

# HILA lattice simulation framework

Kari Rummukainen

University of Helsinki and Helsinki Institute of Physics

**hila** : (noun) *grid, grating, lattice, (archaic) gate, (cricket) wicket*



HILA system authors (so far): K.R, Jarno Rantaharju, Aaron Haarti, José Ricardo Correia  
Application authors: Asier Lopez Eiguren, Eelis Mielonen, Lauri Niemi, Jaakko Hällfors,

## Oh no – not another framework!

- **Lattice:** regular  $d$ -dimensional grid of points, with **Fields** logically living on points / links / faces etc.
- Interactions (nearest-) neighbour, boundary conditions usually periodic but sometimes something else
- Many applications map to this, at Helsinki:
  - ▶ QCD
  - ▶ exotic gauge-fermion systems
  - ▶ many 3d effective theories of BSM models at high temperature
  - ▶ relativistic hydrodynamics @ cosmology
  - ▶ cosmic string network evolution
- *Well-known problem: accelerators, vector instructions etc. – hard to use, custom codes needed to get the full benefit.*
- Nevertheless: lattice simulations are very regular – should be possible to generate optimised code automatically?
- (Old Helsinki code: C preprocessor macros  $\mapsto$  CPU/CUDA code)

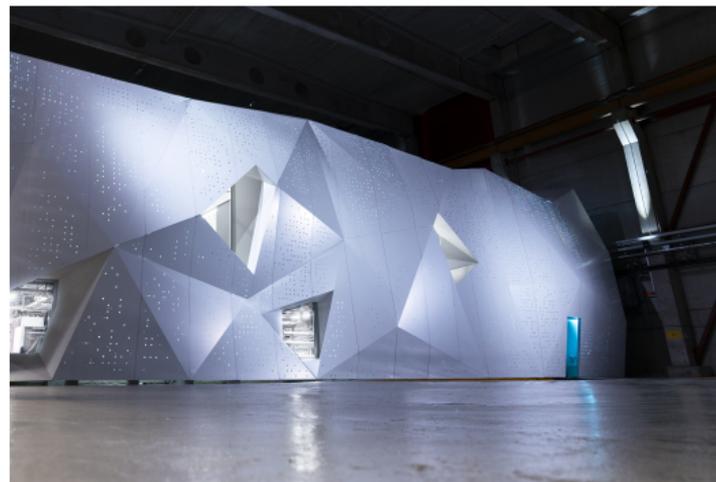
→ HILA lattice framework

# Oh no! Not another framework!

- Some solutions:
  - ▶ Directive-based approach (OpenMP, OpenACC) - not general enough, not powerful enough
  - ▶ C++ template metaprogramming library (GRID)
  - ▶ Domain-specific languages
  - ▶ HILA: custom preprocessor/code generator + template library (C++)
- Goals of HILA:
  - ▶ Application programs run everywhere without modifications
  - ▶ Writing applications “as easy as possible”
    - ★ Easy to experiment with new code
    - ★ Students
  - ▶ Strict separation between the application layer and the platform/backend layer
  - ▶ Optimized code generation – CUDA, HIP, AVX2, AVX512, OpenMP ...
  - ▶ Excellent scaling & high performance
  - ▶ When ported to new architectures, only the framework is updated, no need to update any of the applications

