLaplacian<NDim, decltype(phi)>(phi);
```

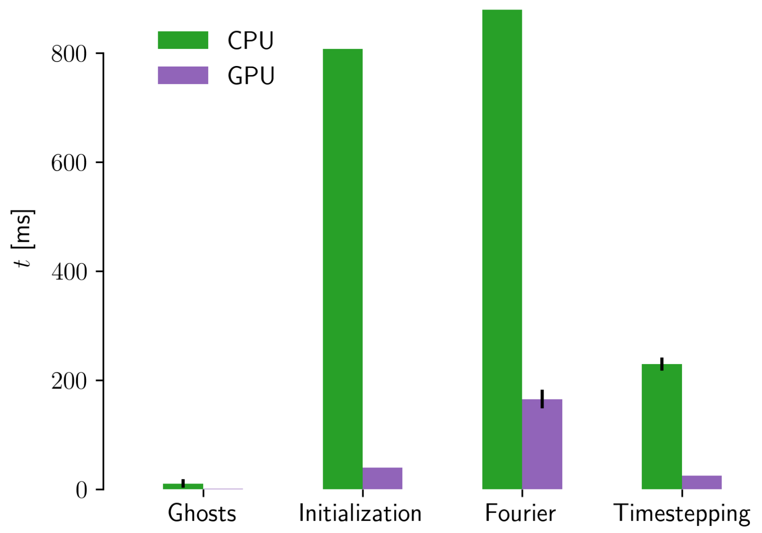


CPU on MPI, GPU on CUDA

- Fourier transformation: `cuFFT`
  (automatic switch to GPU native FFTs)

Taking a closer look

```
...
Benchmark bench([&](Benchmark::Measurer &measurer) {
  measurer.measure("x->k fourier", [&]() {
    phi.getMemoryManager()->confirmFourierSpace();
    pi.getMemoryManager()->confirmFourierSpace();
  });

  measurer.measure("initialize field", [&]() {
    phi.inFourierSpace() = RandomGaussianField<NDim, T>("Hoi", toolBox);
