TDDD56

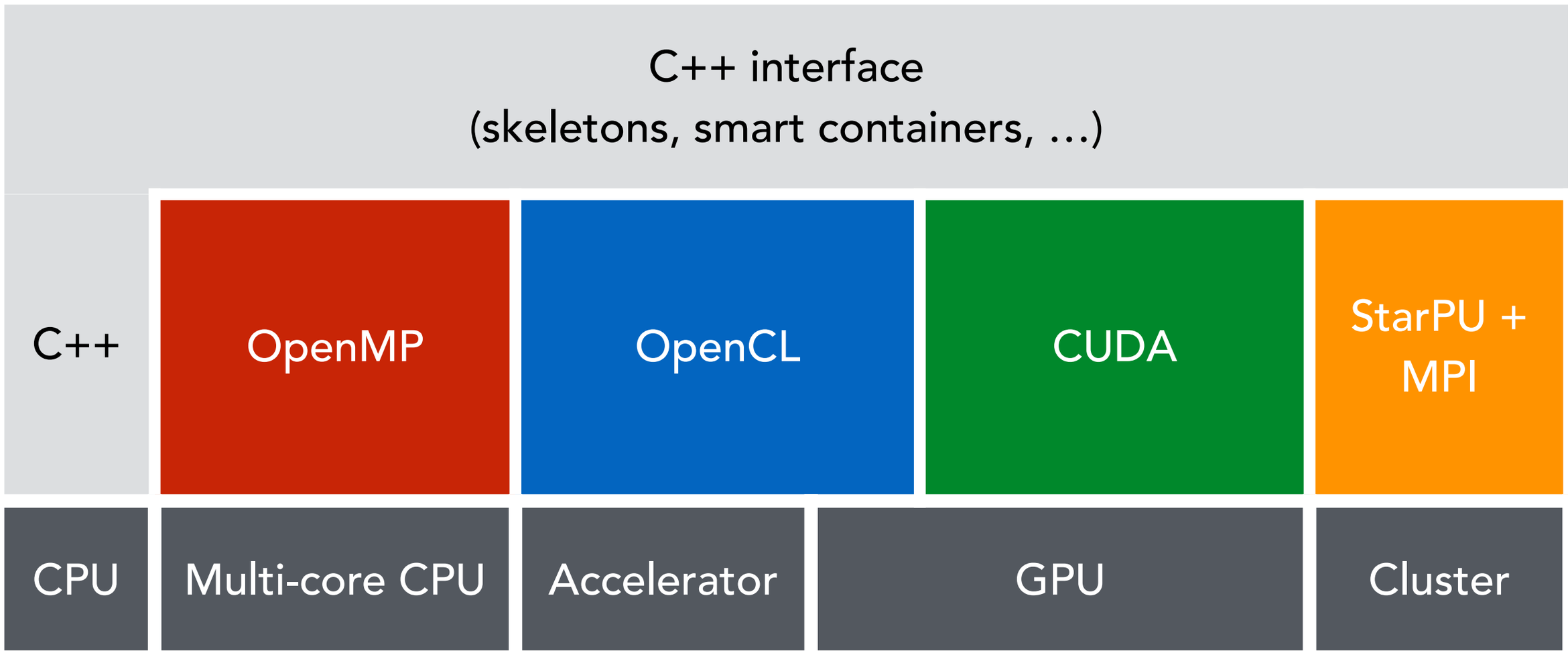
Lab 3: Skeleton programming with SkePU

August Ernstsson

SkePU

- Skeleton programming framework
  - C++11 **library** with skeleton and data-container classes
  - A source-to-source **translator tool**
- Smart containers: `Vector<T>, Matrix<T>,` tensors
- For **heterogeneous multicore** systems and clusters
  - Multiple backends
- Active research tool (A good topic for your thesis?)