Marketing slogan: *Write once – run everywhere*

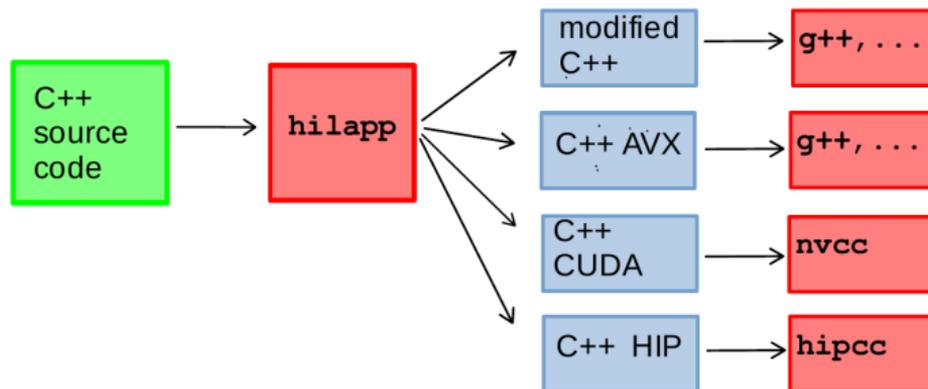
# LUMI supercomputer



- Hosted by CSC, Finnish IT center for science
- HPE Cray EX, Slingshot 11 – # 3 on Top 500 6/22 (LUMI was still incomplete)
- Large CPU partition LUMI-C and even larger GPU partition LUMI-G
- LUMI-G: 2560 nodes $\times$ 4 $\times$ AMD MI250X ( $\sim$  Frontier/4), theoretical speed  $\sim$  0.5 EFlops (float/double “vectors”)

# How does HILA work?

C++ source-to-source preprocessor, hilapp + hila library



- 2-stage compilation (taken care of by the HILA build system)
- `hilapp` is built using the *libtooling* interface to the `clang++` open source compiler<sup>1</sup> front-end. <https://clang.llvm.org/docs/LibTooling.html>
- Gives full “compiler view” to the program, AST (Abstract Syntax Tree), and tools to rewrite the source

<sup>1</sup>For short introduction, see

<https://eli.thegreenplace.net/2014/05/01/modern-source-to-source-transformation-with-clang-and-libtooling/>

# Why?

- Why C++? It is mature and everywhere, and has decent support for generic programming (templates).
- Why not template metaprogramming? Very difficult for complicated transformations.
- Why clang-based hilapp preprocessor?
  - ▶ To transform code, need to know variables, expression types etc → compiler does it for you.
  - ▶ Full clang front-end → nice error reporting.
  - ▶ Tooling exists.
  - ▶ Scratch an itch ...
- C++17 standard.
  
- HILA is available at <https://github.com/CFT-HY/HILA> (GPL2, MPL)

# Hello world

```
#include "hila.h"

int main(int argc, char *argv[]) {

    hila::initialize(argc, argv);
    lattice.setup({32, 32, 32});
    hila::seed_random(32345);

    // complex field f, make it Gaussian
    Field<Complex<double>> f;

    onsites(ALL) f[X].gaussian_random();

    // Measure hopping term and f^2
    Complex<double> hopping = 0;
    double fsqr = 0;

    onsites(ALL) {
        foralldir(d) {
            hopping += f[X] * f[X + d].conj();
        }
        fsqr += f[X].squarenorm();
    }

    hila::out0 << "Average f^2 : " << fsqr / lattice.volume() << '\n';
    hila::out0 << "Average hopping term " << hopping / (NDIM*lattice.volume()) << '\n';

    hila::finishrun();
    return 0;
}
```

# Traversing the lattice

“Site loop”: `onsites (parity) { ... }` where *parity* is of Parity type expression: EVEN, ODD or ALL

Example: plaquette

```
Field<SU<3,double>> U[NDIM];
...
double plaqsum = 0;
foralldir(d1) foralldir(d2) if (d1 < d2) {
    onsite(ALL) {
        auto p = U[d1][X] * U[d2][X+d1] * U[d1][X+d2].dagger() * U[d2][X].dagger();
        plaqsum += p.trace().real();
    }
}
```

- $X$  : current location symbol
- $[X \pm d]$  : nearest neighbour (can do  $[X + d1 - d2]$ ,  $[X + 3*d1]$  ...)
- *Reduction*: += (or \*=) to a loop extern variable
- MPI communications done if needed (remembers gathers)

# HILA Types

- C++ native types: `float`, `double`, `long double`, `int` ...
- `Complex<T>`, where T is one of the above types (do not use C++ complex type)
- Basic built-in types: `Matrix<n,m,S>`, `SquareMatrix<n,S>`, `Vector<n,S>`, `Array<n,m,S>`, ...  
where S is C++ native or `Complex` type
- Derived from basic types: `SU<N>`, `Algebra<SU<N>>`, `WilsonVector`, `GammaMatrix` ...  
Simple enough to create new types
- Infrastructure types: `Parity`, `Direction`, `CoordinateVector`