    pi.inFourierSpace() = RandomGaussianField<NDim, T>("Hai", toolBox);
  });

  measurer.measure("k->x fourier", [&]() {
    phi.getMemoryManager()->confirmConfigSpace();
    pi.getMemoryManager()->confirmConfigSpace();
  });

  for (size_t i = 0; i < nSteps; ++i) {
    measurer.measure("ghosts", [&]() {
      pi.updateGhosts();
      device::iteration::fence();
    });
    measurer.measure("timestepping", [&]() {
      pi = pi + dt * LatticeLaplacian<NDim, decltype(phi)>(phi);
```



CPU on MPI, GPU on CUDA

- Fourier transformation: `cuFFT`
  (automatic switch to GPU native FFTs)

Benchmarking -theory

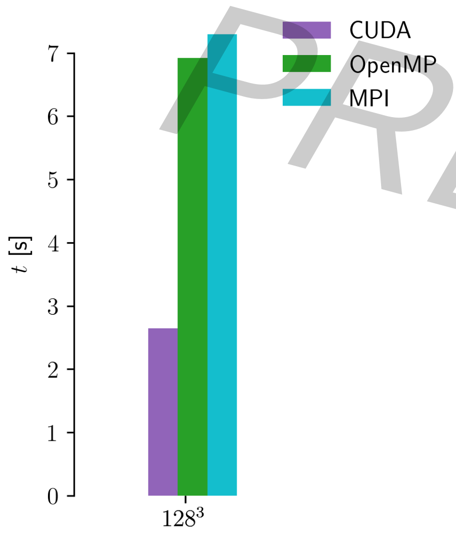
with the `lphi4` model in CosmoLattice

Benchmarking -theory

with the `lphi4` model in CosmoLattice

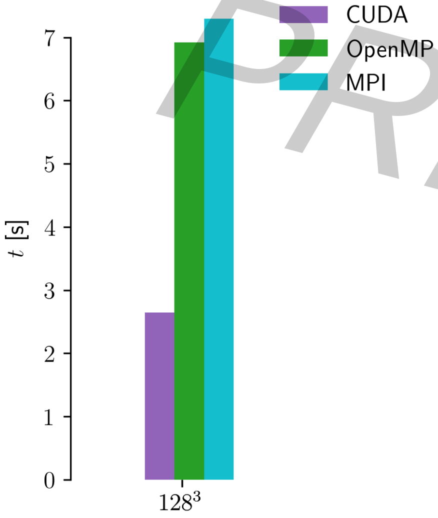
Benchmarking -theory

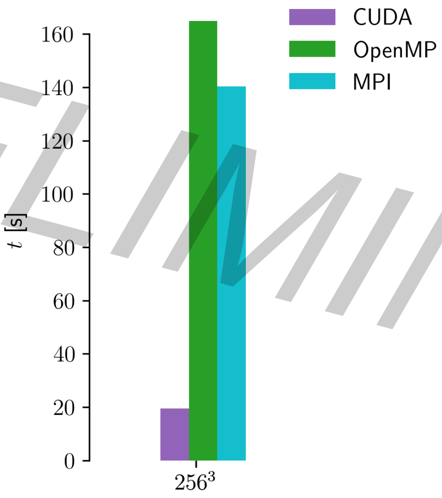
with the `lphi4` model in CosmoLattice



Benchmarking -theory

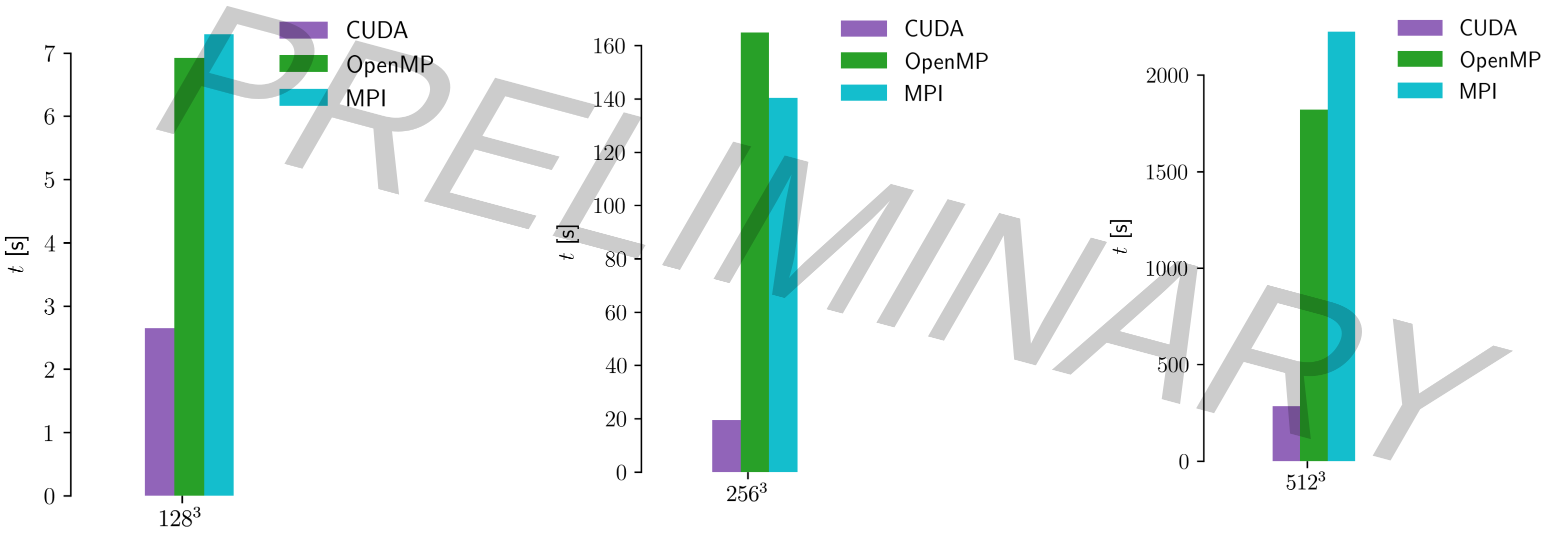
with the `lphi4` model in CosmoLattice





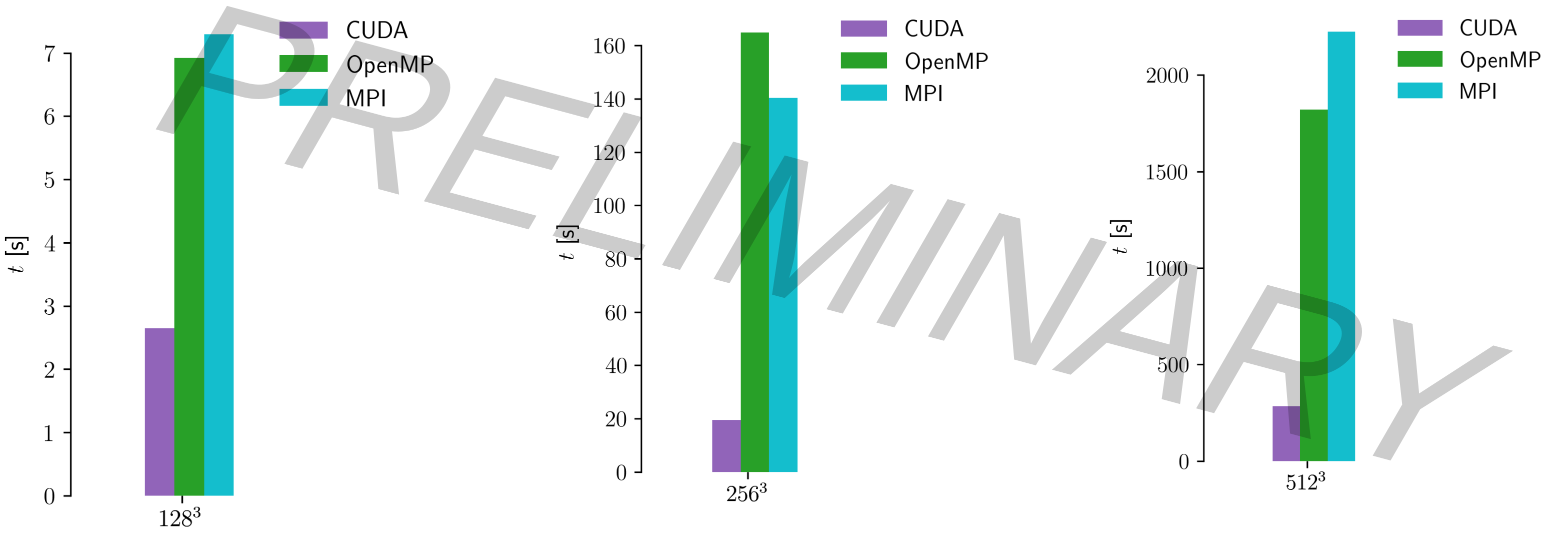