SkePU architecture



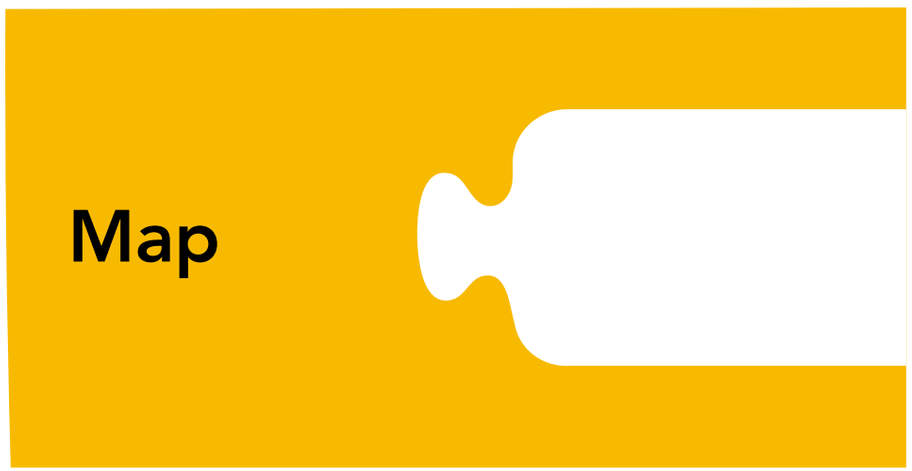
SkePU skeletons

- Parametrizable higher-order functions implemented as C++ template classes
- Data-parallel