Example: `SquareMatrix<3,Complex<double>>`

Lattice fields: `Field<type>`, where *type* one of the above types

- `Field<double>`
- `Field<Vector<3,Complex<double>>>`

Basic field traversal:

```
onsites(par) a[X] = b[X] + 3 * c[X];
```

Equivalent form for 1-liner assignments:

```
a[par] = b[X] + 3 * c[X];
```

This achieves the same but less efficiently (if par = ALL)

```
a = b + 3 * c;
```

Handy for initialization, e.g.

```
a = 0;
```

onsites()-loop can contain variable definitions, if/for/while -statements, function calls

```
onsites(par) {  
    if (X.coordinate(e_x) > lattice.size(e_x)/2)  
        sum += a[X];  
}
```

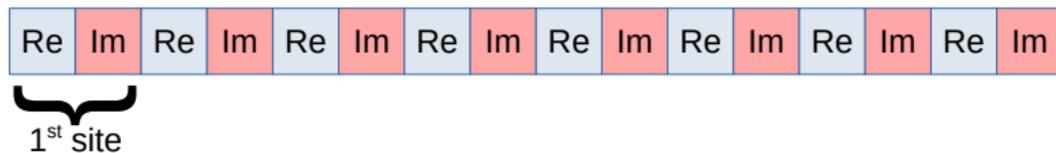
Single site access (here in 3d):

```
CoordinateVector p1 = {0,0,0}, p2 = {2,3,4};  
a[p1] += b[p2];
```

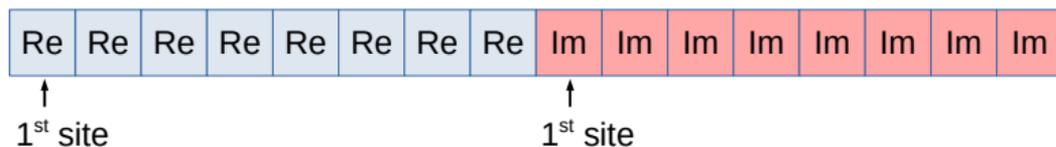
# Field layout

Example: `Field<Complex<double>>`

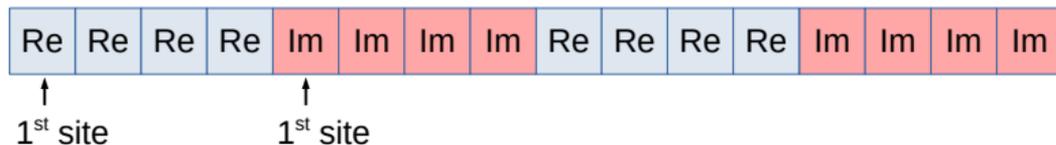
- On “vanilla” architectures stored as array-of-structs:



- On GPUs struct-of-arrays:



- On vector targets array-of-structs-of-arrays, on AVX2 in groups of 4 doubles or 8 floats:



AVX layout uses “virtual nodes” within MPI rank (as in GRID)

Vectorized operations implemented using the VCL library by Agner Fog

<https://github.com/vectorclass/version2>

# Code generation

- Architecture chosen by `make` switch (always includes MPI):
  - ▶ `make ARCH=vanilla`
  - ▶ `make ARCH=AVX2`
  - ▶ `make ARCH=cuda`
  - ▶ `make ARCH=openmp` (hybrid MPI/OpenMP)
  - ▶ `make ARCH=lumi-hip` (tailored make stub for supers) ...
- GPU targets:
  - ▶ Every `onsites()` -loop becomes a CUDA/HIP kernel
  - ▶ All field operations are on GPUs – Fields are not copied to cpu except when reading/writing.
  - ▶ GPU-aware MPI (but can do also non-GPU-aware)
  - ▶ Own GPU memory manager
- AVX targets:
  - ▶ `hilapp` analyses `onsites()` -loops for vectorised or site-by-site access.
- Scales to very large volumes, max number of sites  $\sim 2^{63}$ . Main limitation:  
(lattice sites)/(MPI process)  $< 2^{32} \approx 4.2 \times 10^9$