Benchmarking -theory

with the `lphi4` model in CosmoLattice



Benchmarking -theory

with the `lphi4` model in CosmoLattice



	CUDA runtime [s]	OpenMP runtime [s]	MPI runtime [s]	speedup CUDA/MPI factor
N=128	2.6	6.9	7.3	2.8
N=256	19.5	164.9	140.4	7.2
N=512	283.1	1820.6	2224.7	7.9

Slightly unfair comparison
(my CPU is "stronger")

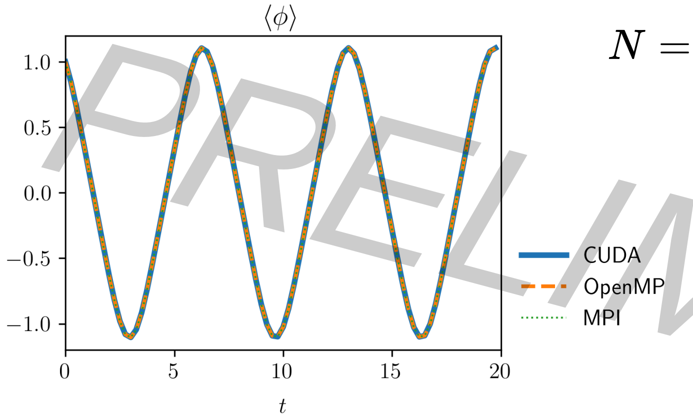
Benchmarking -theory

with the `lphi4` model in CosmoLattice

$N = 512$

Benchmarking -theory

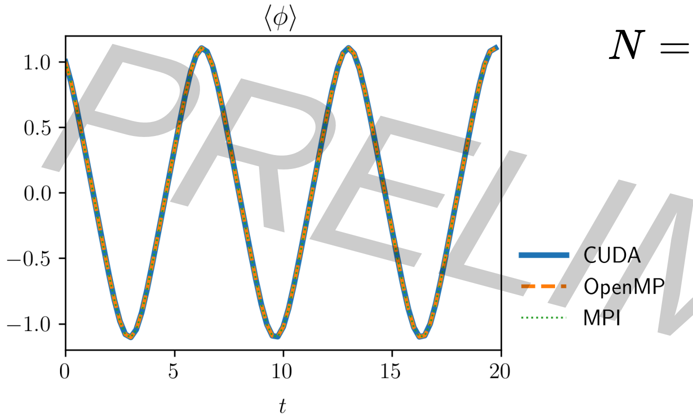