Used in lab

- **Map**
- **Reduce**
- **MapReduce**
- **MapOverlap**

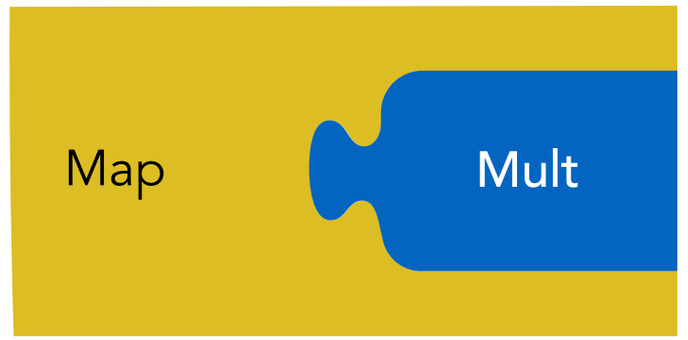
- MapPairs
- MapPairsReduce
- Scan



SkePU skeletons

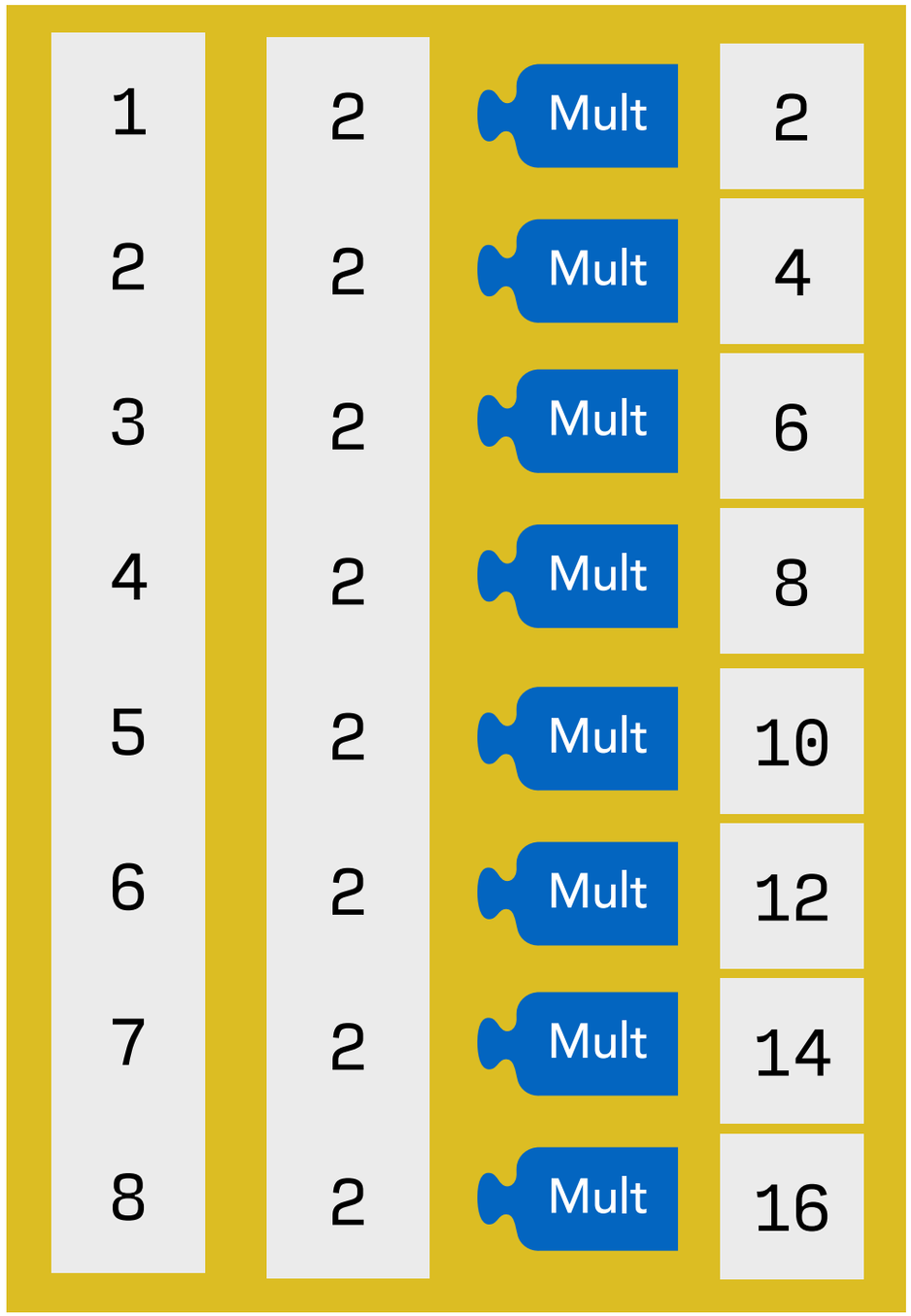
Sequential algorithm

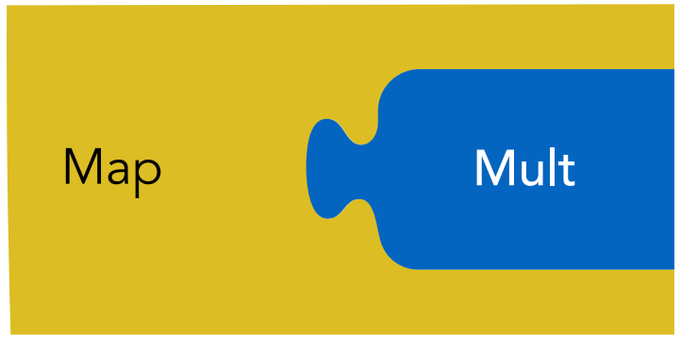
1	2	Mult	2
2	2	Mult	4
3	2	Mult	6
4	2	Mult	8
5	2	Mult	10
6	2	Mult	12
7	2	Mult	14
8	2	Mult	16



SkePU skeletons

Parallel algorithm



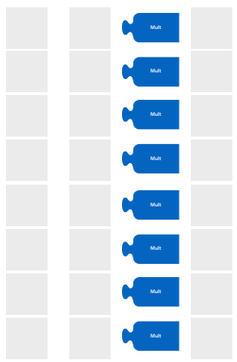


SkePU syntax

```
int add(int a, int b, int m)
{
    return (a + b) % m;
}
```


```
auto vec_sum = Map<2>(add);
```


```
vec_sum(result, v1, v2, 5);
```



Short example in editor

SkePU syntax, advanced

```
template<typename T>
T abs(T input)
{
   return input < 0 ? -input : input;
}
```

