June 2022

SkePU

https://skepu.github.io

SkePU Framework
Introduction & Tutorial

August Ernstsson Christoph Kessler
{firstname.lastname}@liu.se
Linköping University, Sweden

LINKÖPING UNIVERSITY

Overview

Basic Topics

- Skeleton Programming
- SkePU Introduction and History
- Using SkePU
- Compilation Options and Backend Selection
- Smart Containers and Consistency Model

Advanced Topics

- SkePU in Current Research
  - Lazy Skeleton Evaluation
  - Multi-Variant User Functions
  - Deterministic Pseudo-Random Number Generation

Intermediate Topics

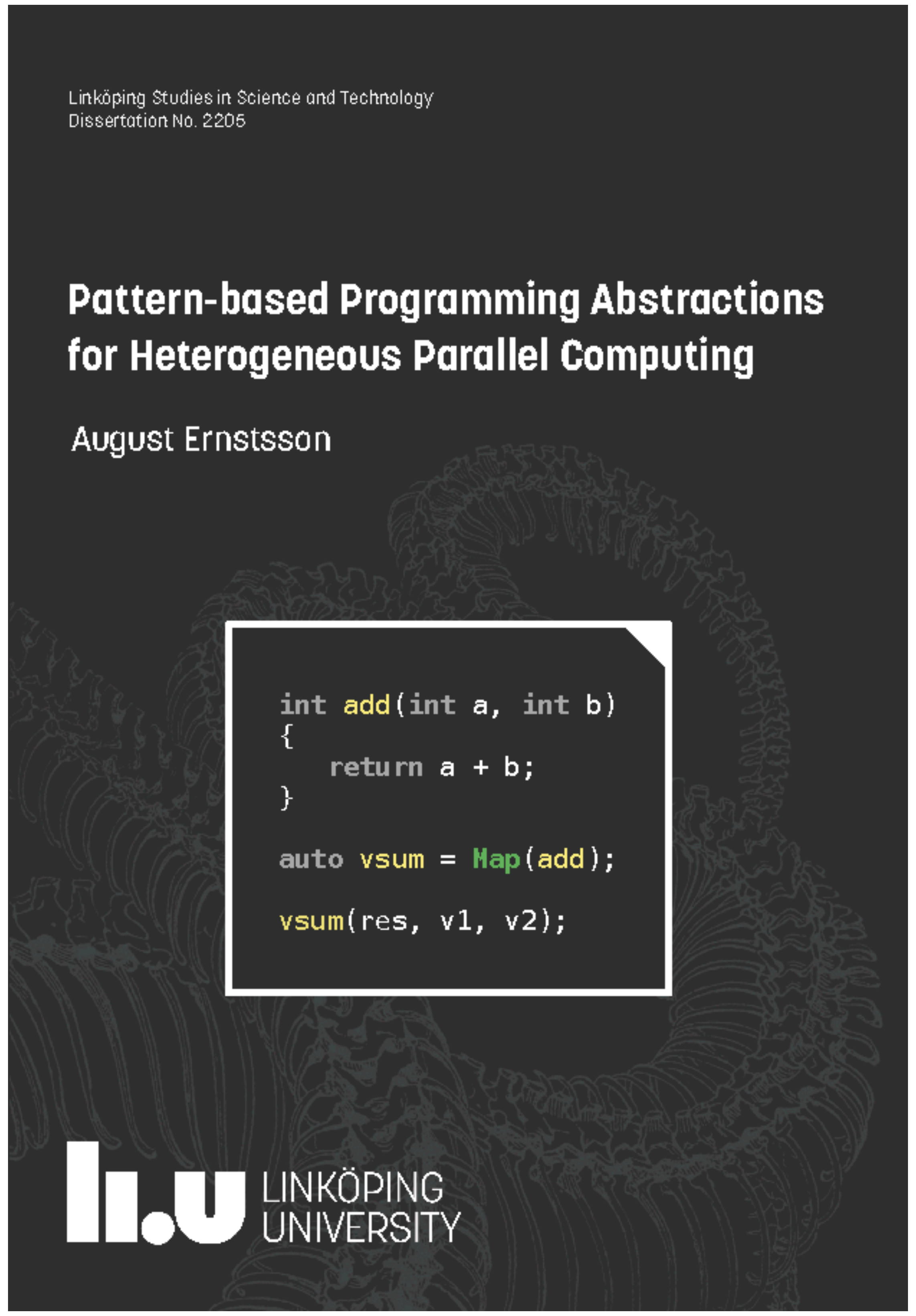
- Skeletons in Depth
  - Map
  - Reduce
  - MapReduce
  - Scan
  - MapOverlap
  - MapPairs
  - MapPairsReduce
- Nested User Functions
- SkePU Standard Library
  - BLAS
  - Deterministic PRNG
  - Image Filters
  - Utilities

Other

- Live Demo
- Behind the Scenes

Further Reading

- Currently, the best information resource on SkePU is August Ernstssons doctoral thesis (2022)
  - https://doi.org/10.3384/9789179291969
- See also the user guide on https://skepu.github.io for more concrete instructions on e.g. installation.
- Revised user guide and documentation is upcoming!



Skeleton Programming

Skeleton Programming :: Motivation

Programming parallel systems is hard!

- Resource utilization
- Synchronization, Communication
- Memory consistency
- Different hardware architectures, heterogeneity

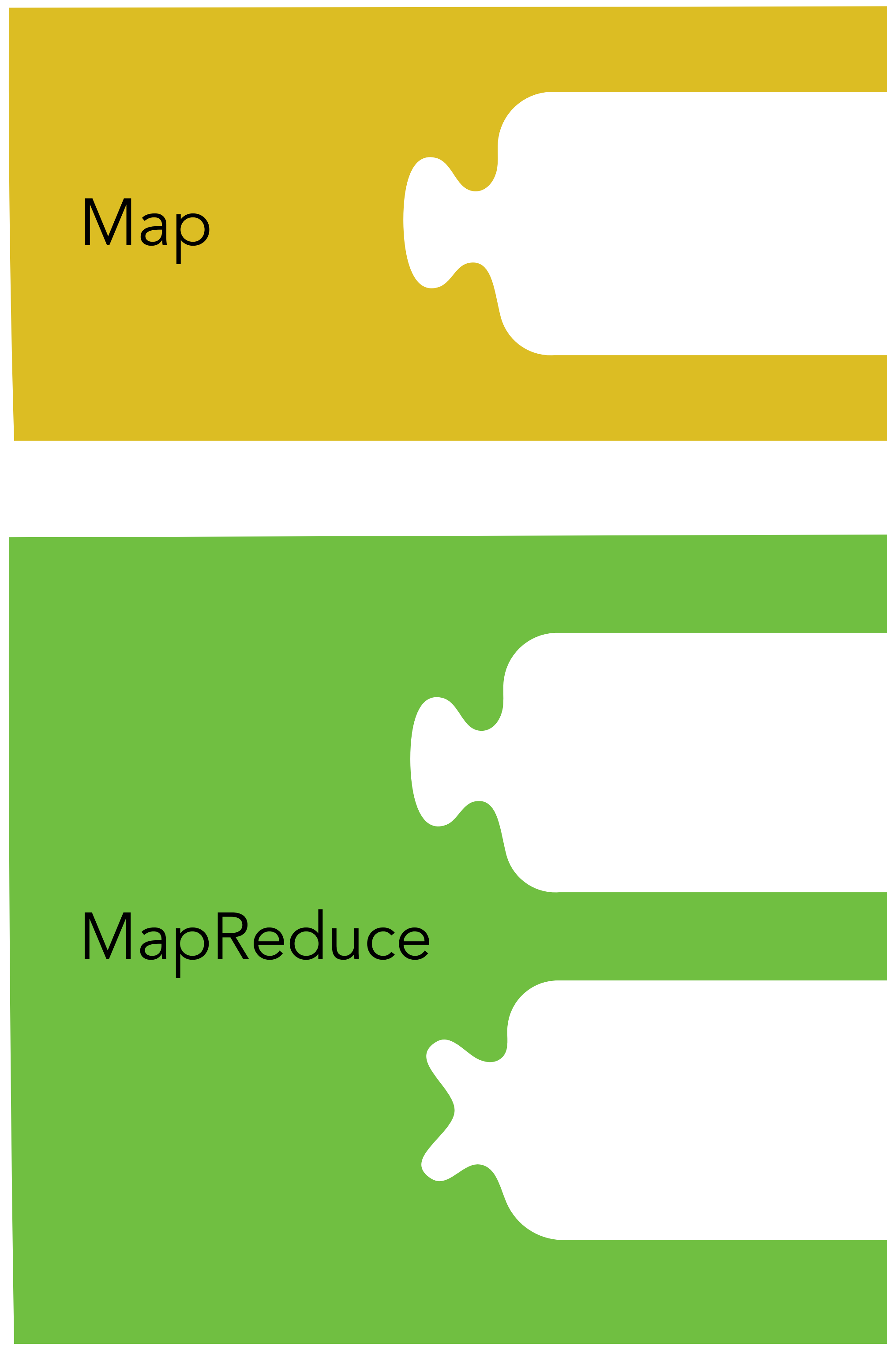
Skeleton programming (algorithmic skeletons)

- A high-level parallel programming concept
- Inspired by functional programming
- Generic computational patterns
- Abstracts architecture-specific issues