with the `lphi4` model in CosmoLattice



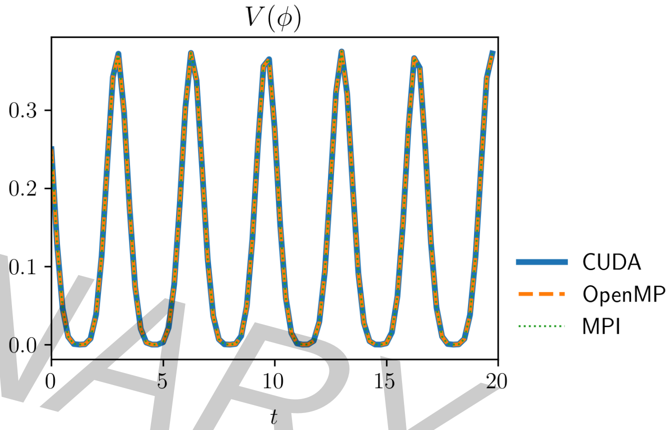
$$N = 512$$

Benchmarking -theory

with the `lphi4` model in CosmoLattice

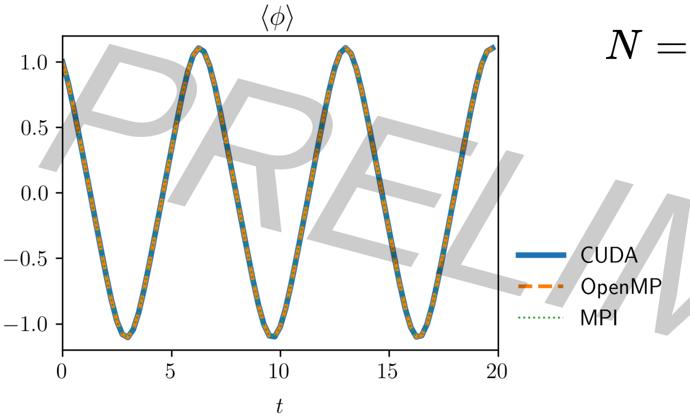


$$N = 512$$

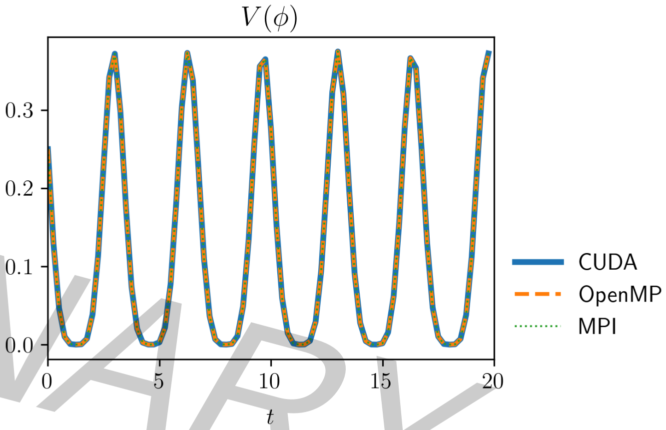


Benchmarking -theory

with the `lphi4` model in CosmoLattice



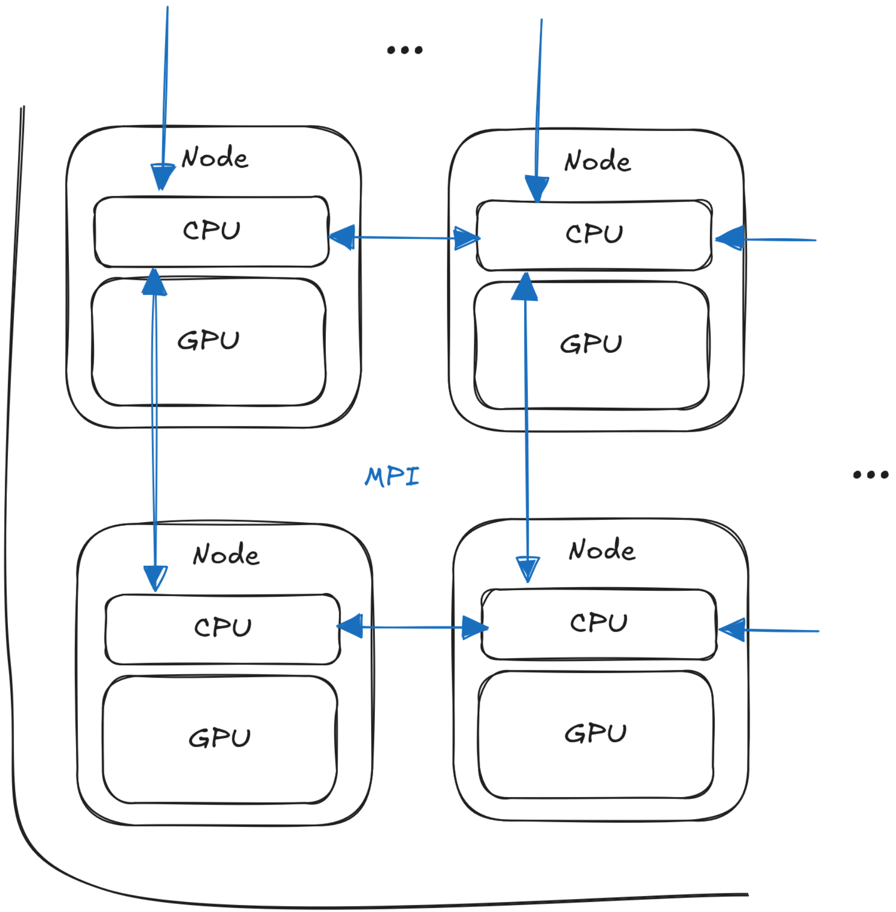
$$N = 512$$



Scaling it up

Using large GPU clusters

- To use large clusters and link up many nodes, CosmoLattice uses the **Message-Passing Interface (MPI)** (see lecture yesterday).