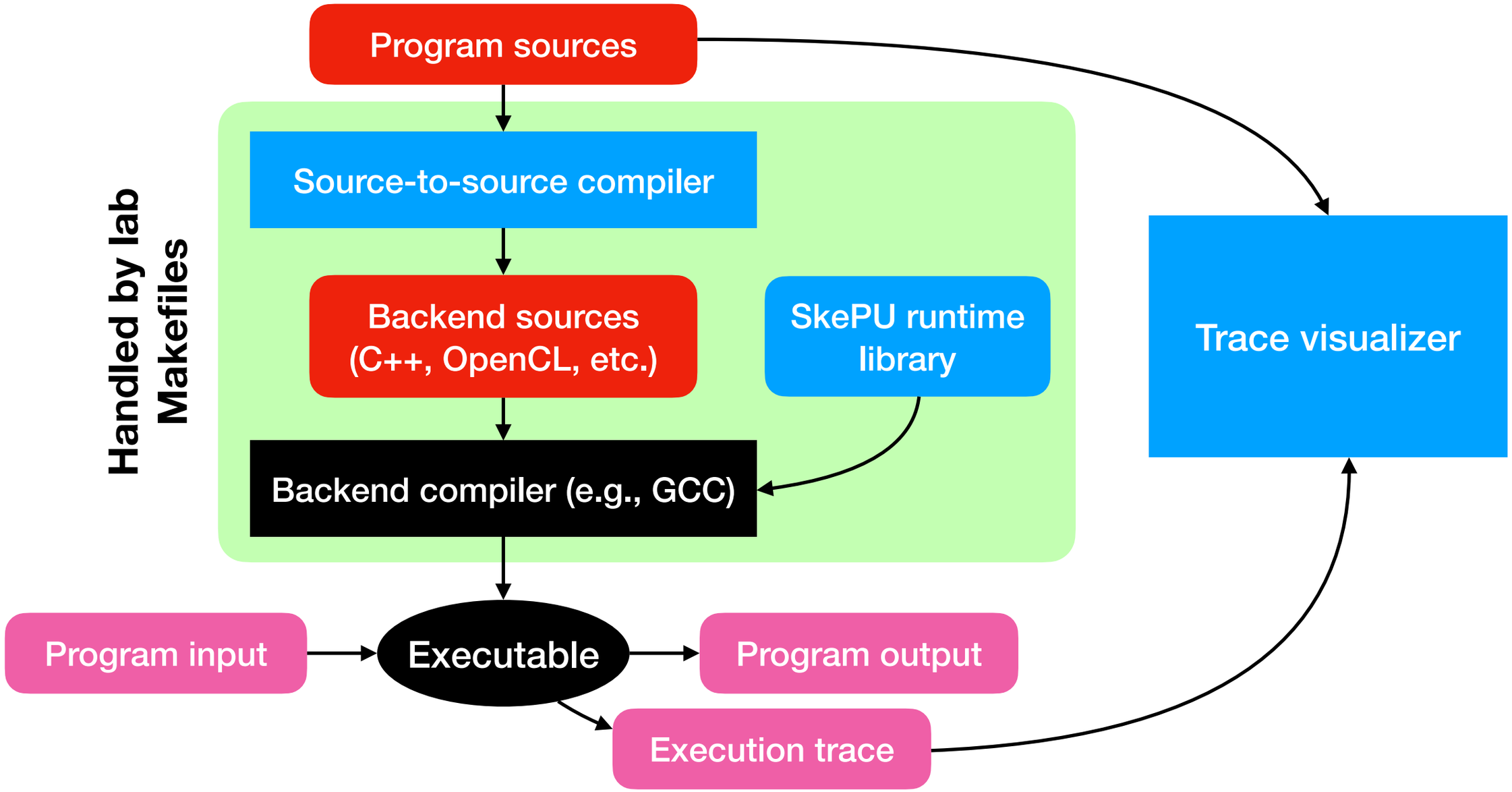
```
template<typename T>
T userfunc(Index1D row, const Mat<T> m, const Vec<T> v)
{
   T res = 0;
   for (size_t i = 0; i < v.size; ++i)
      res += m(row.i, i) * v(i);

   return abs(res);
}
```