Skeletons

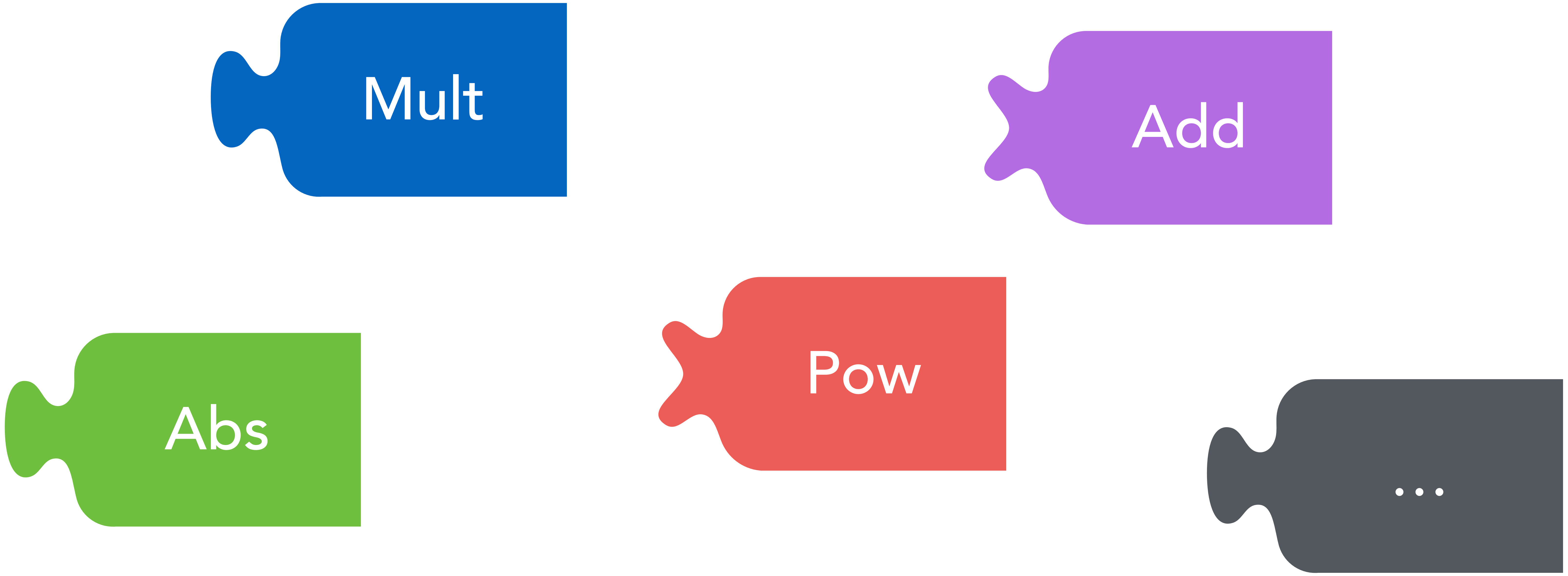
Parametrizable higher-order constructs

- Map
- Reduce
- MapReduce
- Scan
- and others



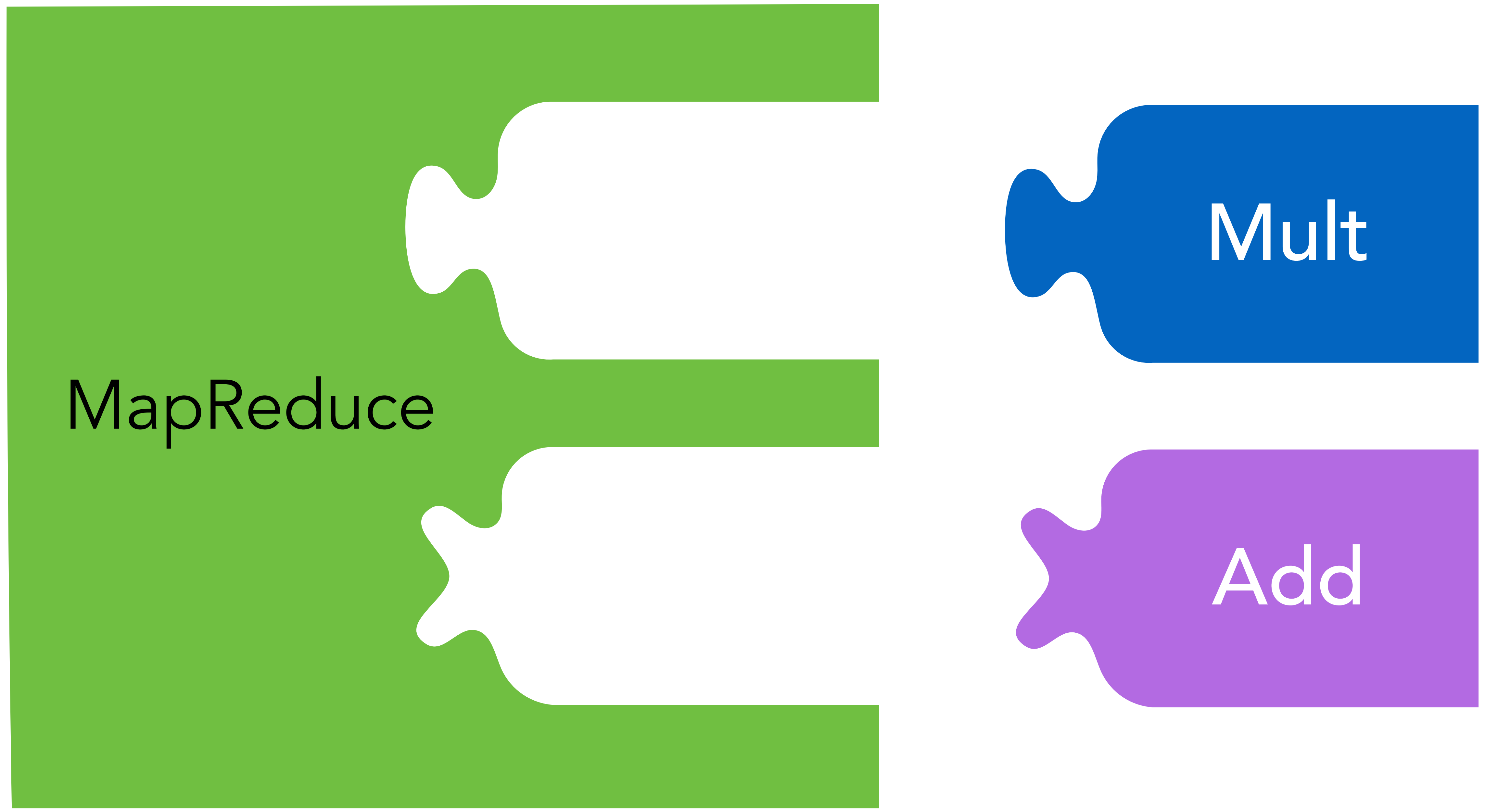
User functions

User-defined operators



Skeleton parametrization example

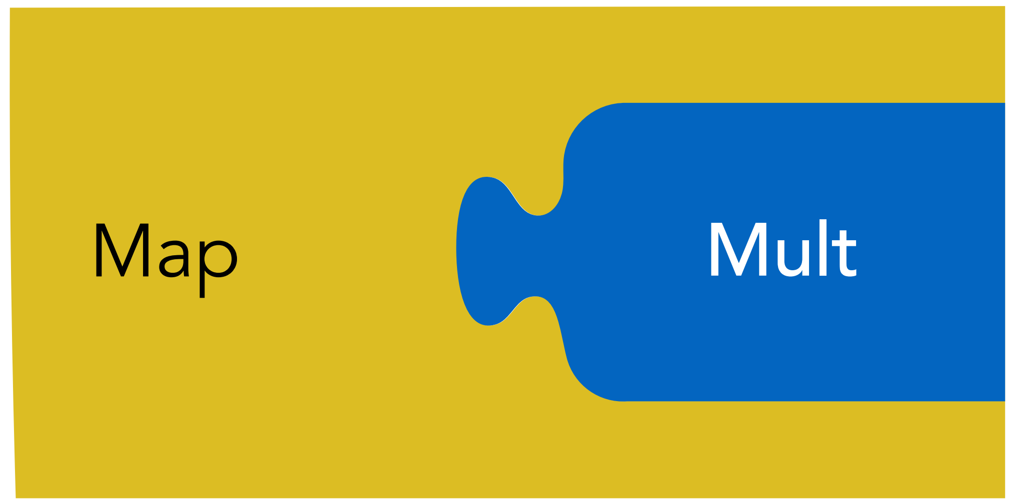
Dot product operation



Skeleton Programming :: Map

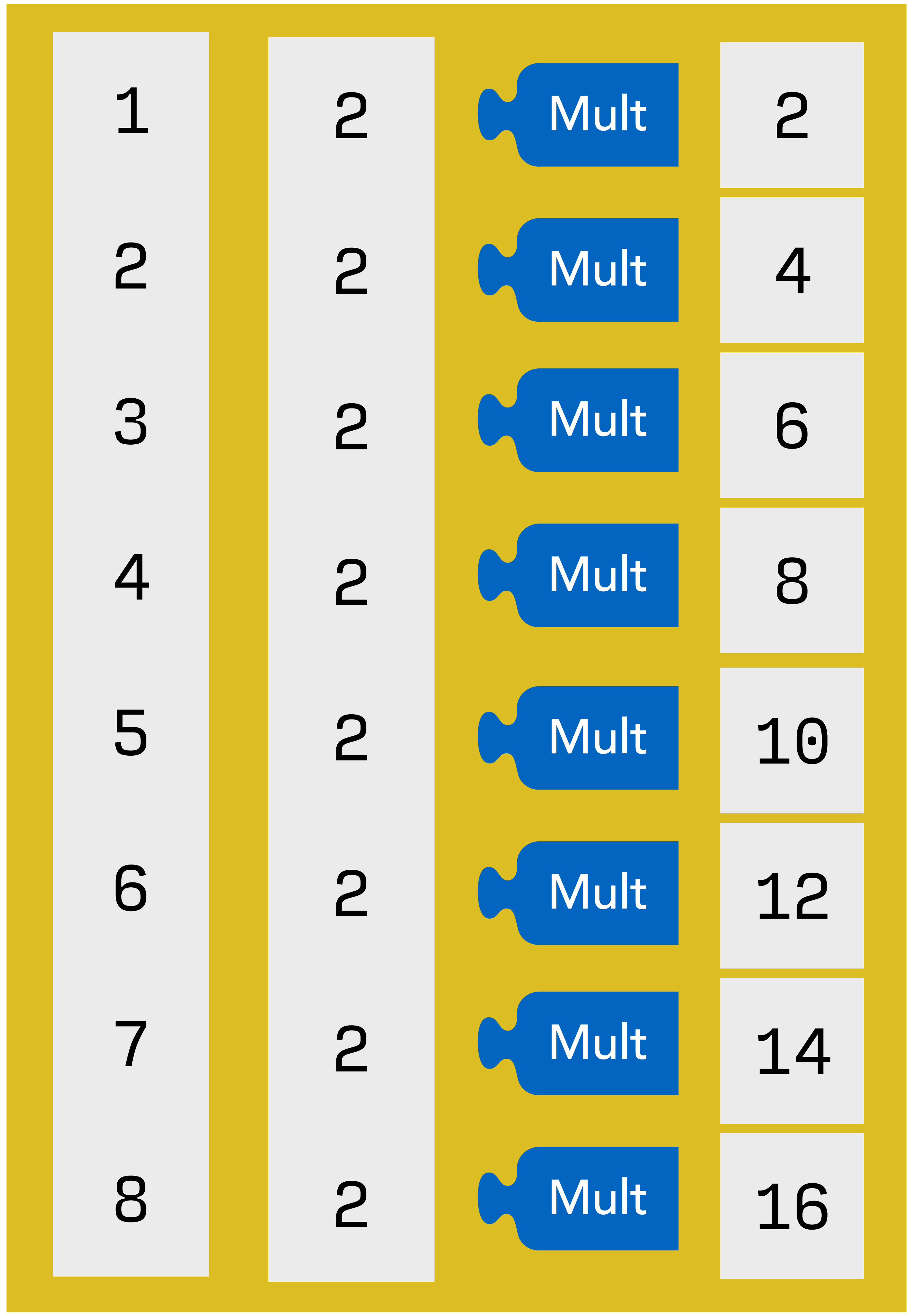
Sequential algorithm

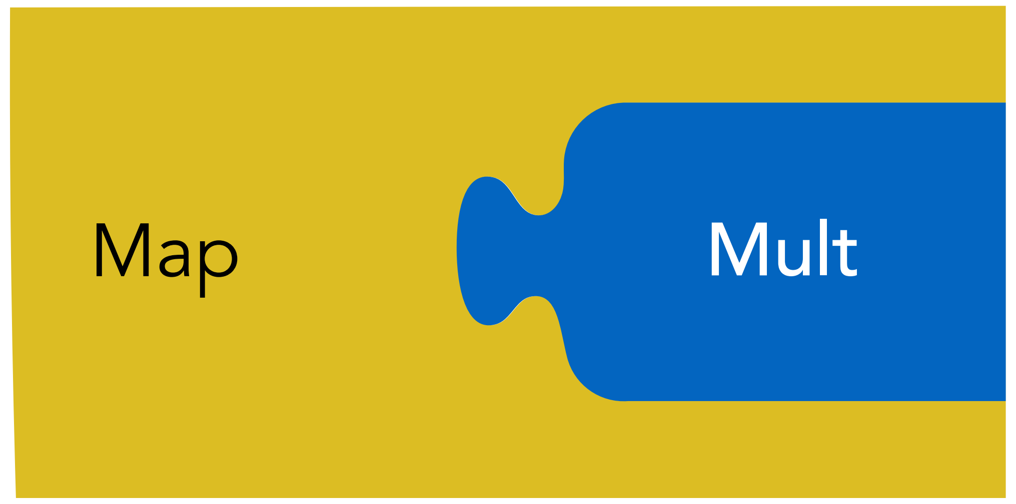
1	2	Mult	2
2	2	Mult	4
3	2	Mult	6
4	2	Mult	8
5	2	Mult	10
6	2	Mult	12
7	2	Mult	14
8	2	Mult	16



Skeleton Programming :: Map

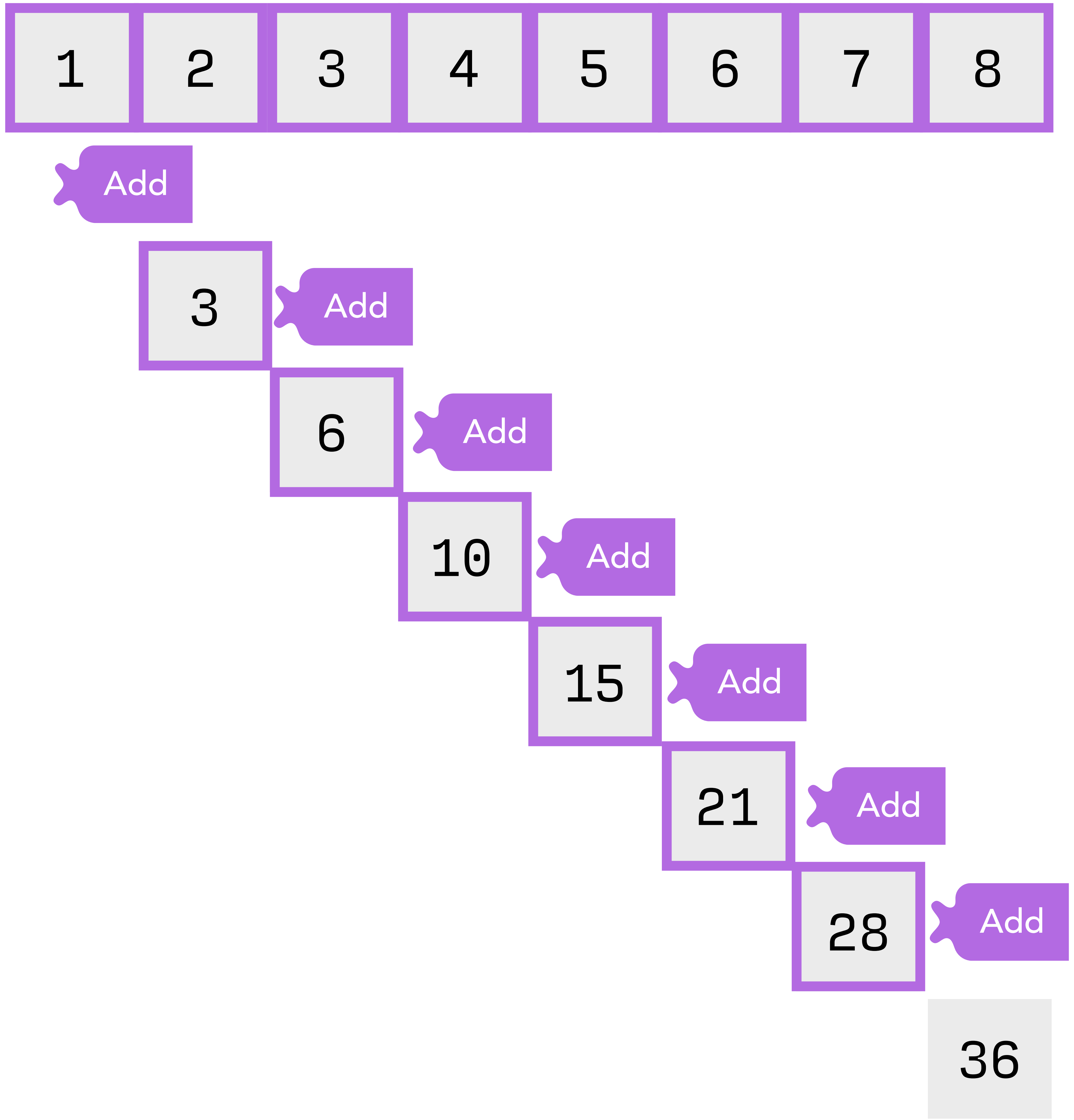
Parallel algorithm





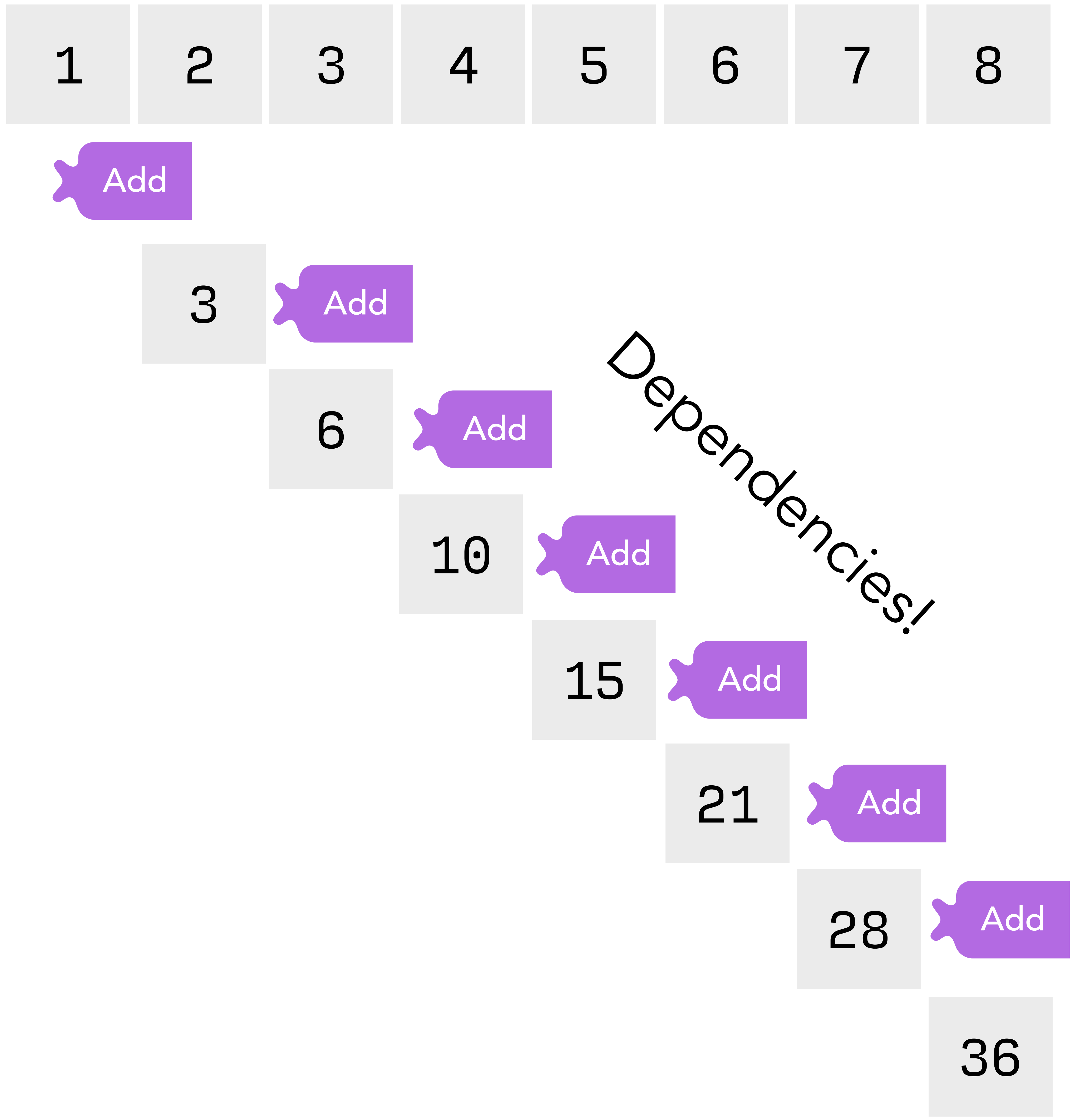
Skeleton Programming :: Reduce

Sequential algorithm



Skeleton Programming :: Reduce

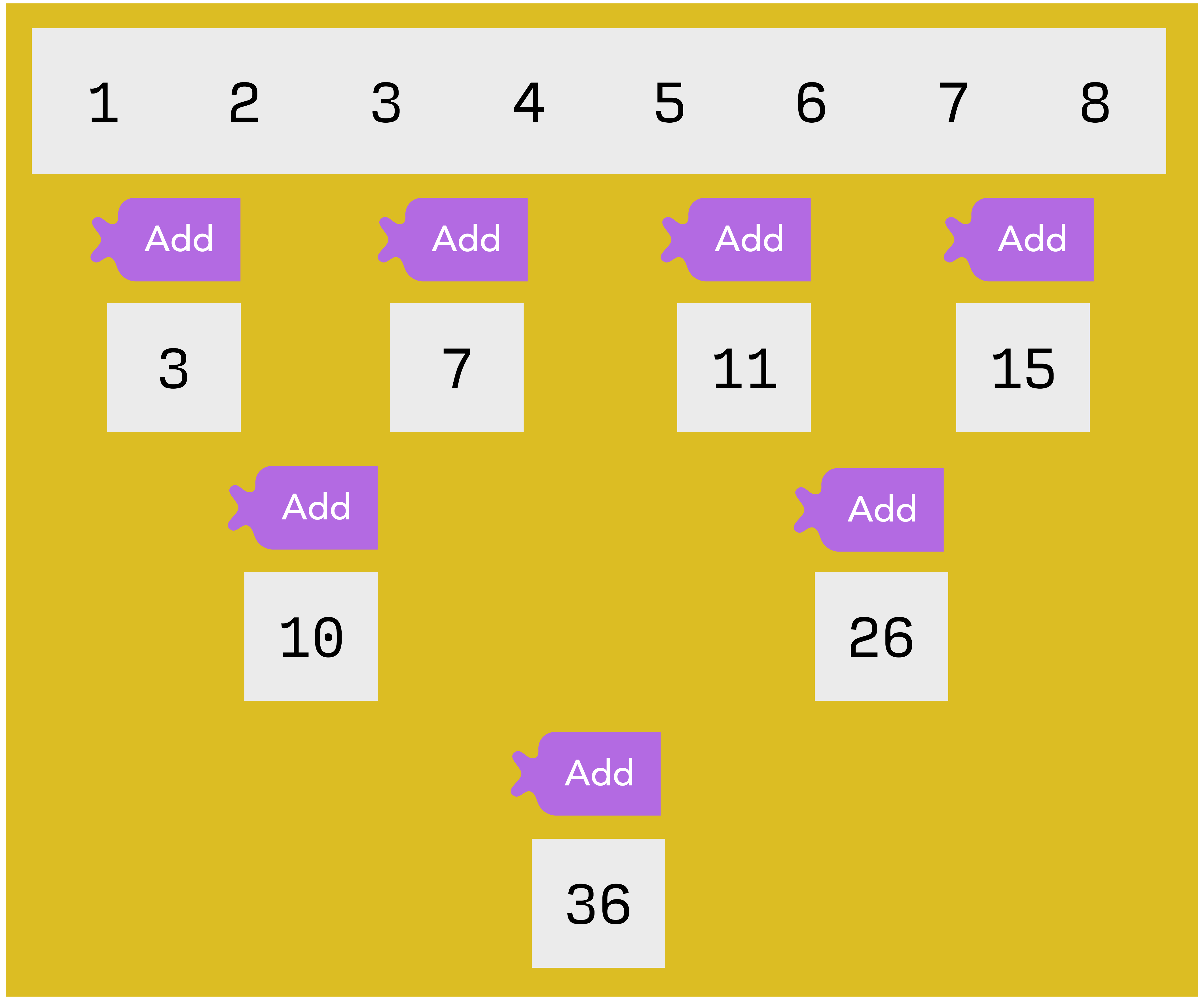
Parallel algorithm?