# Makefile stub

Every application program needs makefile stub (gmake):

```
# Give the location of the top level distribution directory wrt. this location.
# Can be absolute or relative
HILA_DIR := path/to/HILA

# Set default goal and arch
.DEFAULT_GOAL := my_own_program

ifndef ARCH
ARCH := vanilla
endif

# Read in the main makefile contents, incl. platforms
include $(HILA_DIR)/libraries/main.mk

APP_OPTS += -DNDIM=3

# With multiple targets we want to use "make target", not "make build/target".
# This is needed to carry the dependencies to build-subdir
my_own_program: build/my_own_program ; @:

# Now the linking step for each target executable
build/my_own_program: Makefile build/my_own_program.o $(HILA_OBJECTS) $(HEADERS)
    $(LD) -o $$@ build/my_own_program.o $(HILA_OBJECTS) $(LDFLAGS) $(LDLIBS)
```

## What hilapp does on GPU targets?

```
using myvector = Vector<3,Complex<double>>;
Field<myvector> a,b,c;
Complex<double> coeff = 3.14 + 2*I;
...
double n = 0;
onsites(par) {
    a[X] += b[X] + coeff * c[X + e_x];
    n += norm(a[X]);
}
```

# What hilapp does on GPU targets?

```
using myvector = Vector<3,Complex<double>>;  
Field<myvector> a,b,c;  
Complex<double> coeff = 3.14 + 2*I;  
...  
double n = 0;  
onsites(par) {  
    a[X] += b[X] + coeff * c[X + e_x];  
    n += norm(a[X]);  
}
```

Field variables allocated on GPU

## What hilapp does on GPU targets?

```
using myvector = Vector<3,Complex<double>>;
Field<myvector> a,b,c;
Complex<double> coeff = 3.14 + 2*I;
...
double n = 0;
onsites(par) {
    a[X] += b[X] + coeff * c[X + e_x];
    n += norm(a[X]);
}
```

Field  $c$  is needed from  $\hat{e}_x$ -direction: initialize halo gather if not cached before (overlapping compute and communication)

## What hilapp does on GPU targets?

```
using myvector = Vector<3,Complex<double>>;
Field<myvector> a,b,c;
Complex<double> coeff = 3.14 + 2*I;
...
double n = 0;
onsites(par) {
    a[X] += b[X] + coeff * c[X + e_x];
    n += norm(a[X]);
}
```

Convert the whole `onsites()` -loop to a `__global__` function (GPU kernel). Replace the loop with a kernel call (possibly more than one kernel).

## What hilapp does on GPU targets?

```
using myvector = Vector<3,Complex<double>>;
Field<myvector> a,b,c;
Complex<double> coeff = 3.14 + 2*I;
...
double n = 0;
onsites(par) {
    a[X] += b[X] + coeff * c[X + e_x];
    n += norm(a[X]);
}
```

Convert functions, operators and constructors to `__host__ __device__` functions, if not done before.

## What hilapp does on GPU targets?

```
using myvector = Vector<3,Complex<double>>;
Field<myvector> a,b,c;
Complex<double> coeff = 3.14 + 2*I;
...
double n = 0;
onsites(par) {
    a[X] += b[X] + coeff * c[X + e_x];
    n += norm(a[X]);
}
```

Pass the addresses (on GPU memory) of the Field variables as kernel call arguments.

## What hilapp does on GPU targets?

```
using myvector = Vector<3,Complex<double>>;
Field<myvector> a,b,c;
Complex<double> coeff = 3.14 + 2*I;
...
double n = 0;
onsites(par) {
    a[X] += b[X] + coeff * c[X + e_x];
    n += norm(a[X]);
}
```

Pass the values of the non-field variables as kernel call arguments.

## What hilapp does on GPU targets?

```
using myvector = Vector<3,Complex<double>>;
Field<myvector> a,b,c;
Complex<double> coeff = 3.14 + 2*I;
...
double n = 0;
onsites(par) {
    a[X] += b[X] + coeff * c[X + e_x];
    n += norm(a[X]);
}
```

Insert accessors for Field variables *a*, *b* and *c* before using them in the kernel.

## What hilapp does on GPU targets?

```
using myvector = Vector<3,Complex<double>>;
Field<myvector> a,b,c;
Complex<double> coeff = 3.14 + 2*I;
...
double n = 0;
onsites(par) {
    a[X] += b[X] + coeff * c[X + e_x];
    n += norm(a[X]);
}
```

Because Field *a* changes, insert “de-accessor” (i.e. store the value back to field) for it.  
Also invalidate possible cached halos of *a*.