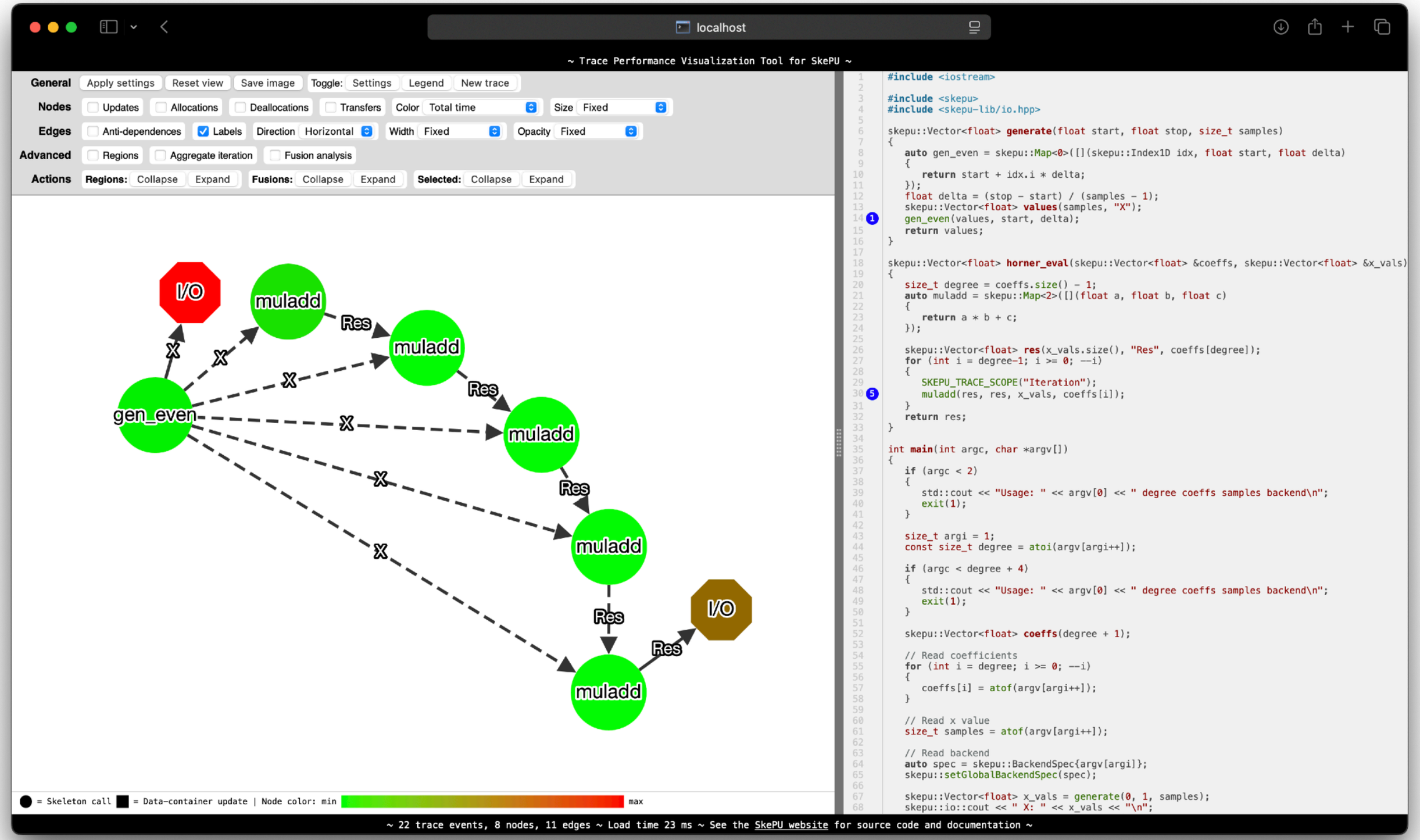
SkePU data-containers

- **Smart** data-containers: `Vector<T>, Matrix<T>, …`
- Manages data across CPU and GPU
- No data transfers unless necessary (lazy copying)
- Keeps track of most recent writes
  - *Memory consistency* through software