Skeleton Programming :: Reduce

Parallel algorithm

(assuming associativity)



SkePU Introduction and History

But First: Modern C++

- SkePU uses "modern" C++

```
// "auto" type specifier
auto addOneMap = skepu::Map<1>(addOneFunc);

skepu::Vector<float> input(size), res(size);
input.randomize(0, 9);

// Lambda expression
auto dur = skepu::benchmark::measureExecTime([&]
{
    addOneMap(res, input);
});
```

capture by reference

- Implementation reliant on variadic templates, template metaprogramming, and other C++11 features

Goals

✓ Efficient parallel algorithms

✓ Accessible interface

✓ Memory management and data movement

✓ Automatic backend selection and tuning

History

- **SkePU 1**, public release **2010**
  - C++ template-based interface (limited arity)
  - Multi-backend using macro-based code generation

Enmyren, Kessler, HLPP 2010
https://doi.org/10.1145/1863482.1863487
(See website for more publications)

- **SkePU 2**, public release **2016**
  - C++11 *variadic* template interface (flexible arity)
  - Multi-backend using source-to-source precompiler

Ernstsson, Li, Kessler, Int J Parallel Prog 46, 62–80 (2018)
https://doi.org/10.1007/s10766-017-0490-5

- **SkePU 2.1 (2017)** *Experimental feature*: Lazy evaluation

Ernstsson, Kessler,
Concurrency Computat Pract Exper. 2019; 31:e5003
https://doi.org/10.1002/cpe.5003

- **SkePU 2.2**, public release **2018**
  - Hybrid CPU-GPU backends

Öhberg, Ernstsson, Kessler,
***J Supercomput* 76, 5038–5056 (2020)**
https://doi.org/10.1007/s11227-019-02824-7

- **SkePU 2.3 (2019)** *Experimental feature*: Multi-variant user functions

Ernstsson, Kessler,
Parallel Computing: Technology Trends, IOS PRESS , 2020, Vol. 36, pp. 475-484
http://doi.org/10.3233/APC200074

History

- **SkePU 3**, public release **2020:** Expanding skeleton set, container set, and expressivity
  - MapPairs, MapPairsReduce
  - Tensor containers (3D and 4D) and new "proxy" containers MatRow, MatCol
  - Cluster backend with StarPU-MPI
  - Improved syntax and memory consistency model
  - Dynamic scheduling

Ernstsson, Ahlqvist, Zouzoula, Kessler, *Int J Parallel Prog* (2021). https://doi.org/10.1007/s10766-021-00704-3

Panagiotou, Ernstsson, Ahlqvist, Papadopoulos, Kessler, Soudris, SCOPES '20 proceedings, Pages 74–77 https://doi.org/10.1145/3378678.3391889

Papadopoulos *et al.*, *IEEE Transactions on Parallel and Distributed Systems* https://doi.org/10.1109/TPDS.2021.3104257

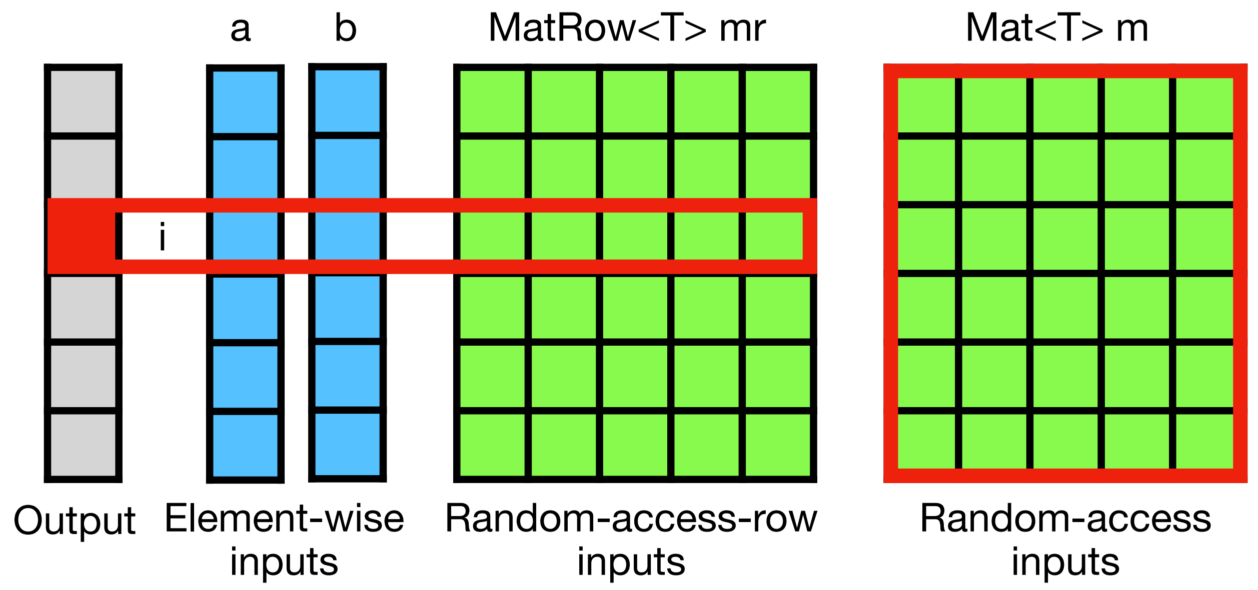
- **SkePU 3.1,** public release **2021:** SkePU "standard library"
  - Complex number API
  - BLAS (level 1 + dense level 2, 3)
  - Deterministic PRNG **Ernstsson, Kessler, HLPP 2021, to appear.**
  - Strided Map skeletons

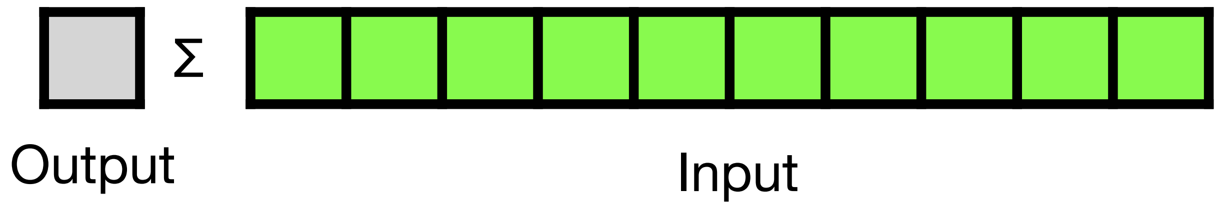
Features

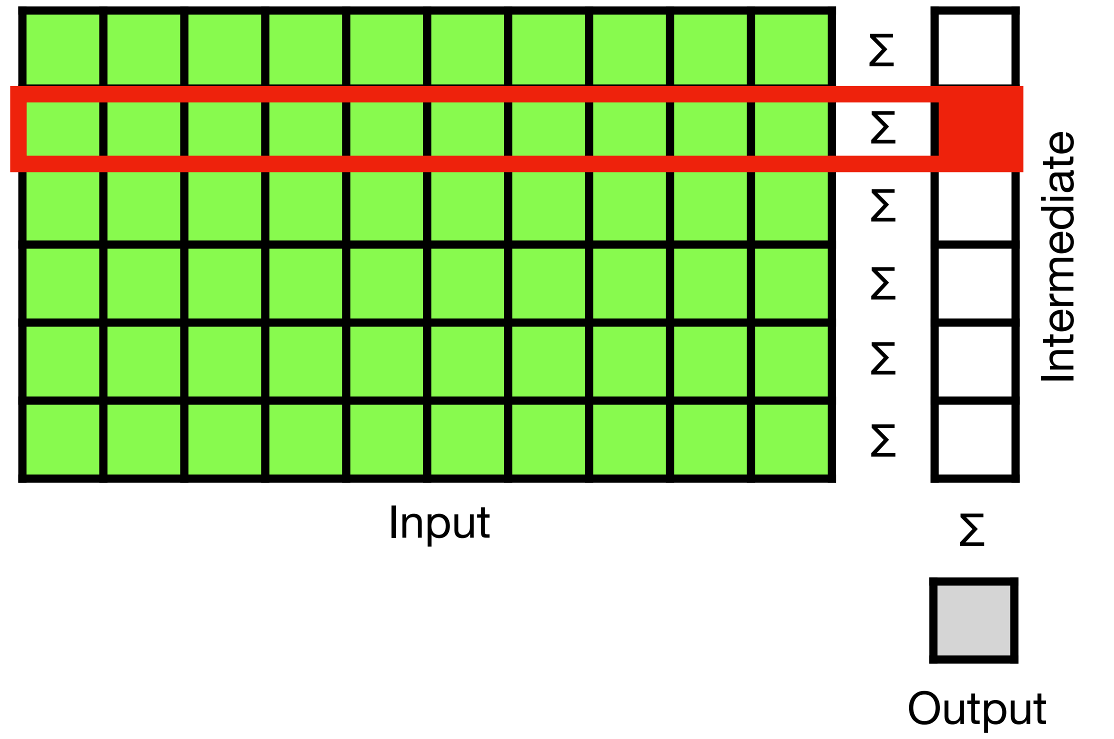
- Skeleton programming framework
  - C++11 **library** with skeleton and data container classes
  - A **Clang**-based source-to-source **pre-compiler**
- Smart containers: Vector<T>, Matrix<T>, Tensor3<T>, Tensor4<T>
  - In development: SparseMatrix<T>
- For **heterogeneous multicore** systems
  - Multiple backends
- Active research tool with a number of publications 2010-2022 (see website)

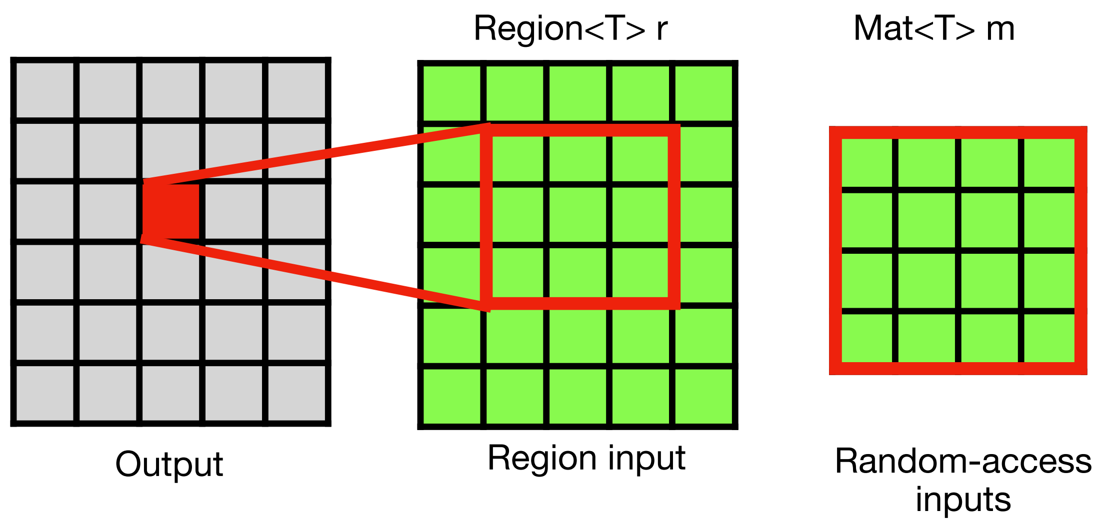
Skeletons

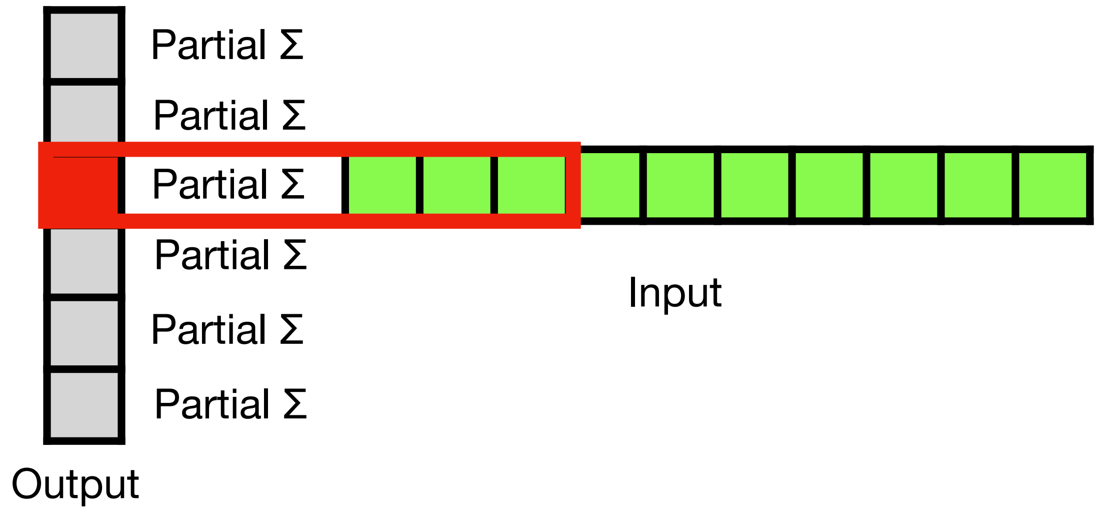
- **Skeletons provided by SkePU**
  - Map
  - Reduce
  - MapReduce
  - Scan
  - MapOverlap
  - MapPairs
  - MapPairsReduce