## What hilapp does on GPU targets?

```
using myvector = Vector<3,Complex<double>>;
Field<myvector> a,b,c;
Complex<double> coeff = 3.14 + 2*I;
...
double n = 0;
onsites(par) {
    a[X] += b[X] + coeff * c[X + e_x];
    n += norm(a[X]);
}
```

Loop contains **reduction**: reorganise the kernel so that the reduction can be done efficiently using `__shared__` GPU memory.

After the kernel completes, copy the result of the reduction to the host CPU.

Finally, do `MPI_Reduce()` across MPI processes to produce the final result.

# What hilapp does on GPU targets?

```
using myvector = Vector<3,Complex<double>>;
Field<myvector> a,b,c;
Complex<double> coeff = 3.14 + 2*I;
...
double n = 0;
onsites(par) {
    a[X] += b[X] + coeff * c[X + e_x];
    n += norm(a[X]);
}
```

If `onsites()` appears inside *template function* or *class*, `hilapp` specializes the templates explicitly as needed (this is not the case in this simple example).

Kernels are always generated as non-templated functions.

## What hilapp does on GPU targets - result:

```
const Parity _HILA_parity = (par);
Field<Matrix<3, 1, Complex<double>>> & _HILA_field_a = a;
const Field<Matrix<3, 1, Complex<double>>> & _HILA_field_b = b;
const Field<Matrix<3, 1, Complex<double>>> & _HILA_field_c = c;
dir_mask_t _dir_mask_ = 0;
_dir_mask_ |= _HILA_field_c.start_gather(e_x, _HILA_parity);
double r_HILA_var_n;
r_HILA_var_n = 0;
_HILA_field_c.wait_gather(e_x, _HILA_parity);
const lattice_struct & loop_lattice = lattice;
backend_lattice_struct lattice_info = *(lattice.backend_lattice);
lattice_info.loop_begin = lattice.loop_begin(_HILA_parity);
lattice_info.loop_end = lattice.loop_end(_HILA_parity);
int N_blocks = (lattice_info.loop_end - lattice_info.loop_begin + N_threads - 1)/N_threads;
double * dev_r_HILA_var_n;
gpuMalloc( (void **)& dev_r_HILA_var_n, sizeof(double) * N_blocks );
gpu_set_zero(dev_r_HILA_var_n, N_blocks);
_HILA_kernel_main_583<<< N_blocks, N_threads >>>( lattice_info, _HILA_field_a.fs->payload, _HILA_field_b.fs->payload,
_HILA_field_c.fs->payload, coeff, dev_r_HILA_var_n);

check_device_error("_HILA_kernel_main_583");
r_HILA_var_n = gpu_reduce_sum( dev_r_HILA_var_n, N_blocks);
gpuFree(dev_r_HILA_var_n);
if (hila::myrank() == 0) { n += r_HILA_var_n; }
else { n = r_HILA_var_n; }
hila_reduce_sum_setup( &n);
hila_reduce_sums();
hila::set_allreduce(true);
_HILA_field_a.mark_changed(_HILA_parity);
```