SkePU usage process



Visualizer




```cpp
#include <iostream>

#include <skepu>
#include <skepu-lib/io.hpp>

skepu::Vector<float> generate(float start, float stop, size_t samples)
{
   auto gen_even = skepu::Map<0>([](skepu::Index1D idx, float start, float delta)
   {
      return start + idx.i * delta;
   });
   float delta = (stop - start) / (samples - 1);
   skepu::Vector<float> values(samples, "X");
   gen_even(values, start, delta);
   return values;
}

skepu::Vector<float> horner_eval(skepu::Vector<float> &coeffs, skepu::Vector<float> &x_vals)
{
   size_t degree = coeffs.size() - 1;
   auto muladd = skepu::Map<2>([](float a, float b, float c)
   {
      return a * b + c;
   });

   skepu::Vector<float> res(x_vals.size(), "Res", coeffs[degree]);
   for (int i = degree-1; i >= 0; --i)
   {
      SKEPU_TRACE_SCOPE("Iteration");
      muladd(res, res, x_vals, coeffs[i]);
   }
   return res;
}

int main(int argc, char *argv[])
{
   if (argc < 2)
   {
      std::cout << "Usage: " << argv[0] << " degree coeffs samples backend\n";
      exit(1);
   }

   size_t argi = 1;
   const size_t degree = atoi(argv[argi++]);

   if (argc < degree + 4)
   {
      std::cout << "Usage: " << argv[0] << " degree coeffs samples backend\n";
      exit(1);
   }

   skepu::Vector<float> coeffs(degree + 1);

   // Read coefficients
   for (int i = degree; i >= 0; --i)
   {
      coeffs[i] = atof(argv[argi++]);
   }

   // Read x value
   size_t samples = atof(argv[argi++]);

   // Read backend
   auto spec = skepu::BackendSpec{argv[argi]};
   skepu::setGlobalBackendSpec(spec);

   skepu::Vector<float> x_vals = generate(0, 1, samples);
   skepu::io::cout << " X: " << x_vals << "\n";
```

Short demo of trace visualizer

A warning about warnings (and errors)

- SkePU is a C++ template library
- As such, gets very long and unreadable diagnostic messages if used incorrectly!
- Following the structure of the lab files should minimize errors
- Avoid using `const`! SkePU is not ”const-correct”.

The goal of Lab 3

- SkePU is a niche framework, a research prototype here at LiU
- The goal of the lab is <u>not to learn</u> SkePU
- Try to be *critical* and *reflect* over e.g. design and implementation decisions of a high-level parallel pattern framework
  - Are there programs not expressible in SkePU skeletons?
  - Is SkePU introducing performance overheads or bottlenecks?
  - **=> What is the cost of abstraction? When is it worth it?**
- When and how to do performance debugging

Questions?

- About skeleton programming?
- About SkePU?

Lab structure

- Three exercises:

    1. Warm-up: Dot product
    2. Averaging image filter + gaussian filter
    3. Median filter
    4. Cooldown: Performance debugging (Game of Life)

1. Dot product

- Implement two variants of dot product:
  - With **MapReduce** skeleton
  - With **Map** + **Reduce** skeletons
- Compare and contrast the variants
  - Why does SkePU have the MapReduce skeleton?
- Measure with different backends and problem sizes

2. Averaging filters

- Averaging filter: find average color value in surrounding region
- Gaussian filter: averaging filter with **non-uniform** weights
- Use the MapOverlap skeleton