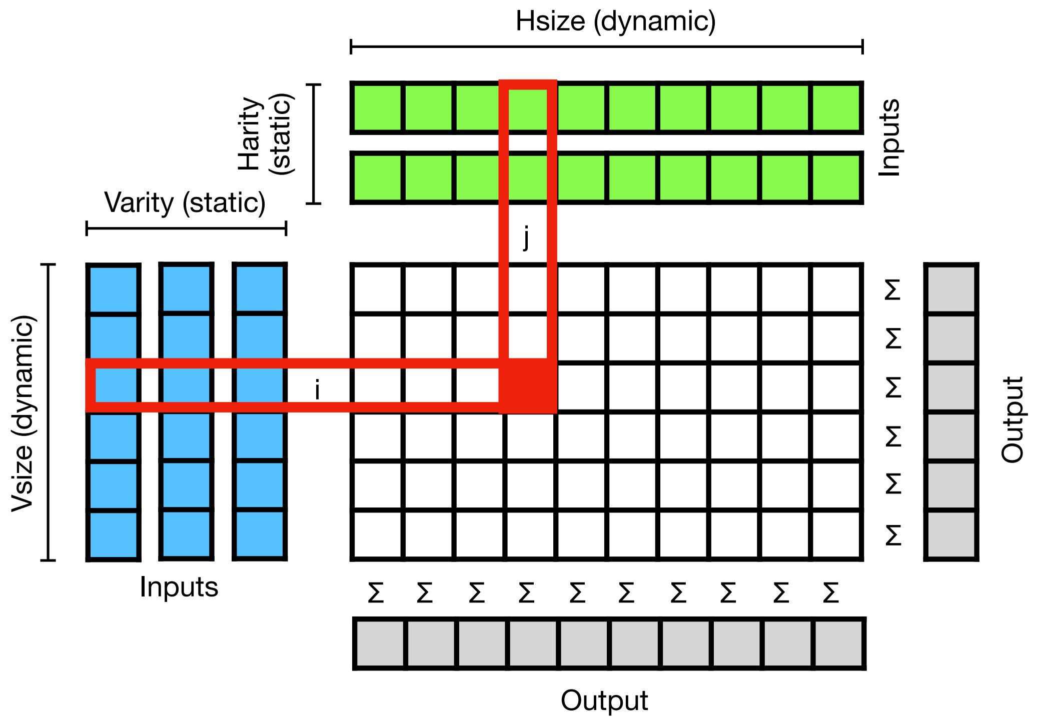


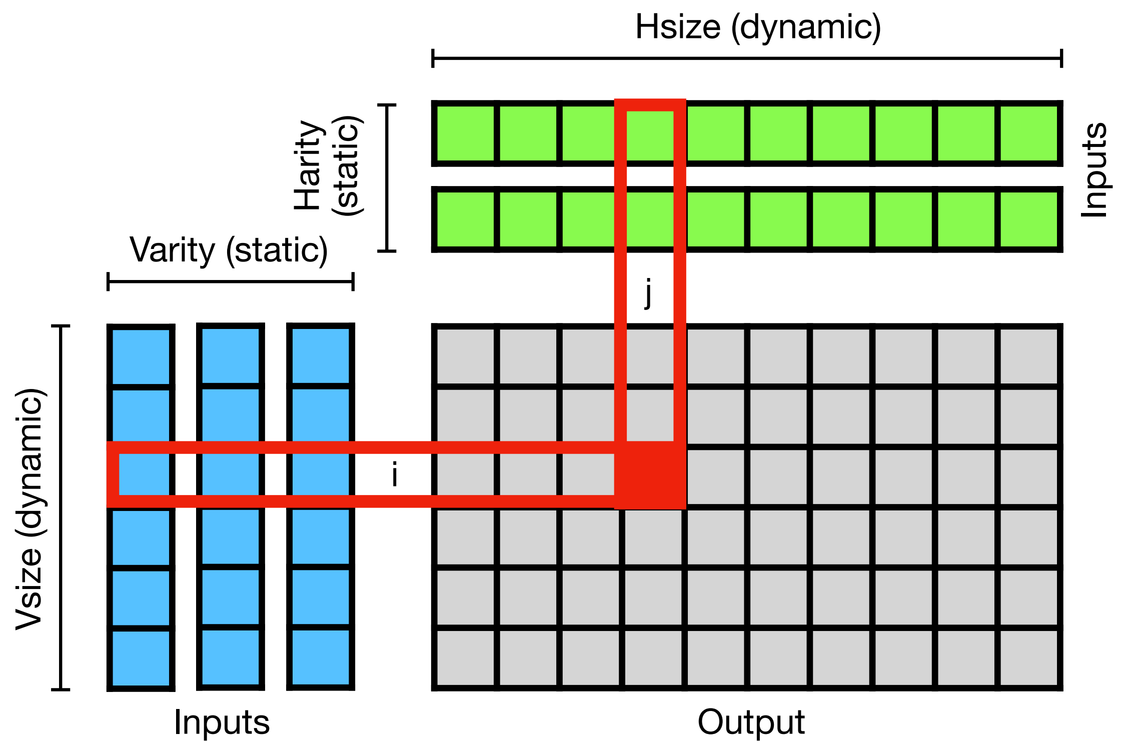




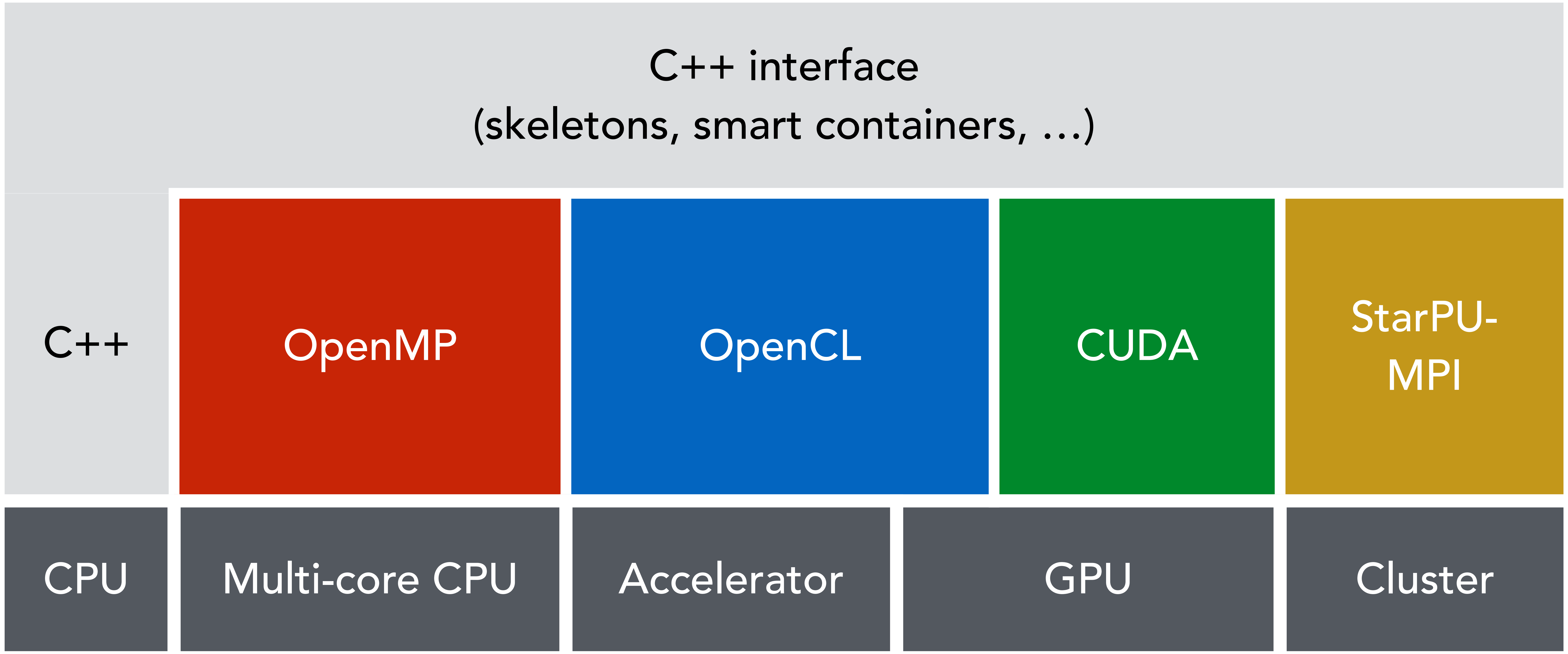




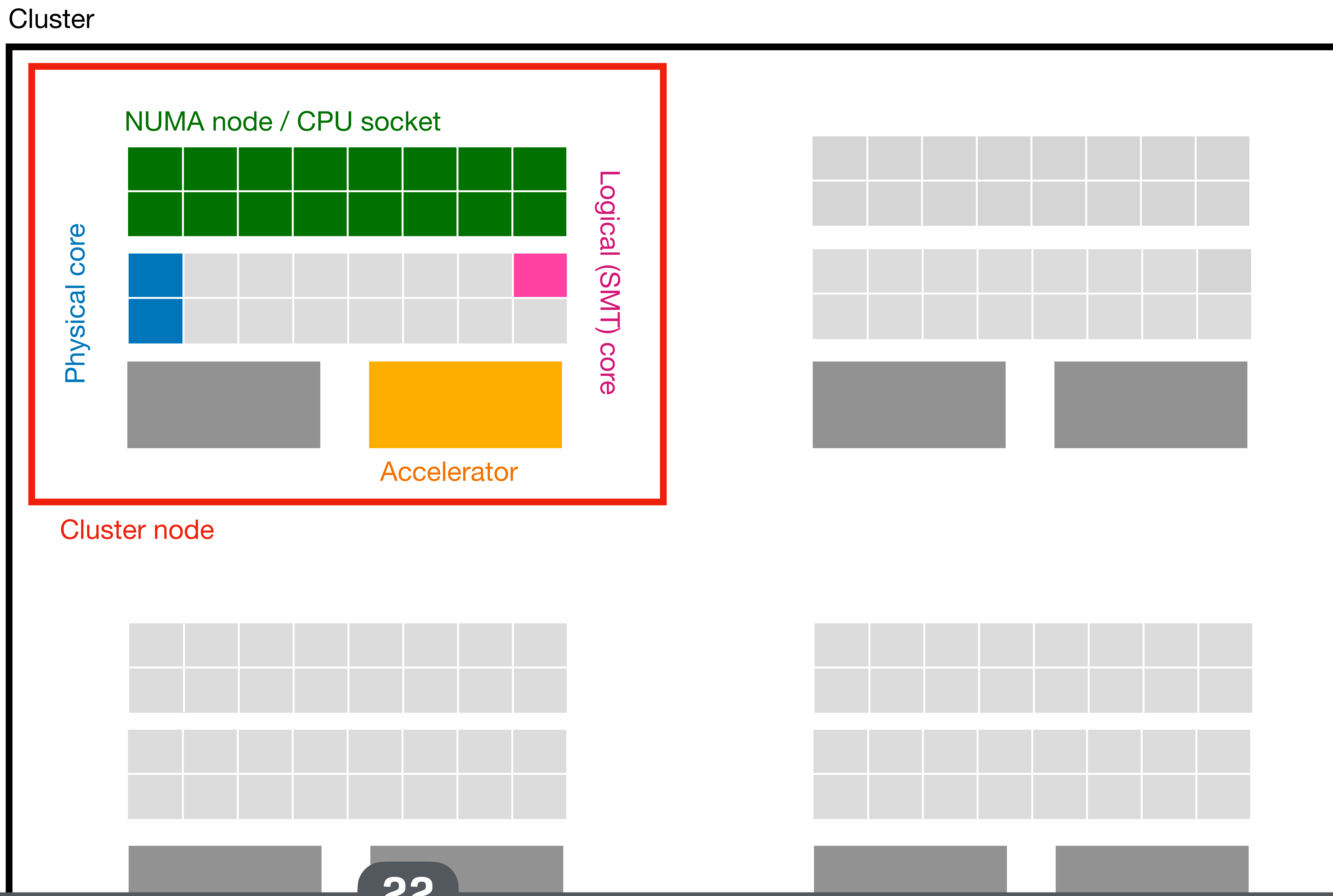




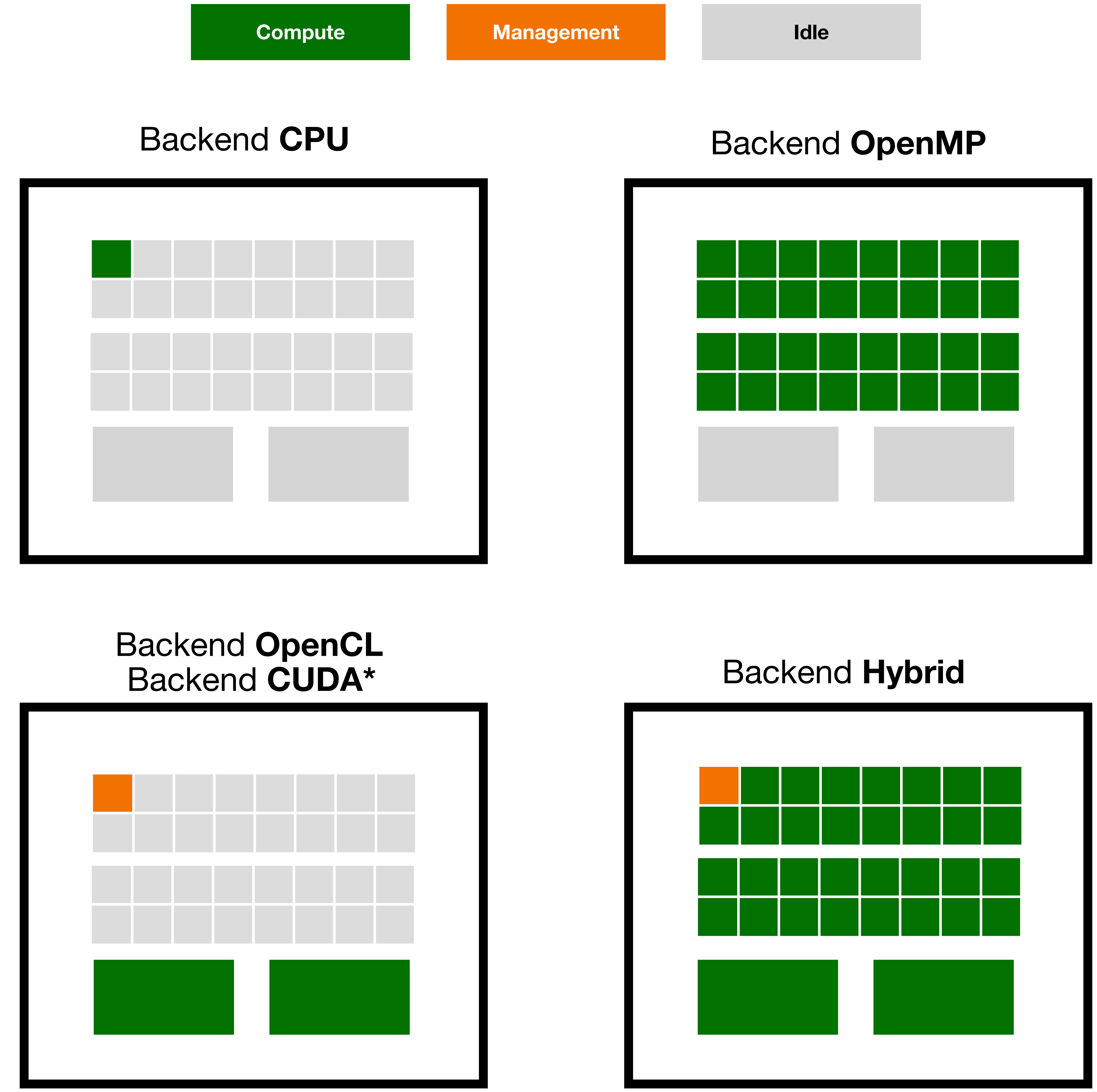
Backend Architecture



Backend Architecture



Node-Level Backend Architecture

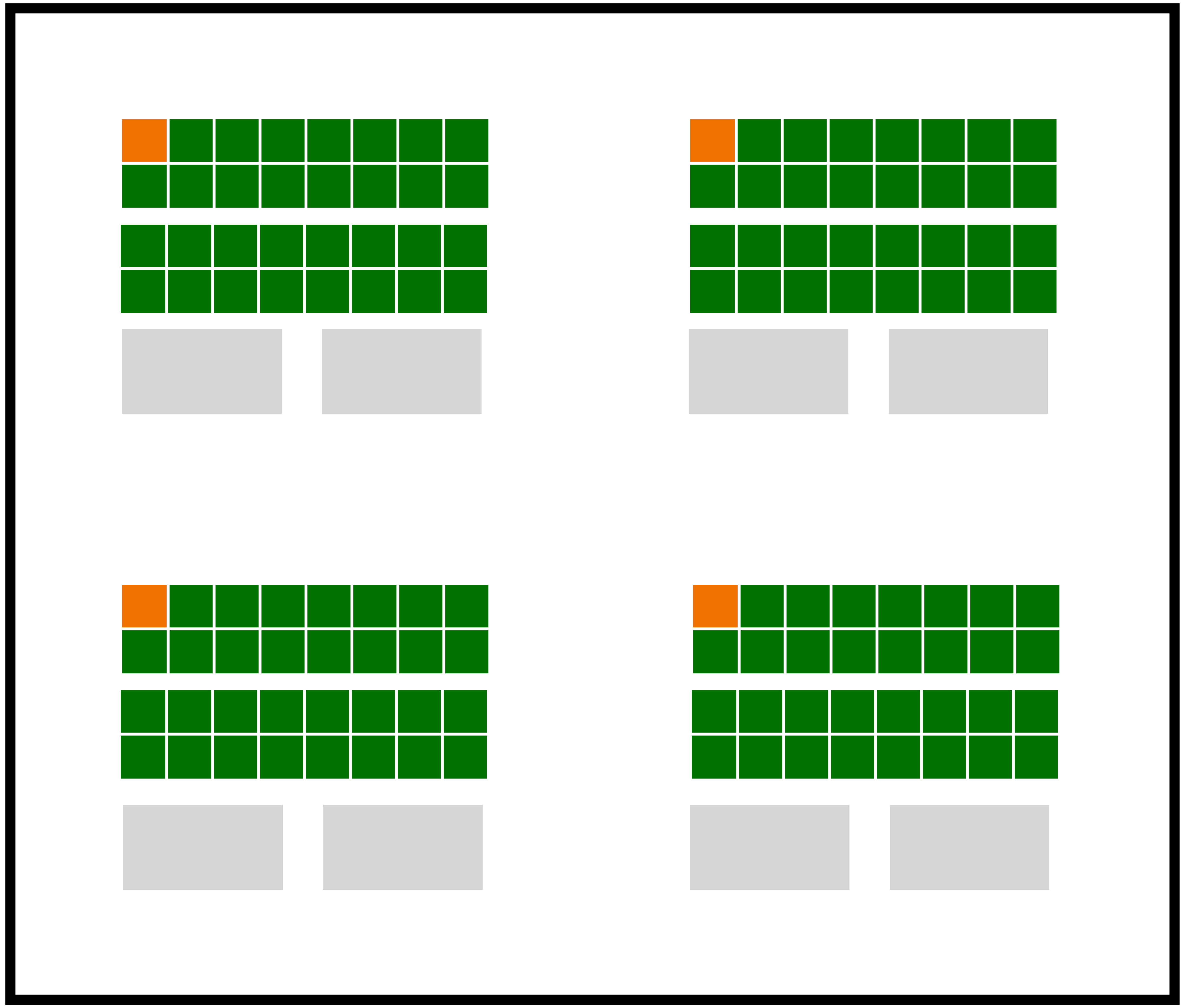


*) NVIDIA GPU accelerators only

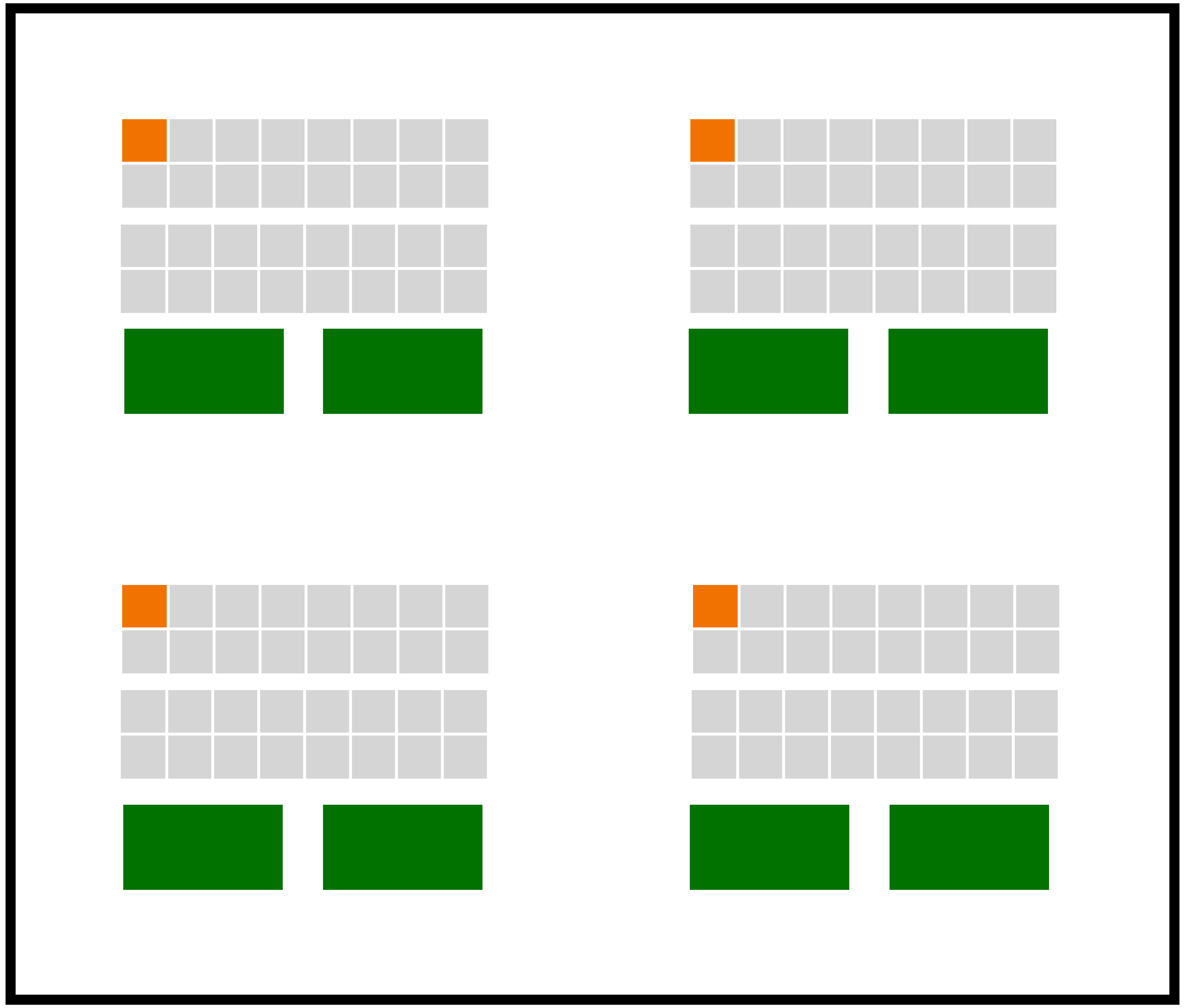
Cluster Backend Architecture

Compute | Management | Idle

Backend **StarPU-MPI**



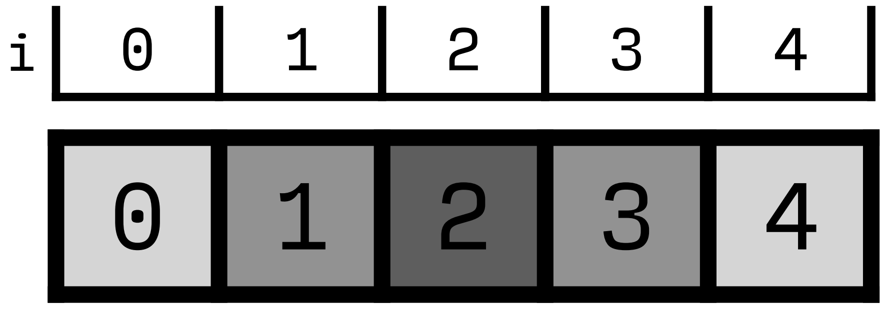
Backend **StarPU-MPI-CUDA***

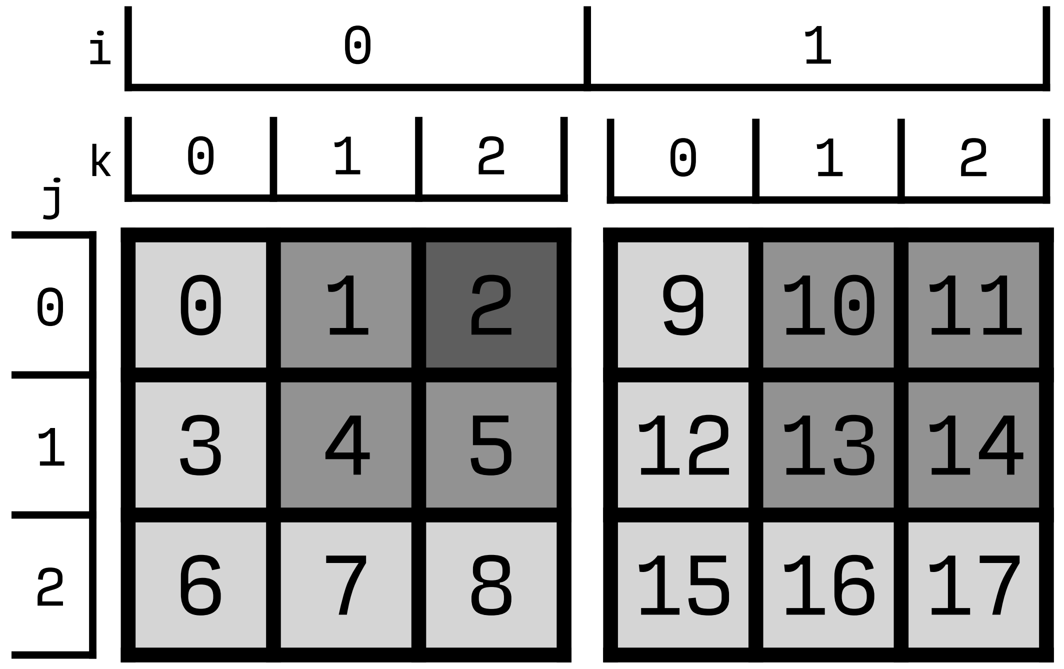


*) NVIDIA GPU accelerators only

Smart Containers

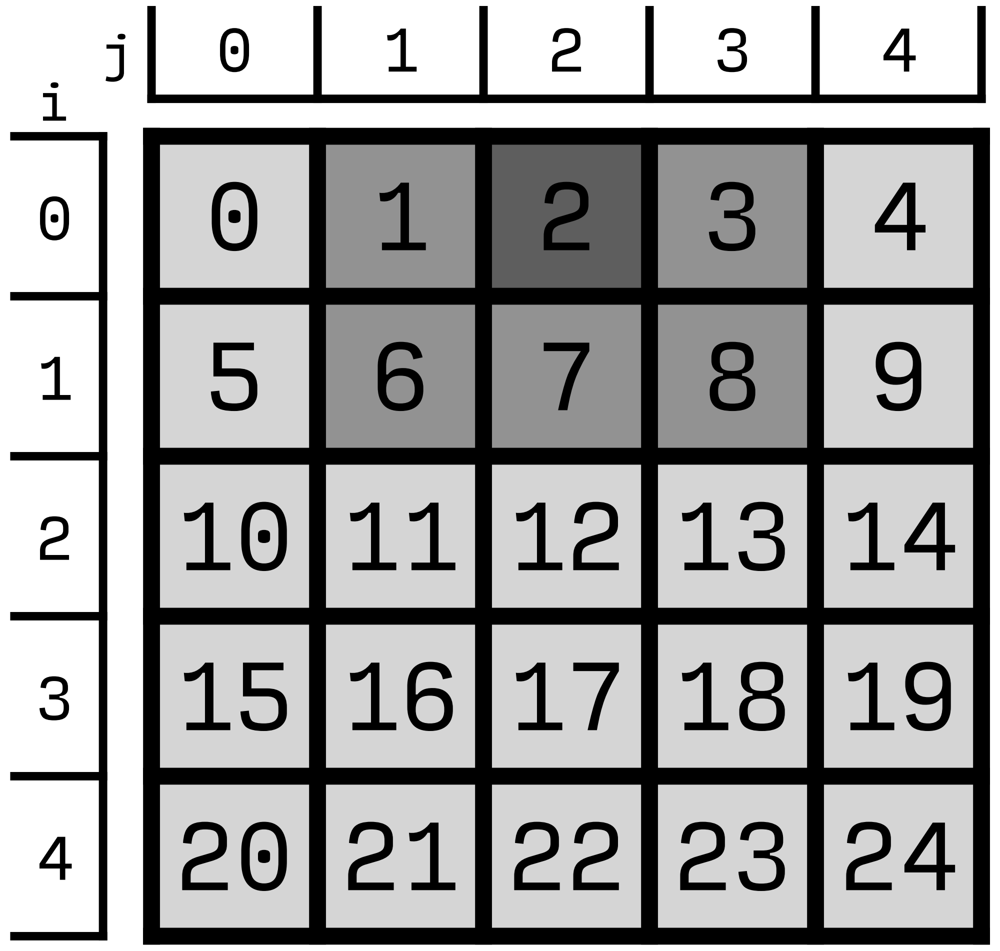
Vector

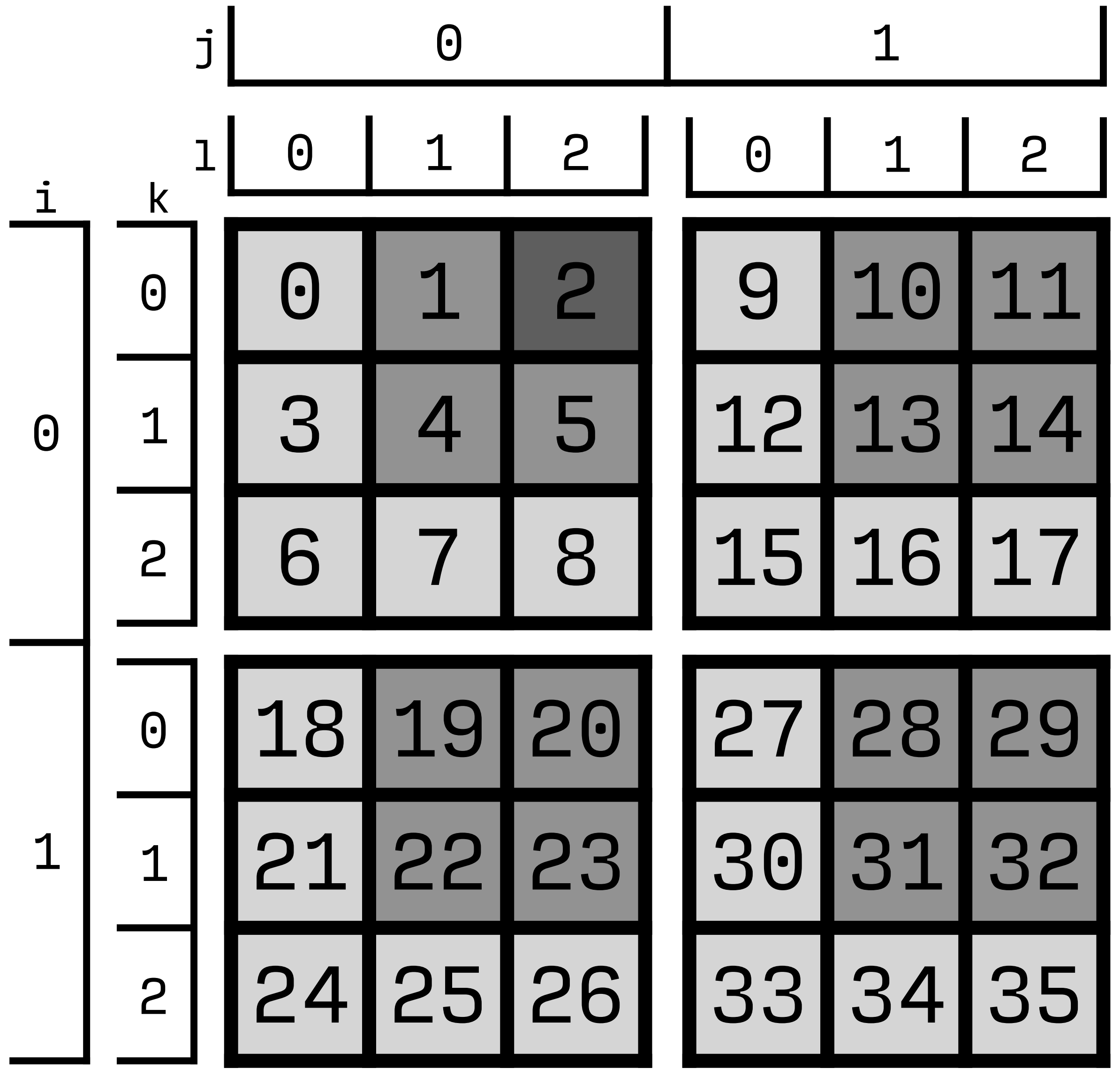




Tensor3

Matrix

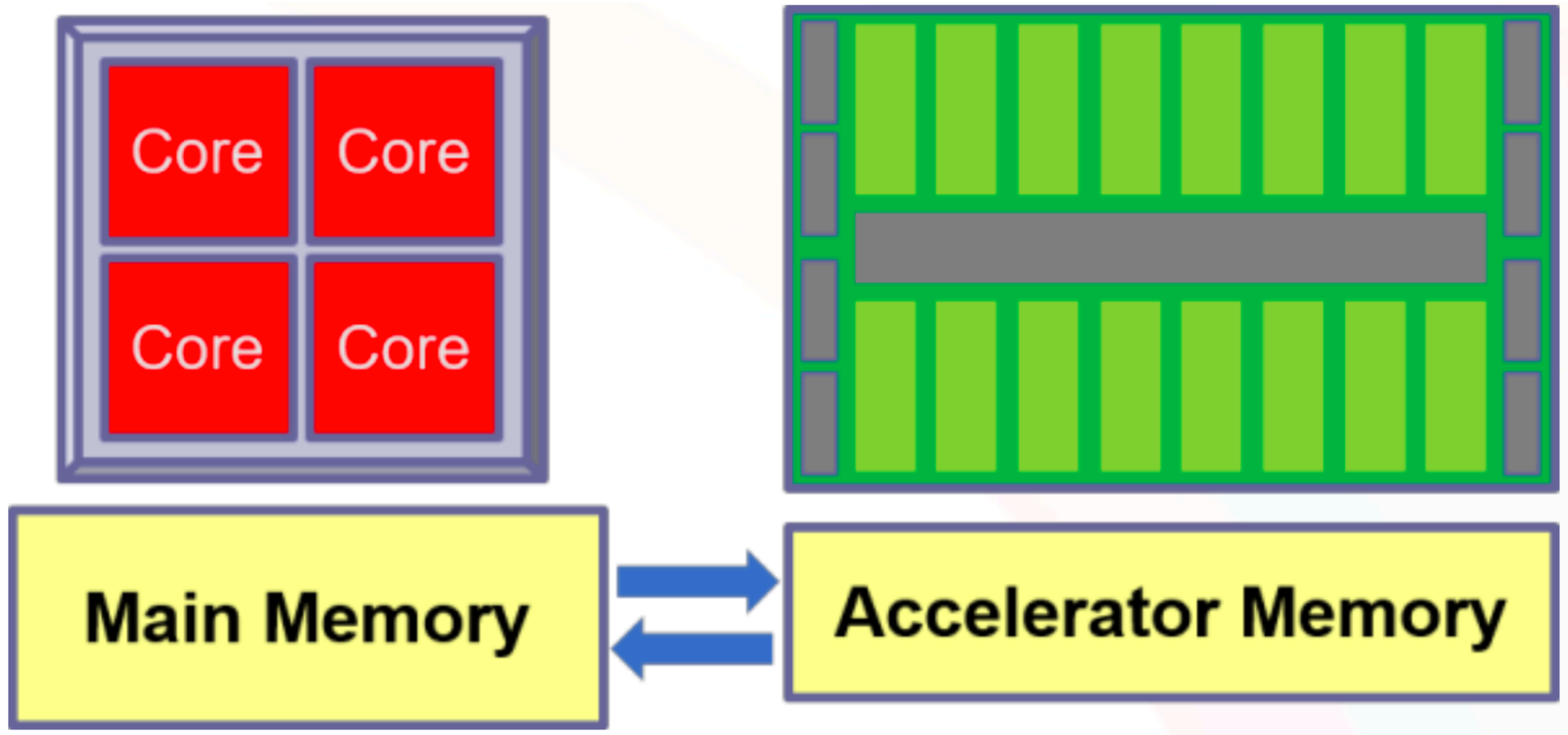




Tensor4

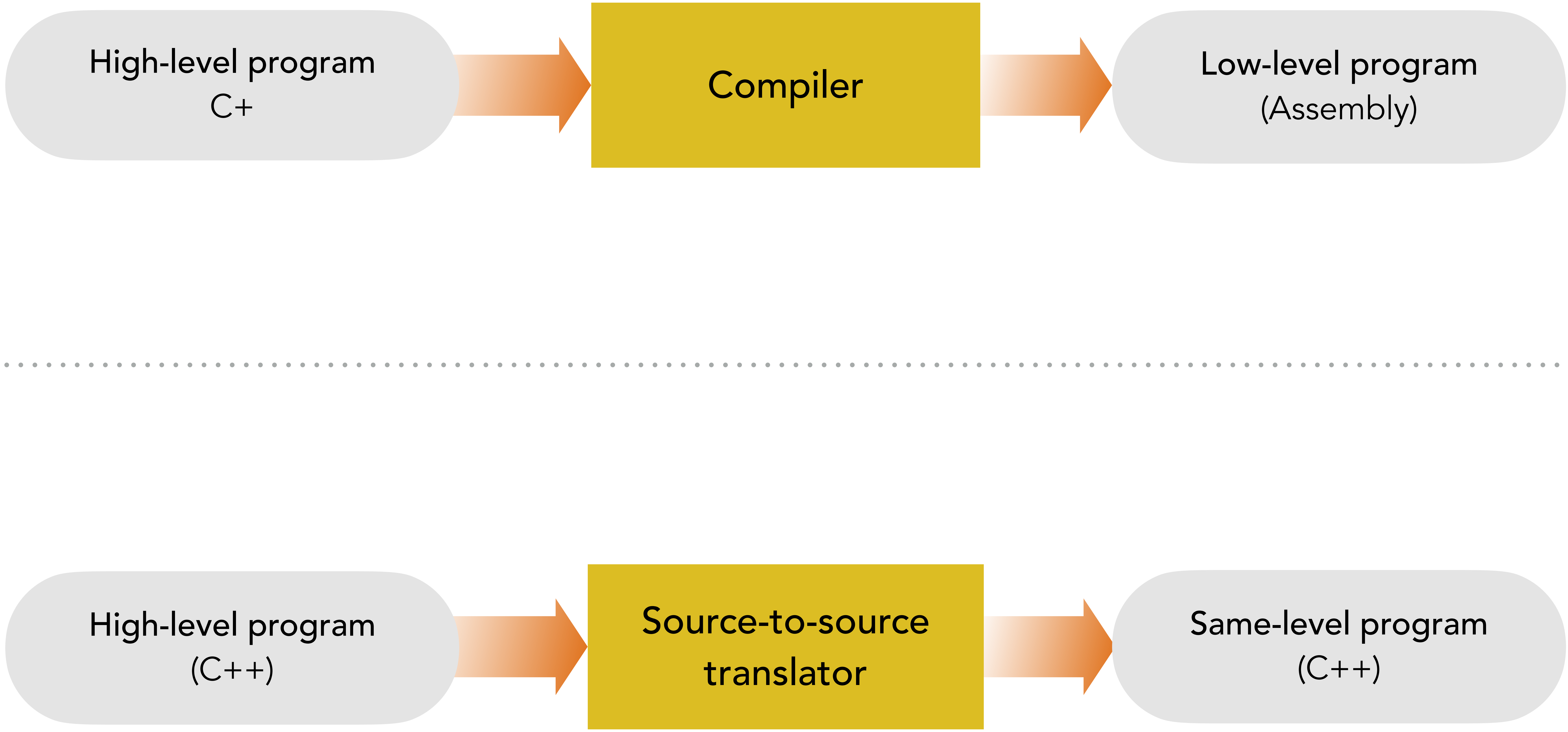
Smart Containers

- C++ template class instance
- Contains:
  - CPU memory buffer **pointer** (alt. StarPU handles)
  - Size information (size, width/height)
  - OpenCL/CUDA/MPI **handles**
  - Consistency states
- Template type can be **custom struct**, but be careful!
  - Data layout not verified across backends/languages

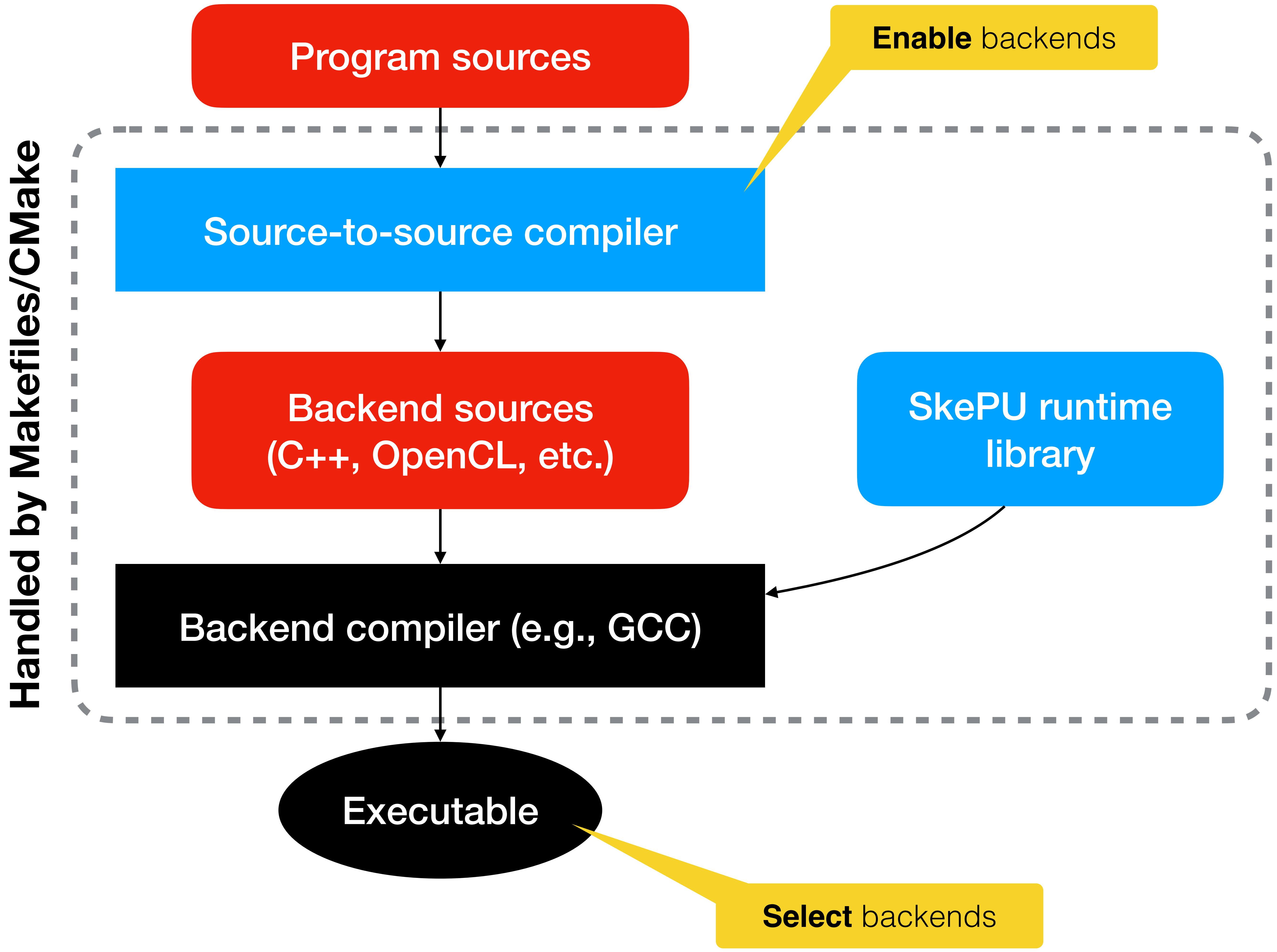


Using SkePU

Source-to-Source Translation



Compilation Architecture

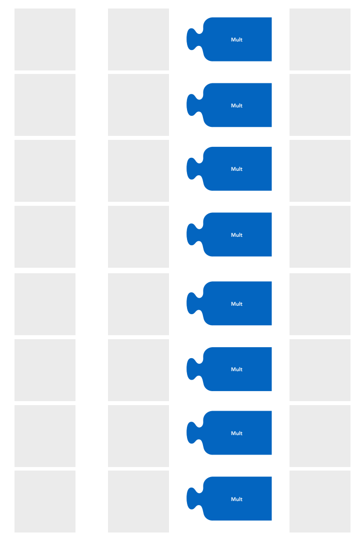


Syntax

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


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


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



User Functions

- User functions are C++ (rather, C) functions
- The signature is analyzed by the pre-compiler to extract the skeleton signature
- Each skeleton has their own expected patterns for UF parameters (but the general structure is shared)
- The UF body is **side-effect free** C (compatible with CUDA/OpenCL)