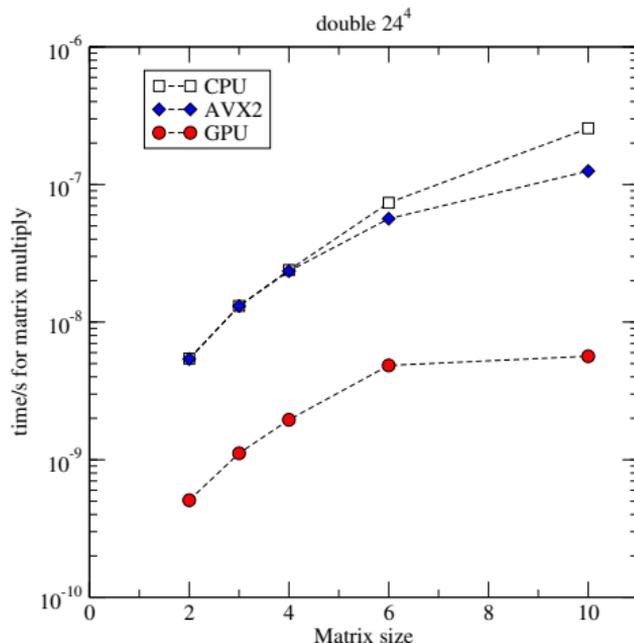
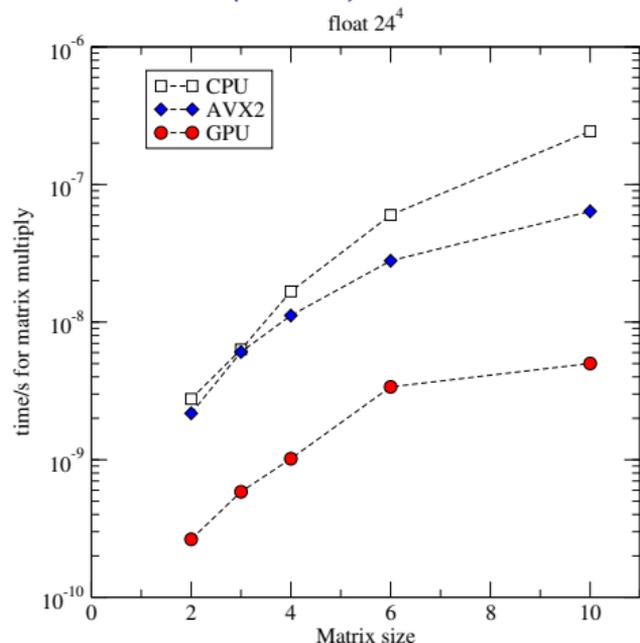
## What hilapp does on GPU targets - result:

```
inline __global__ void __launch_bounds__(N_threads) _HILA_kernel_main_583( backend_lattice_struct d_lattice,
    field_storage<Matrix<3, 1, Complex<double>>> _HILA_field_a,
    const field_storage<Matrix<3, 1, Complex<double>>> _HILA_field_b,
    const field_storage<Matrix<3, 1, Complex<double>>> _HILA_field_c,
    const Complex<double> kernel_par_0_, double * kernel_par_1_)
{
    unsigned _HILA_index = threadIdx.x + blockIdx.x * blockDim.x + d_lattice.loop_begin;
    __shared__ double kernel_par_1_sh[N_threads];
    double kernel_par_1_sum;
    kernel_par_1_sum = 0;
    kernel_par_1_sh[threadIdx.x] = 0;
    if(_HILA_index < d_lattice.loop_end) {
        Matrix<3, 1, Complex<double>> _HILA_field_a_at_X = _HILA_field_a.get(_HILA_index, d_lattice.field_alloc_size);
        Matrix<3, 1, Complex<double>> _HILA_field_b_at_X = _HILA_field_b.get(_HILA_index, d_lattice.field_alloc_size);
        Matrix<3, 1, Complex<double>> _HILA_field_c_dir1 = _HILA_field_c.get(_HILA_field_c.neighbours[e_x][_HILA_index], d_lattice.field_alloc_size);
        {
            _HILA_field_a_at_X += _HILA_field_b_at_X + kernel_par_0_ * _HILA_field_c_dir1;
            kernel_par_1_sum += norm(_HILA_field_a_at_X);
        }
        _HILA_field_a.set(_HILA_field_a_at_X, _HILA_index, d_lattice.field_alloc_size );
    }
    kernel_par_1_sh[threadIdx.x] = kernel_par_1_sum;
    __syncthreads();
    for( int _H_i=N_threads/2; _H_i>0; _H_i/=2 ){
        if(threadIdx.x < _H_i && _H_i +_HILA_index < d_lattice.loop_end) {
            kernel_par_1_sh[threadIdx.x] += kernel_par_1_sh[threadIdx.x+_H_i];
        }
        __syncthreads();
    }
    if(threadIdx.x == 0) { kernel_par_1[_blockIdx.x] = kernel_par_1_sh[0]; }
```