- Send data in RAM (e.g. ghosts) between neighbouring nodes.

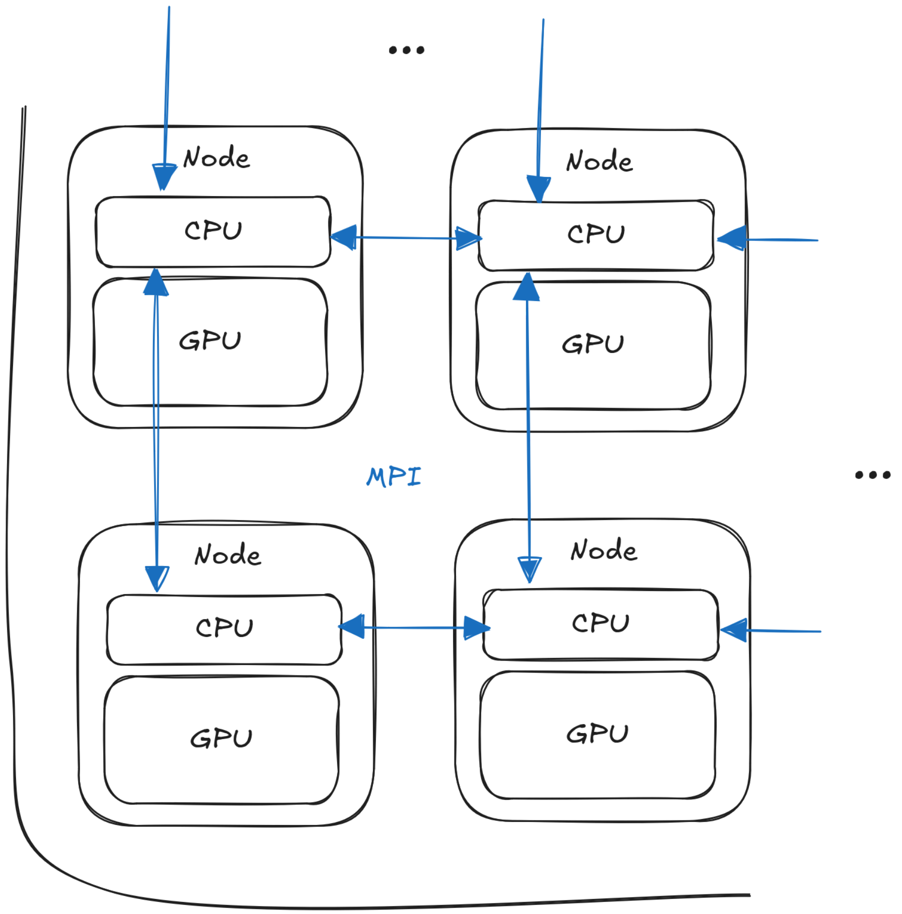


Scaling it up

Using large GPU clusters

- To use large clusters and link up many nodes, CosmoLattice uses the **Message-Passing Interface (MPI)** (see lecture yesterday).
- Send data in RAM (e.g. ghosts) between neighbouring nodes.

What about MPI+GPUs?



Scaling it up

Using large GPU clusters

- To use large clusters and link up many nodes, CosmoLattice uses the **Message-Passing Interface (MPI)** (see lecture yesterday).