  - No communication/synchronization
  - No memory allocation
  - No disk IO

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

Flexibility

- **Variable arity** on Map and MapReduce skeletons
- **Index** argument (of current Map'd container element)
- **Uniform** arguments
- Smart container arguments accessible **freely** inside user function
  - **Read**-only / **write**-only / **read-write** copy modes
- User function **templates**

Advanced Example

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


template<typename T>
T user_function(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 * m.cols + i) * v(i);

  return abs(res);
}
```

Advanced/Experim. Features

- Multi-variant user function **specialization**
  - Targeting backend
- Custom **types**
- **Chained** user functions
- In-line **lambda** syntax for user functions
- "**Intrinsic**" functions
  Some functions exist in the standard library of all SkePU backends.
  Examples: sin(x), pow(x, e)

Lambda Syntax

```
auto vec_sum = skepu::Map([](int a, int b)
{
  return a + b;
});

// ...

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

Compilation Options and Backend Selection

Compilation

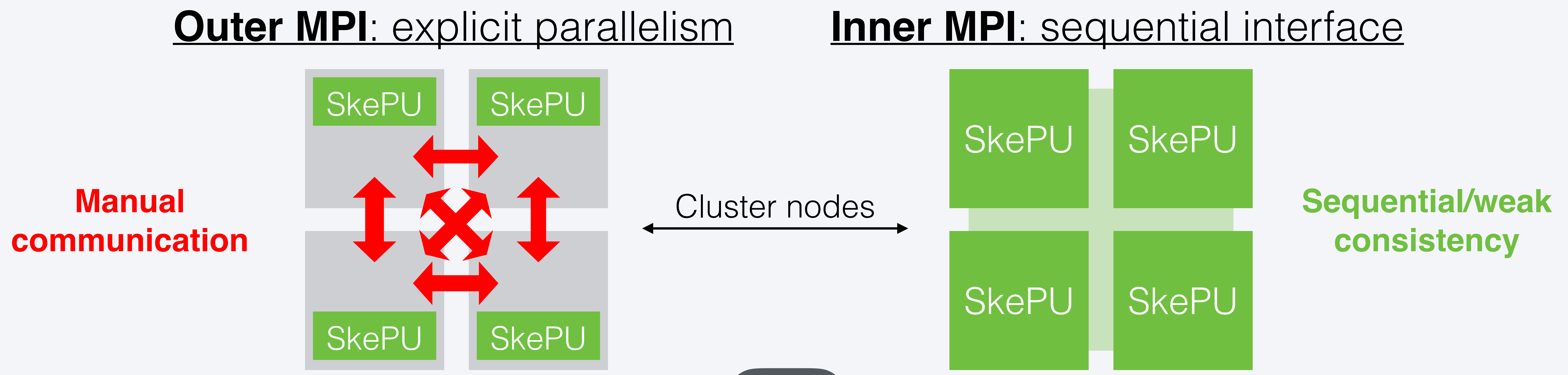
- Precompiler options
  - ```
    skepu-tool
        map.cpp
        -name map_precompiled
        -dir bin
        -openmp -opencl
        -- [Clang flags]
    ```
  - Handled by Makefiles in the binary tutorial distribution
  - Handled by CMake in standard SkePU repository
- Clang flags:
  - Include path to Clang language-headers
  - Include path to SkePU headers
  - Optional OS-specific paths and settings
    ```

Backend Specification

- `auto spec = skepu::BackendSpec{argv[2]};`
  - Sets the **run-time backend** from a string, must match any of:
    - "CPU", "OpenMP", "OpenCL", "CUDA", "Hybrid"