Original

Average

Gaussian

3. Median filter

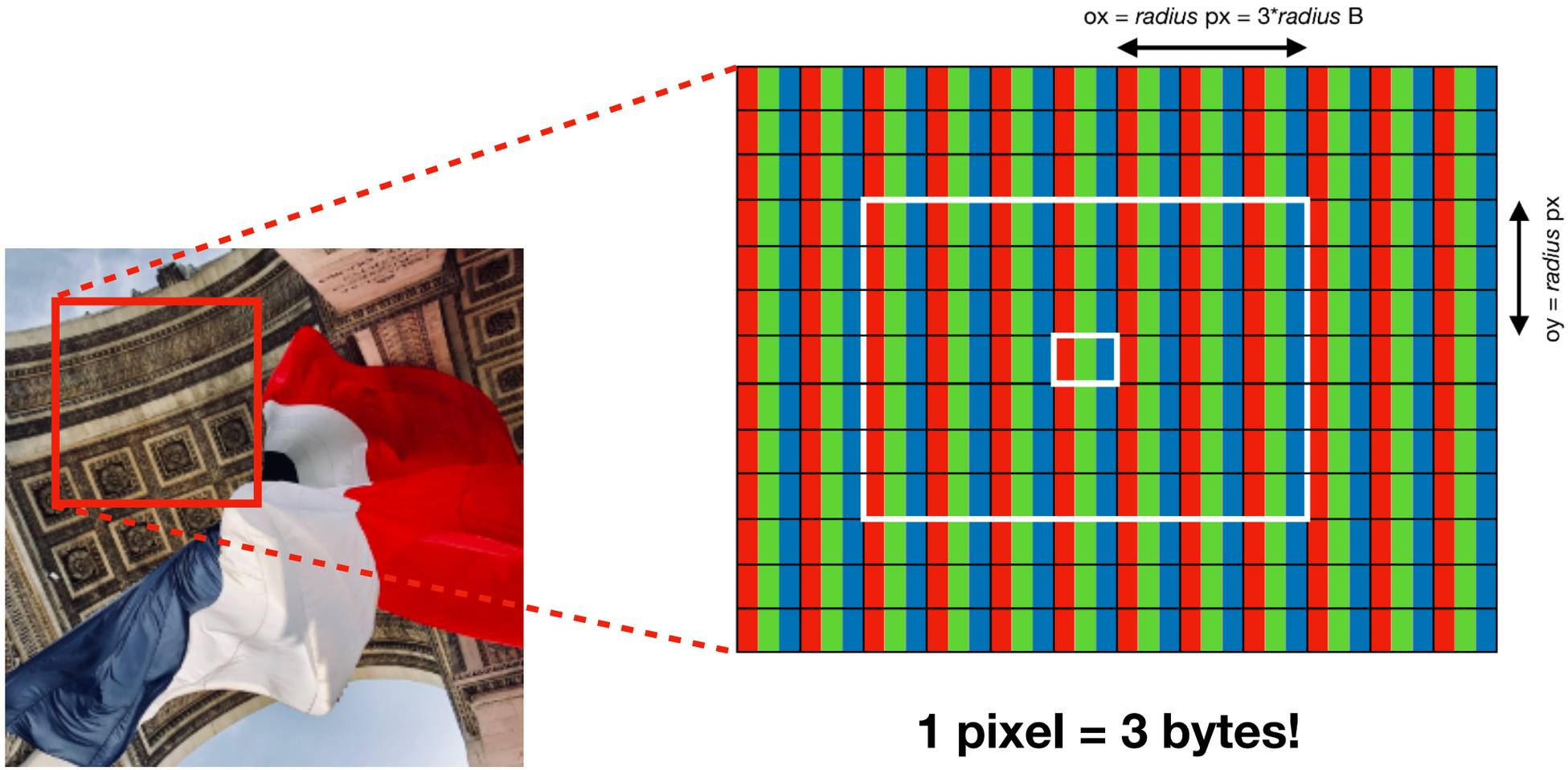
- Median filter: find **median** color value in surrounding region
- Requires sorting the pixel values in some way

Original

Median

Image filters

- Layout of image data in memory



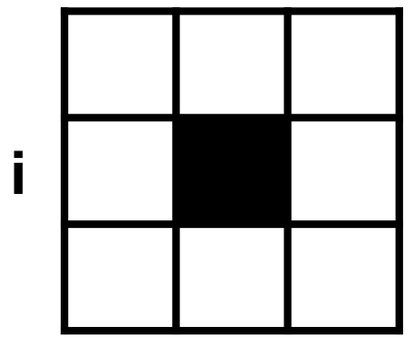
1 pixel = 3 bytes!