# Synthetic benchmark

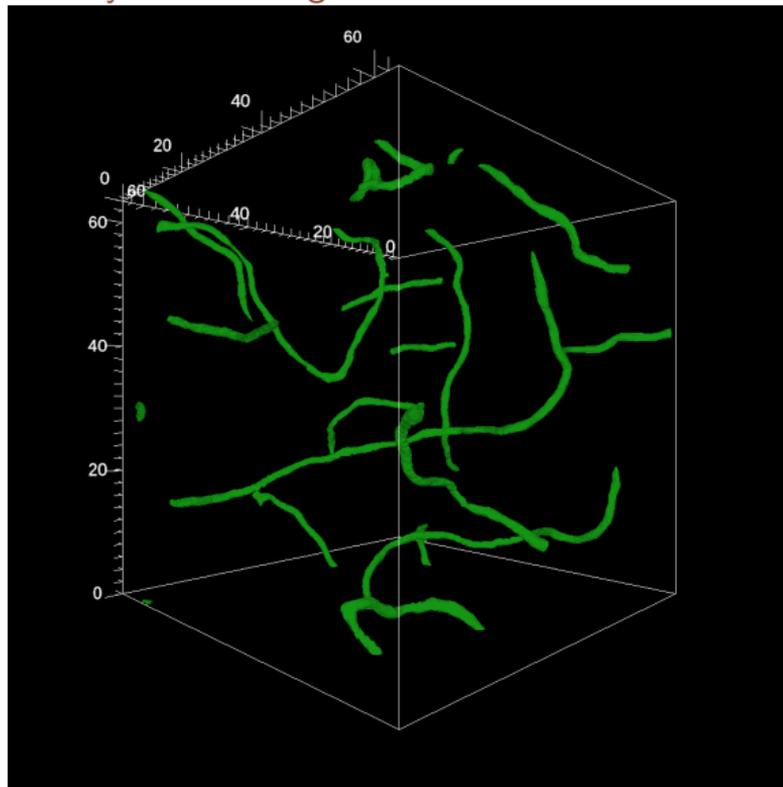
`onsites(ALL) G[X] = M[X] * M[X].dagger();` where  $G$  and  $M$  are  $n \times n$  complex matrix fields  
on  $24^4$  lattice (no communications).

*Intel Core I7-6700K (4 cores) vs. Nvidia GeForce GTX 1080*



# Axion string evolution

## Case study: axion string network evolution



- In essence, single complex field in expanding spacetime
- Axion dark matter, gravitational waves ...