  - Cluster backend is a **compile-time** selection due to SPMD model
    - Uses OpenMP on node level by default
    - Optional GPU backend selection
- Global backend spec for all skeletons, overridden by instance-specific spec.
  - `skepu::setGlobalBackendSpec(spec);`

Cluster backend

- SkePU can **target clusters**
  - Any valid SkePU 3 program needs not be syntactically changed to use the cluster backend ("inner MPI mode")
  - SkePU can also run on clusters by using skeletons locally on each node ("outer MPI mode": MPI+SkePU)
- Implementation atop *StarPU*
  - Challenges: SPMD-style; C-level APIs (StarPU + MPI)

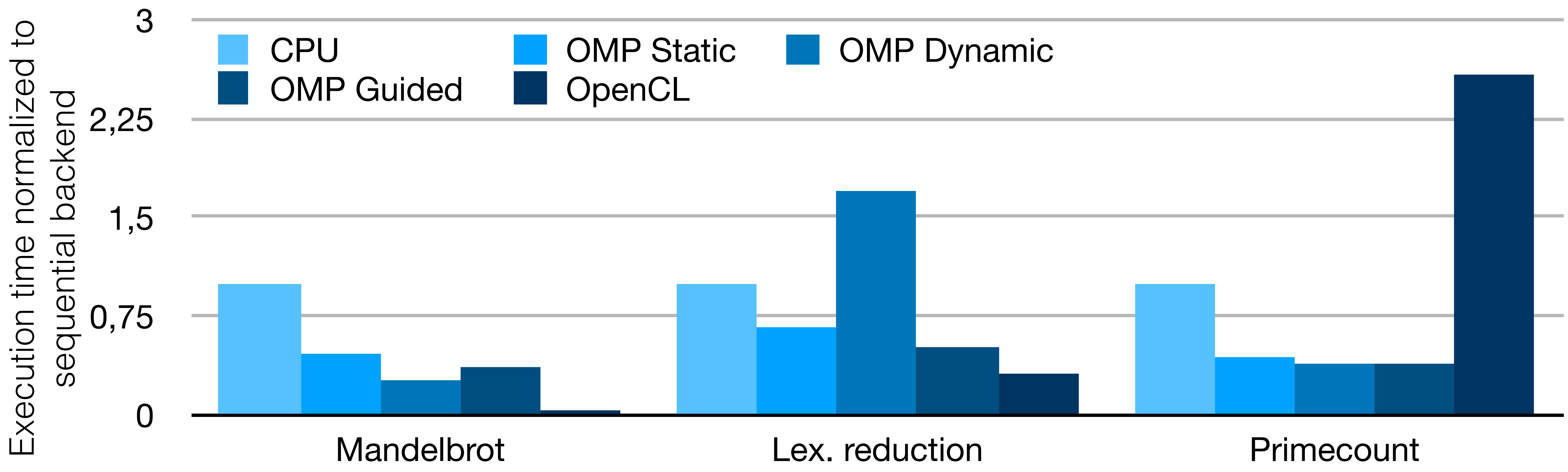


Backend Specification

- **Backend specifications** can furthermore contain execution parameters
  - Number of **threads**
  - Number of **GPUs**
  - GPU block size, thread count hinting
  - OpenMP **scheduling** mode
    - Static (default) / dynamic / guided / auto
    - Chunk size for dynamic scheduling
  - **Partition ratio** for Hybrid backend

Backend Specification

- Three benchmarks that illustrate the effects of dynamic scheduling in OpenMP backend
- All variants are SkePU backend executions
  - Sequential CPU and OpenCL included for reference

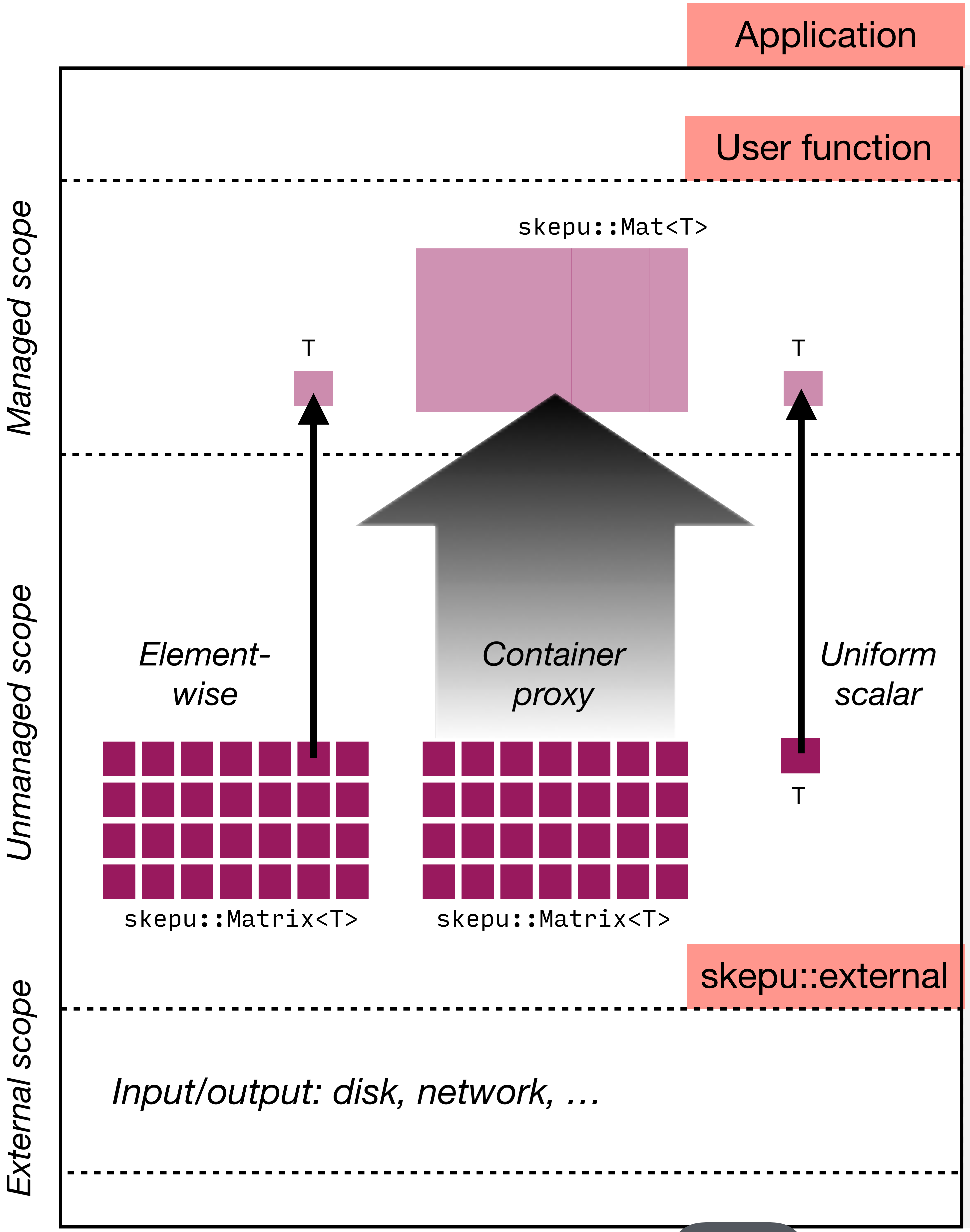


Smart Containers & Consistency Model

Smart Containers

- SkePU provides three different "scopes" with different syntax and consistency rules
- Inside a user function: **Managed scope**
  - C-style syntax allowed, no global synchronization (compare with GPU kernel code)
- Outside user function: **Unmanaged scope**
  - Regular single-threaded C++-land with smart container objects providing weak consistency
- For global side effects: **External scope**
  - In SPMD model (clusters), the C++ program does not run single-threaded
  - External construct wraps file read/write, logging, etc.

Consistency Model