4. Performance debugging

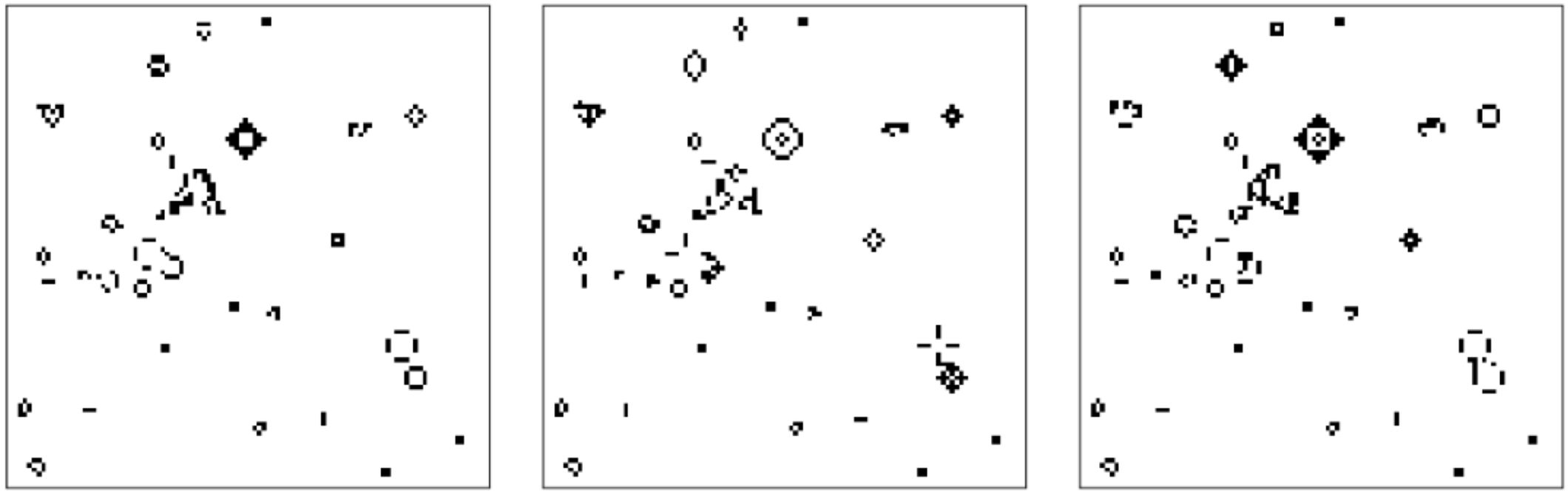
- You get a complete, functioning SkePU program
- **Goal**: Find a performance bottleneck in the given code
- Minimal experimental measurements are needed
- Instead: high-level debugging with trace visualization
  - Find suspicious nodes or patterns in the trace graph

Conway's Game of Life

- Simulate the live state of binary cells in a 2D grid
- Too few/many neighbors = death
- Good number of neighbors = new life / survive
- = stencil computation (MapOverlap)



Conway's Game of Life



(a) Original domain. (b) One iteration. (c) Two iterations.

Figure 3: Two iterations of Conway's Game of Life.

Lab installation

- Get from course website as usual
- Different from public SkePU distribution!
  - Pre-built binary for the Olympen machines
  - May run on other 64-bit Linux (no guarantees)

Lab build/run process

Build lab program:

```
> make bin/addone
```

Run lab program:

```
> bin/addone 100 CPU
```

CPU: Use sequential backend
OpenMP: Use multithreaded backend
OpenCL: Use GPU backend

Lab build/run process

Instructions for setting up the visualizer: see Appendix in the lab 3 compendium

Questions?