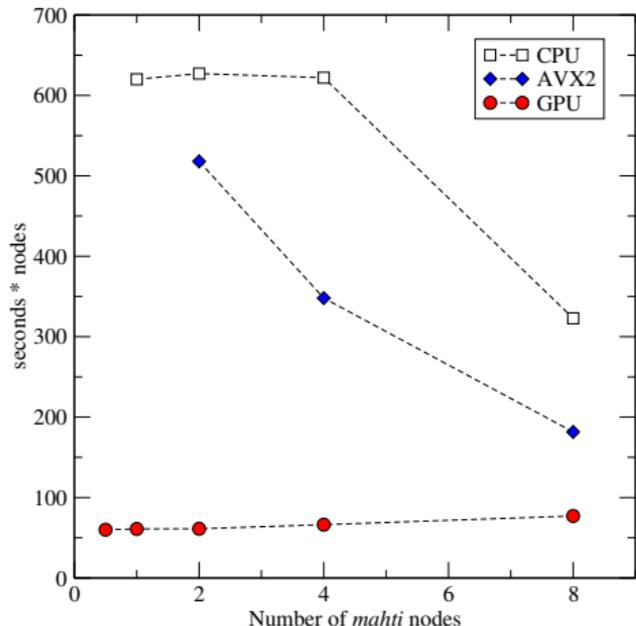
*Picture by José Ricardo Correia*

# Testing production code

## Case study: axion string network evolution code

### Strong scaling test on *Mahti* @ CSC

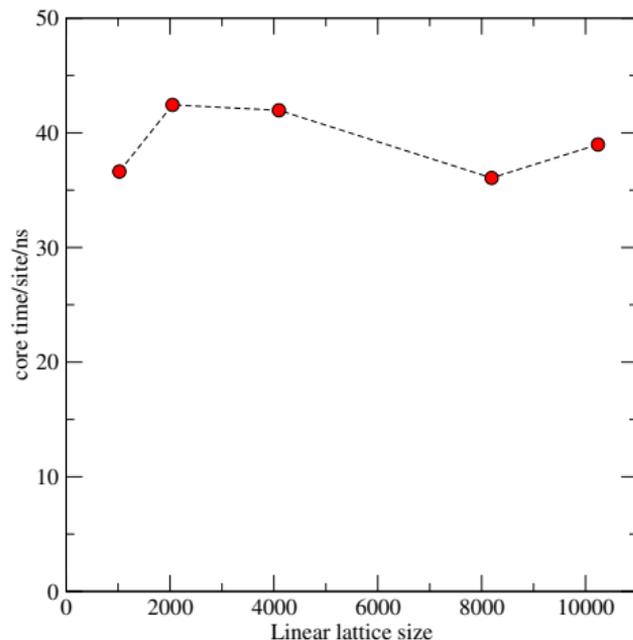
- Lattice size  $512^3$
- 1 Mahti node:
  - ▶  $2 \times$  AMD Rome 7H12 = 128 cores
  - ▶  $2 \times$  NVIDIA A100
- Perfect scaling : horizontal line
- CPU cores: 128 – 1024
- Superscaling on cpus! (memory bandwidth?)
- With 8 nodes, CUDA-aware MPI does not work - extra copies in communications ( $\sim 25\%$  slowdown)



# Production runs on LUMI-C (non-GPU partition)

## Axion string network code $\sim$ weak scaling test

- 1 LUMI-C node:  $2 \times$  AMD EPYC Milan 7742 = 128 cores
- Cosmic string evolution code
- Lattice sizes  $1024^3 - 10240^3$
- Largest lattice: 500 LUMI nodes = 64 000 cores (64 000 MPI processes)
- With AVX2
- Simulation limited by available number of LUMI nodes
- Core nanoseconds / site / update
- Goal for LUMI-G:  $16\,000^3 - 20\,000^3$



# HILA requirements

- C++17 compatible compiler : gcc 8, clang 10, Cray CC ...
- CUDA 10, hipcc (?)
- MPI, FFTW (on cpu targets)
- In order to compile hilapp, clang development libs needed
  - ▶ On Linux (Ubuntu):
    - > `apt install clang llvm clang-tools libclang-common-dev libclang-cpp-dev libclang-dev clang-format`
  - ▶ These are not installed on supercomputers – compile hilapp statically on your workstation/laptop and copy the program.
  - ▶ Mac – compile full clang dev package from sources(?)
  - ▶ *Clang dev libs are **not** needed to compile application programs!*

# HILA requirements

- C++17 compatible compiler : gcc 8, clang 10, Cray CC ...
- CUDA 10, hipcc
- MPI, FFTW (on cpu targets)
- In order to compile hilapp, clang development libs needed
  - ▶ On Linux (Ubuntu):
    - > `sudo apt install clang llvm clang-tools libclang-cpp-dev libclang-dev clang-format`
  - ▶ These are not installed on supercomputers. Either
    - ★ compile hilapp statically on your workstation/laptop and copy the program.
    - ★ if your super has *singularity* installed, make a singularity package from hilapp and copy that over (recommended)
  - ▶ On a Mac or Windows: use virtual machine (docker or WSL)
  - ▶ *Clang dev libs are not needed to compile application programs!*

- It works! AVX/cuda/hip, FFT, field I/O, ...
- Backwards compatibility – (legal) programs will run in the future
- To be done:
  - ▶ Dirichlet and custom funky boundary conditions
  - ▶ **QCD library – fermions to be done**
    - ★ conversion from our old code (Wilson-clover, smearing, rhmc, Hasenbusch ...)
    - ★ hook up QUDA for QCD backend?
  - ▶ Multigrid (switching lattice size)
  - ▶ Nips and tucks, optimisations and cleanups
- Were we successful, is it easy to use?
  - ▶ Yes for writing application programs
  - ▶ Implementing new backends? Well ...

- 1st publication using HILA:  
Laine, Nurmi, Procacci, Rummukainen,  
*Shape of the hot topological charge density spectral function*,  
<https://arxiv.org/abs/2209.13804>  
(code used in this is included in the HILA distribution)
- 3 other projects currently ongoing
  
- HILA system is open source (GPLv2 + MPL for hilapp)  
<https://github.com/CFT-HY/HILA>
- You're welcome to check hila and use it. Code contributions are very welcome!