```
#include <skepu>

T func(T e, skepu::Mat<T> p, T s)
{
    /* iterate through p */
    return /* … */;
}

auto skel = skepu::Map(func);

skepu::Matrix<T> m1(N), m2(N), res(N);

T some_val;

skel(res, m1, m2, some_val);

skepu::external(skepu::read(res), []()
{
    my_serialization(res, my_file);
});
```

Weak Consistency

- `container.flush();`
  - Flushes and ensures that the CPU buffer contains up-to-date values.
- `container(i) = value;`
  - Direct element access into the raw CPU buffer.
  - Weak consistency, unless compile-time flag enabled (optional for debugging purposes)
-
```
skepu::external(skepu::read(particles), [&]{
    save_step(particles, outfile);
});
```
  - Manages IO and other external operations in SPMD mode

SkePU

DEMO

Skeletons
In Depth

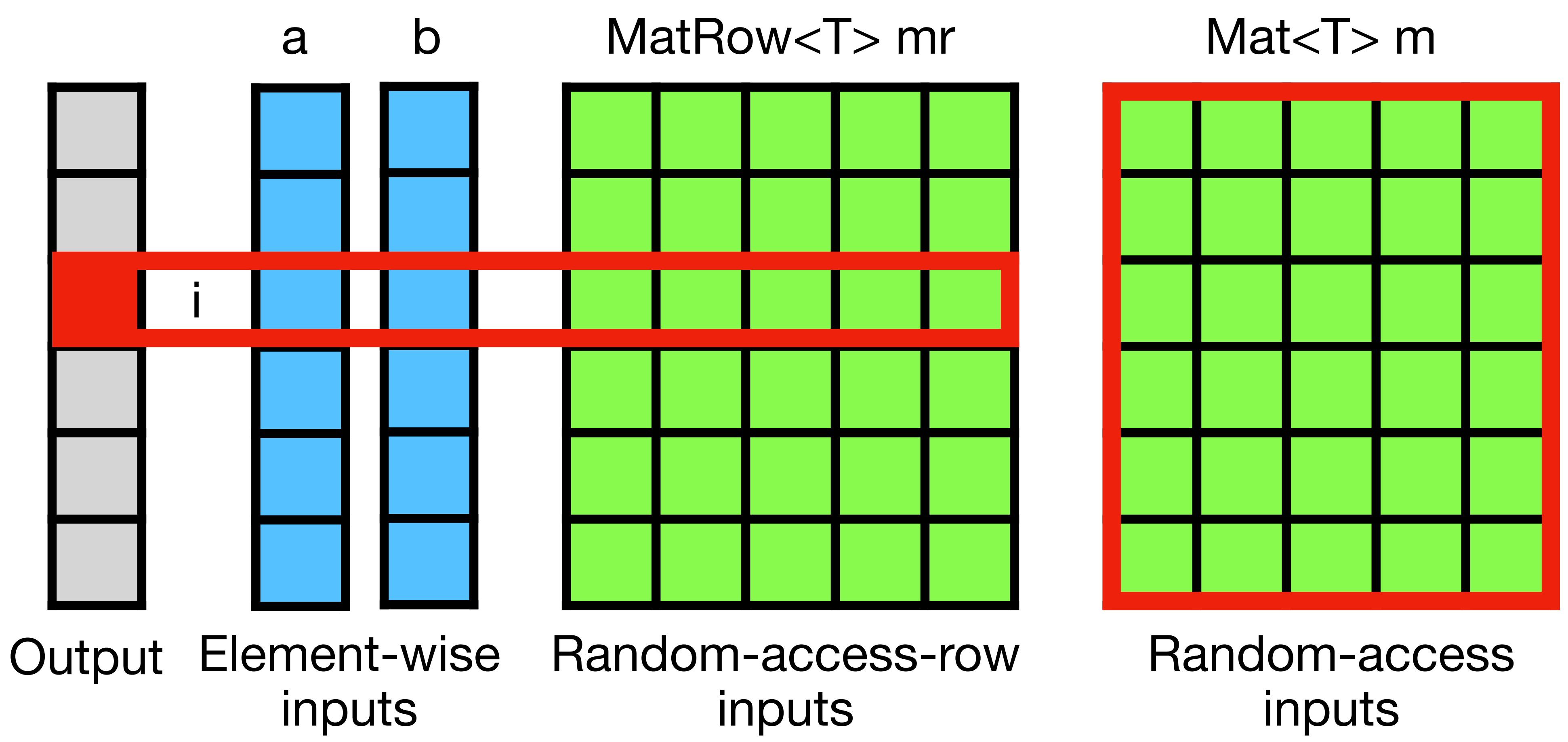
Skeleton Features

Feature \ Skeleton	Map	MapPairs	MapOverlap	Reduce	Scan	MapReduce	MapPairsReduce
Elwise dimension in	1–4	1	1–4	1–4**	1	1–2	1
Elwise dimension out	Same as *in*	2	Same as *in*	0–1	1	0	1
Indexed	Yes	Yes	Yes	-	-	Yes	Yes
Multi-return	Yes	Yes	Yes	-	-	Yes	Yes
Elwise parameters	Variadic	Variadic x2	1***	*	*	Variadic	Variadic x2
Full proxy parameters	Variadic	Variadic	Variadic	-	-	Variadic	Variadic
Uniform parameters	Variadic	Variadic	Variadic	-	-	Variadic	Variadic
Region proxy	-	-	Yes	-	-	-	-
MatRow/MatCol proxy	Yes	Yes	-	-	-	Yes	Yes

Footnotes	
*	Parameters to the user functions can be raw elements from the container or partial results, depending on evaluation order.
**	Dimensions higher than 2 are linearized in the current implementation.
***	A region of elements surrounding the current index is supplied.

Map

Map Parameter Groups



- Three groups of user function parameters:
  - **Element-wise**
    Only one element per user function call
  - **Random-access containers**
    Replicated for each memory space (e.g. GPUs)
    Proxy types Vec<T> and Mat<T> in user function
  - **Uniform scalars**
    Same values everywhere
- Argument groups are variadic (flexible count, including 0)
- Above order must be obeyed (element-wise first etc.)
- The parallelism/number of user function invocations is always determined by the return container (first argument), also in case of element-wise arity of 0.
- Also applies to **MapReduce**, **MapOverlap, MapPairs, MapPairsReduce**!

Map Example

```
float sum(float a, float b)
{
   return a + b;
}

Vector<float> vector_sum(Vector<float> &v1, Vector<float> &v2)
{
   auto vsum = Map<2>(sum);
   Vector<float> result(v1.size());
   return vsum(result, v1, v2);
}
```

Strided Map

- Map (and MapReduce) can index containers in a **strided fashion**
  - `instance.setStride(…);`
- Positive strides indicate length of jump **between elements**
- Negative strides in addition **reverses** indexing order
- Unit, non-unit, and negative strides can be mixed in call

```
auto addr = skepu::Map(add);
auto addrsum = skepu::MapReduce(add, add);

constexpr size_t N{8};
skepu::Vector<int> v1(N*4), v2(N*3), r(N*2), r2(N);

// MAP, POSITIVE STRIDES
addr.setStride(2, 4, 3);
addr(r, v1, v2);

// MAPREDUCE, POSITIVE STRIDES
addrsum.setStride(4, 3);
int res = addrsum(v1, v2);

// MAP, NEGATIVE STRIDES
addr.setStride(2, -4, -3);
addr(rc, v1, v2);
```

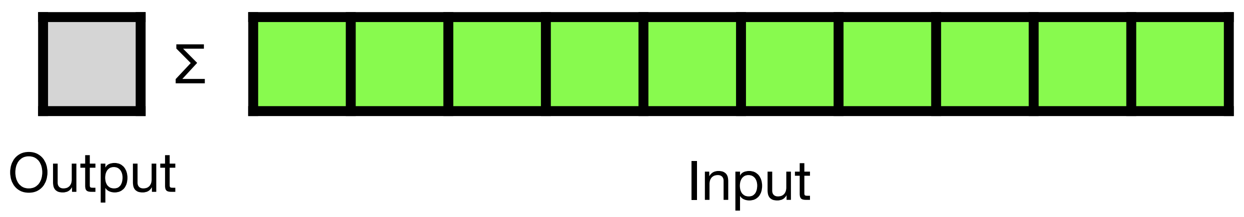
Map – Iterators

- Optionally, use iterators with Map (or MapReduce):
  - `mapper(r.begin(), r.end(), v1.begin(), v2.begin());`

Reduce

Reduce Modes

- **1D Reduce**

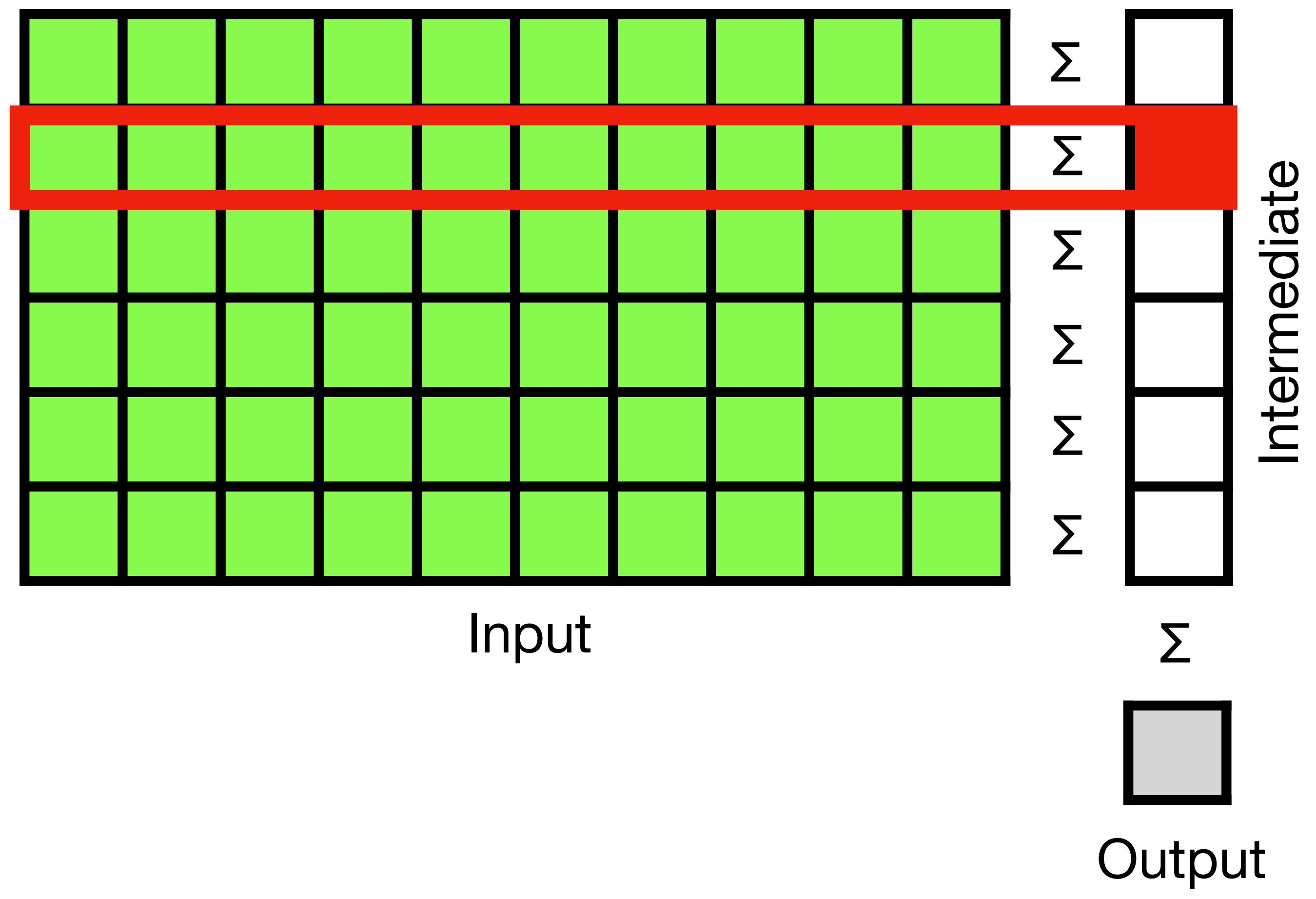


  - Regular Vector
  - Matrix RowWise (returns Vector)
  - Matrix ColWise (returns Vector)

- **"2D" Reduce**

```
instance.setReduceMode(ReduceMode::RowWise) // default
instance.setReduceMode(ReduceMode::ColWise)
```

  - Regular Matrix (treated as a vector)



- `instance.setStartValue(value)`
  - Set Reduction start value. Defaults to 0-initialized.

Reduce Example

```
float min_f(float a, float b)
{
  return (a < b) ? a : b;
}

float min_element(Vector<float> &v)
{
  auto min_calc = Reduce(min_f);
  return min_calc(v);
}
```

Reduce Example