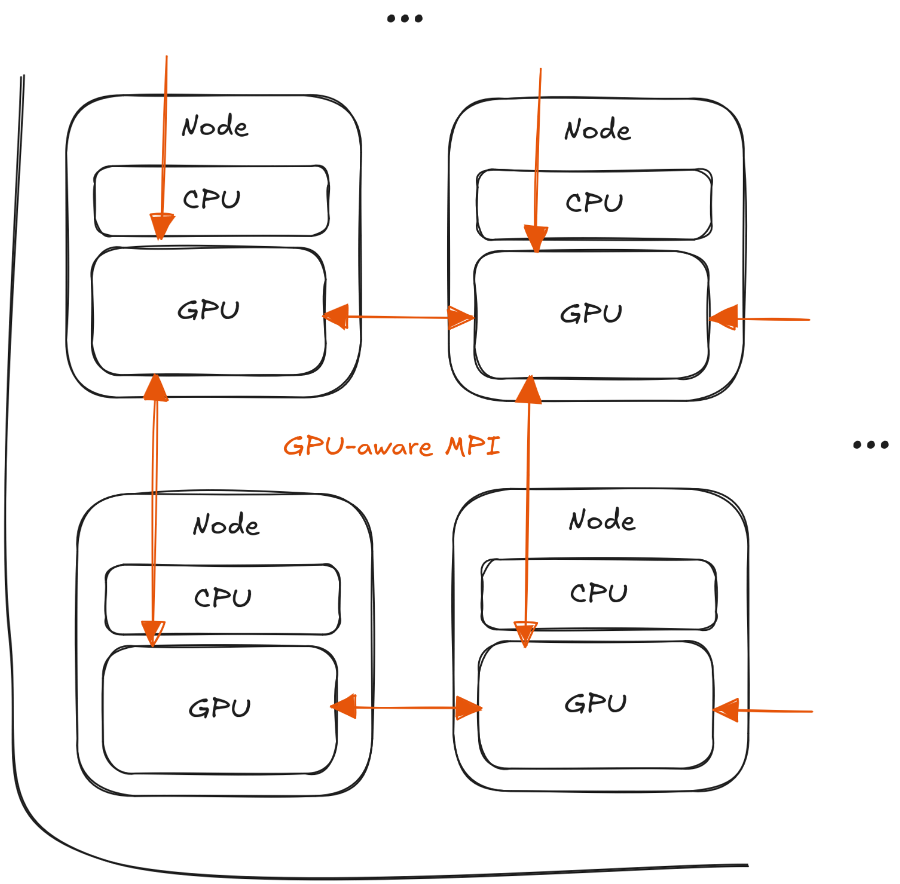
- Send data in RAM (e.g. ghosts) between neighbouring nodes.

What about MPI+GPUs?

- **GPU-aware MPI** can exchange data directly between device memory.

  Support since before 2013:
  - OpenMPI
  - MVAPICH2
  - Cray MPI
  - IBM MPI

- No changes in MPI-code! except in FFT code…



There's more...

Useful, more fine-grained parallelization for even more speedup in the future:

There’s more…

Useful, more fine-grained parallelization for even more speedup in the future:

- Hierarchical parallelism.

  split complex tasks into smaller work-teams with the potential for caching

There's more…

Useful, more fine-grained parallelization for even more speedup in the future:

- Hierarchical parallelism.

  split complex tasks into smaller work-teams with the potential for caching

- Block- and Warp- local caching.

  reduce the number of repeated operations

There's more…

Useful, more fine-grained parallelization for even more speedup in the future:

- Hierarchical parallelism.
  split complex tasks into smaller work-teams with the potential for caching
- Block- and Warp- local caching.
  reduce the number of repeated operations
- Heterogeneous work-balancing.
  e.g. use CPU for measurements while the GPU continues evolving

There's more…

Useful, more fine-grained parallelization for even more speedup in the future:

- Hierarchical parallelism.
  split complex tasks into smaller work-teams with the potential for caching
- Block- and Warp- local caching.
  reduce the number of repeated operations
- Heterogeneous work-balancing.
  e.g. use CPU for measurements while the GPU continues evolving
- …

Questions?

Thanks for your attention!

Release of CosmoLattice with GPUs ~ early 2026