```
float plus_f(float a, float b)
{
  return a + b;
}

float max_f(float a, float b)
{
  return (a > b) ? a : b;
}

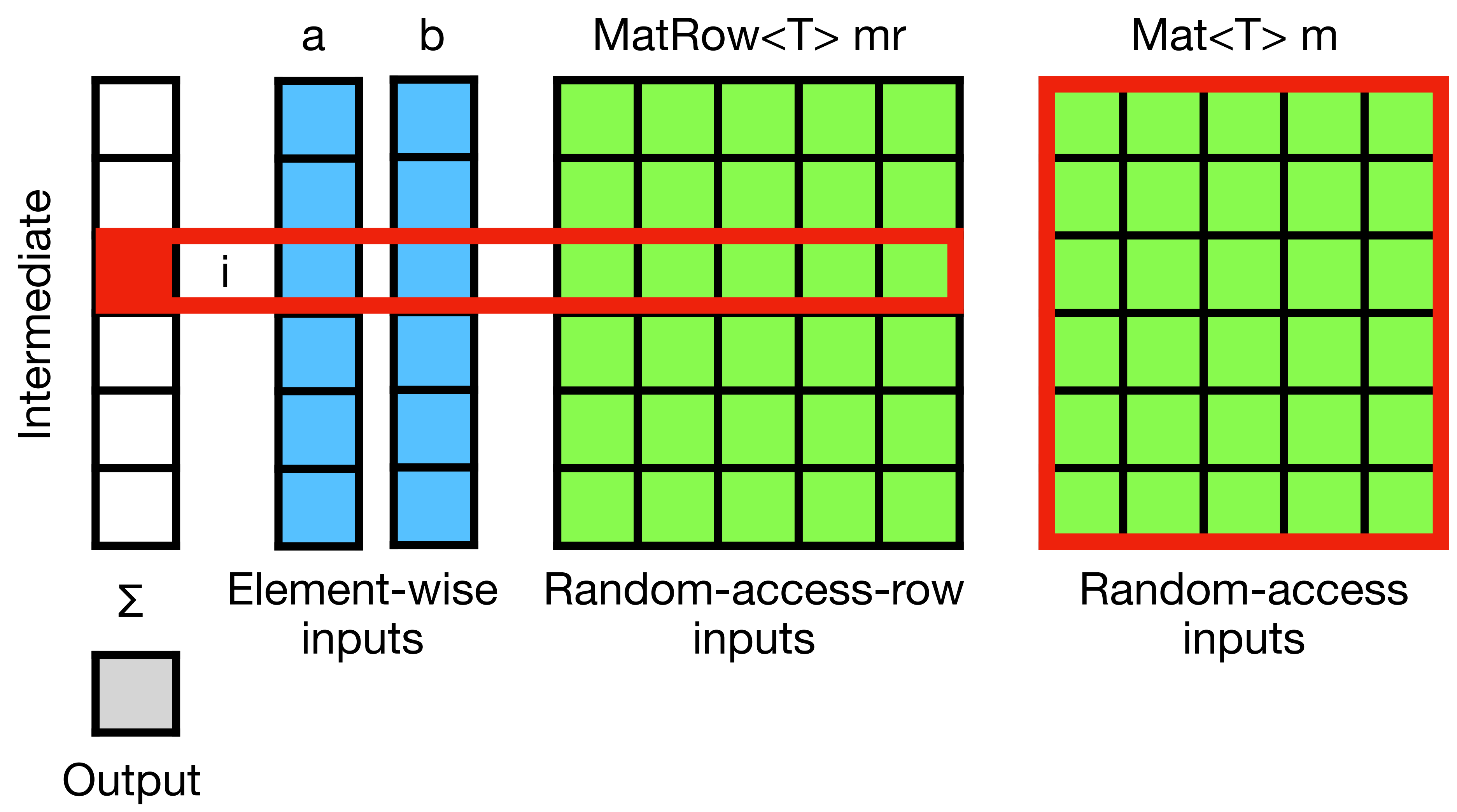
auto max_sum = skepu::Reduce(plus_f, max_f);

max_sum.setReduceMode(skepu::ReduceMode::RowWise);
r = max_sum(m);
```

MapReduce

MapReduce

- `instance.setDefaultSize(size_t)`
  - When the element-wise arity is 0, this controls the number of user function invocations (That is, the size of the "virtual" temporary container in between the Map and Reduce steps)
- `instance.setStartValue(value)`
  - Set Reduction start value. Defaults to 0-initialized.



MapReduce Example

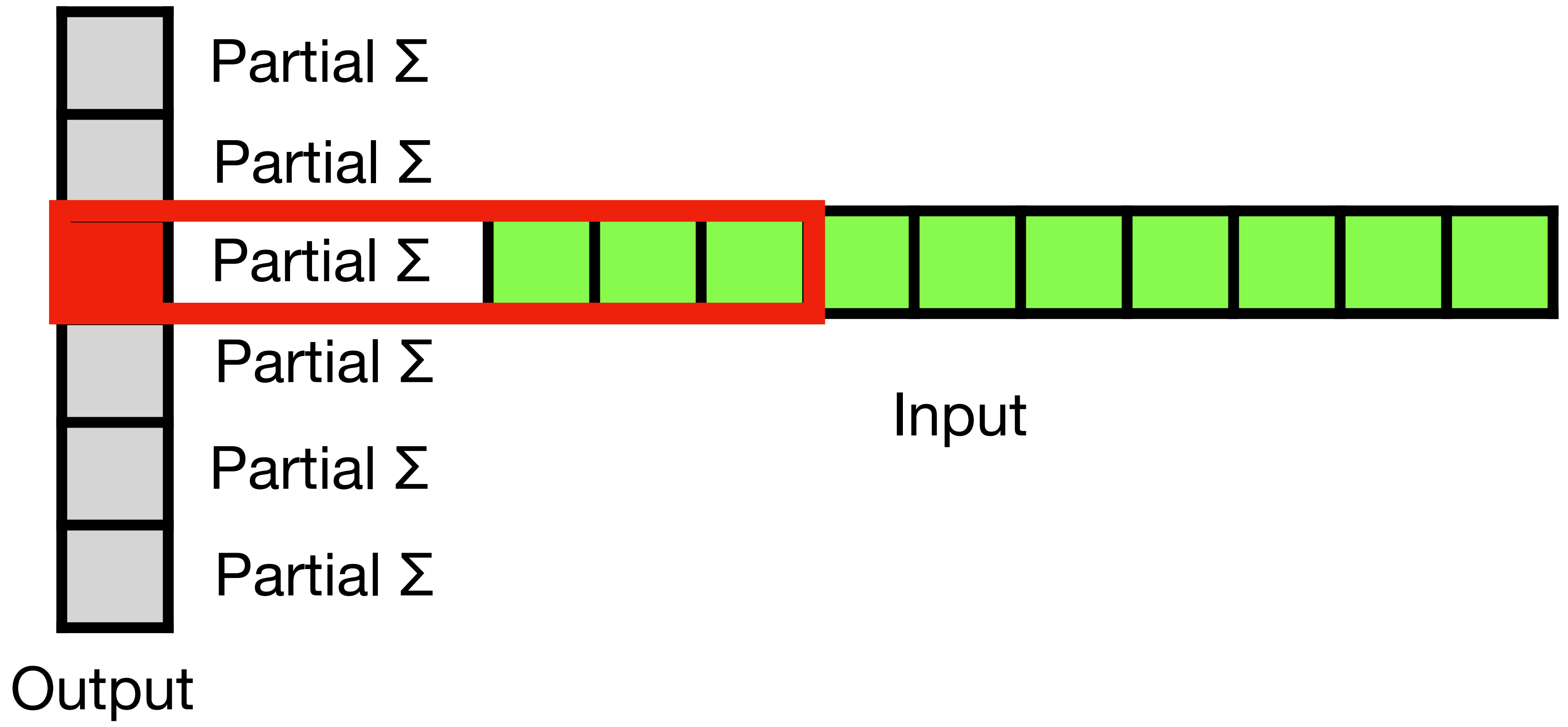
```
float add(float a, float b)
{
  return a + b;
}

float mult(float a, float b)
{
  return a * b;
}

float dot_product(Vector<float> &v1, Vector<float> &v2)
{
  auto dotprod = MapReduce<2>(mult, add);
  return dotprod(v1, v2);
}
```

Scan

Scan



- `instance.setScanMode(mode)`
  - Set the scan mode:
    `ScanMode::Inclusive (default)`
    `ScanMode::Exclusive`
- `instance.setStartValue(value)`
  - Set start value of scans. Defaults to 0-initialized.

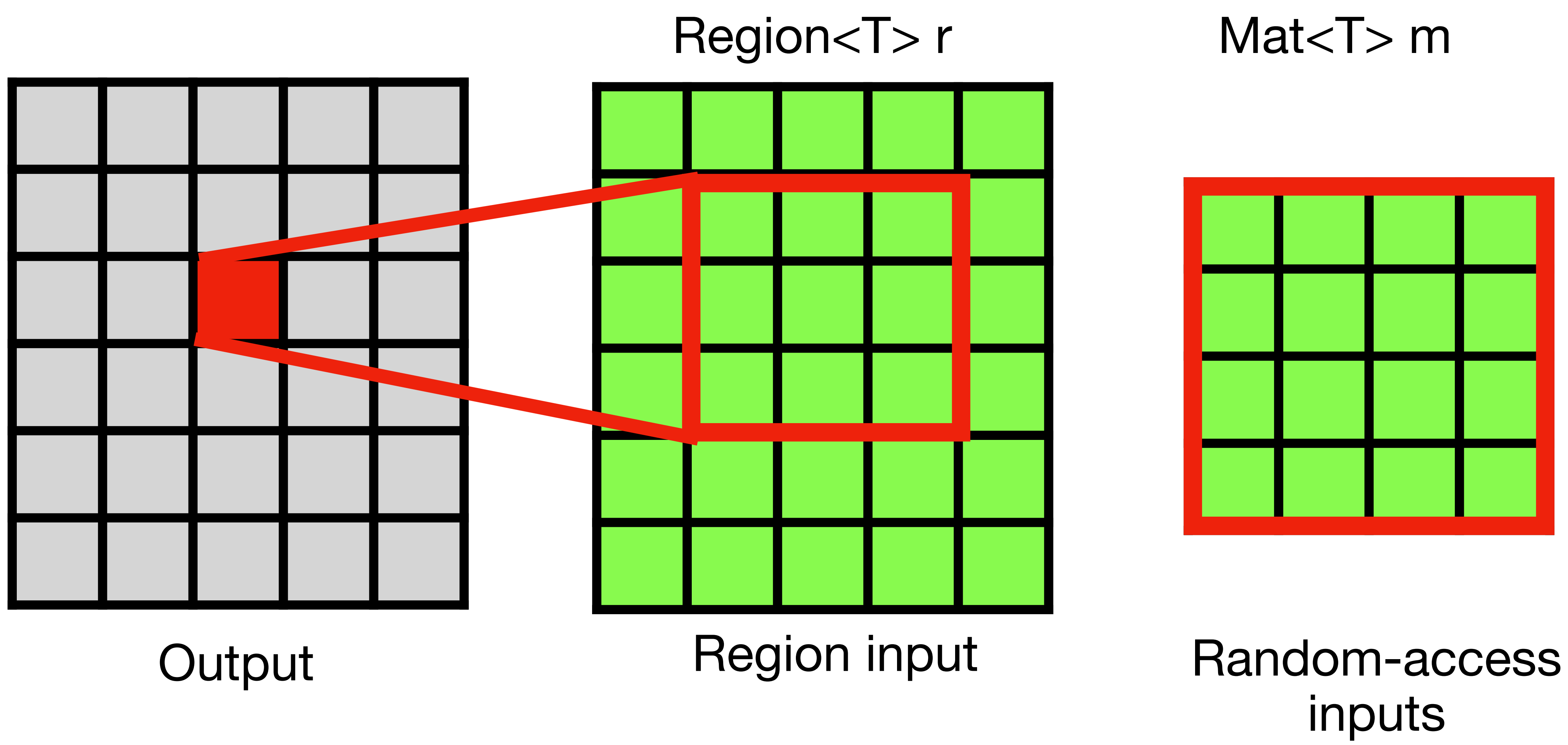
Scan Example

```
float max_f(float a, float b)
{
  return (a > b) ? a : b;
}

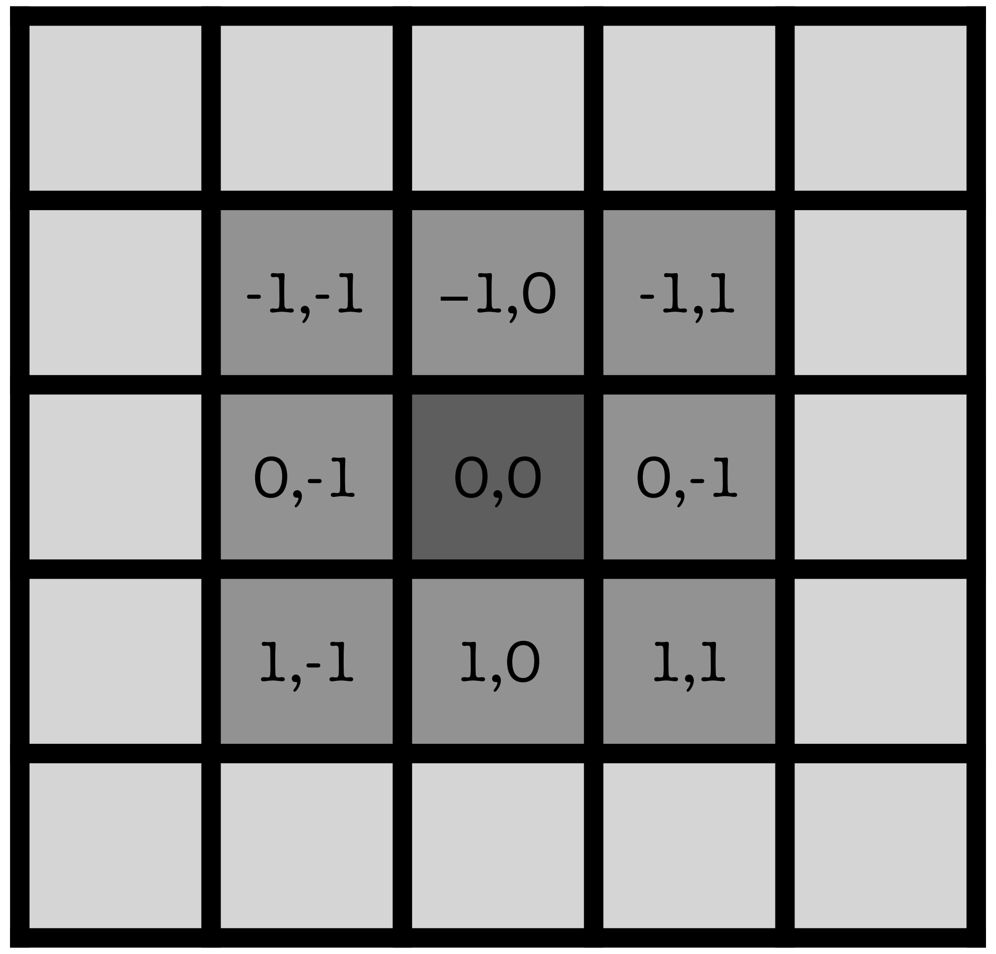
Vector<float> partial_max(Vector<float> &v)
{
  auto premax = Scan(max_f);
  Vector<float> result(v.size());
  return premax(result, v);
}
```

MapOverlap

- Region of the input container accessible in user function
- In addition to optional full random-access parameters



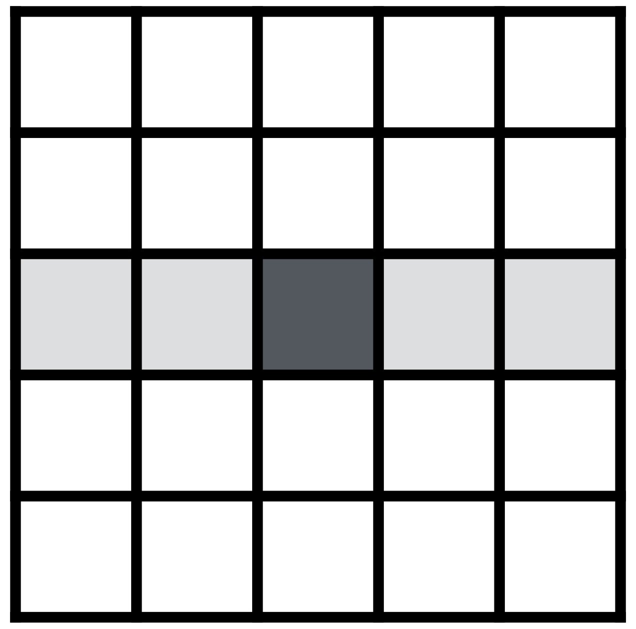
- Region indexing is 0-centered in each dimension

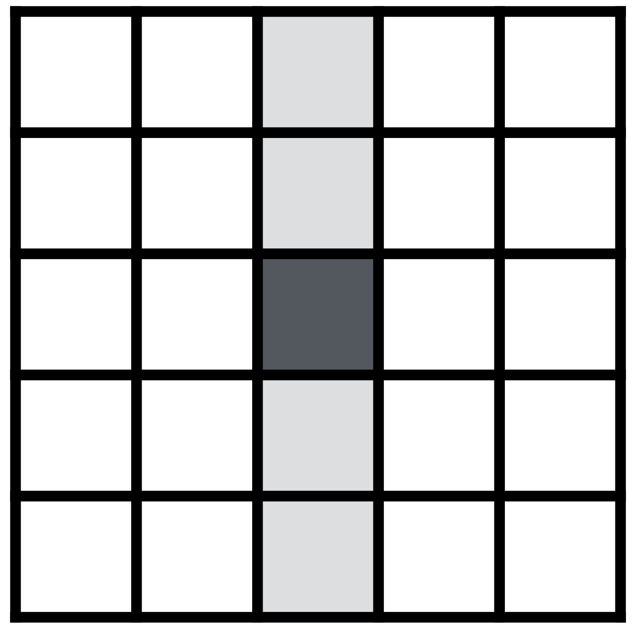


MapOverlap Modes

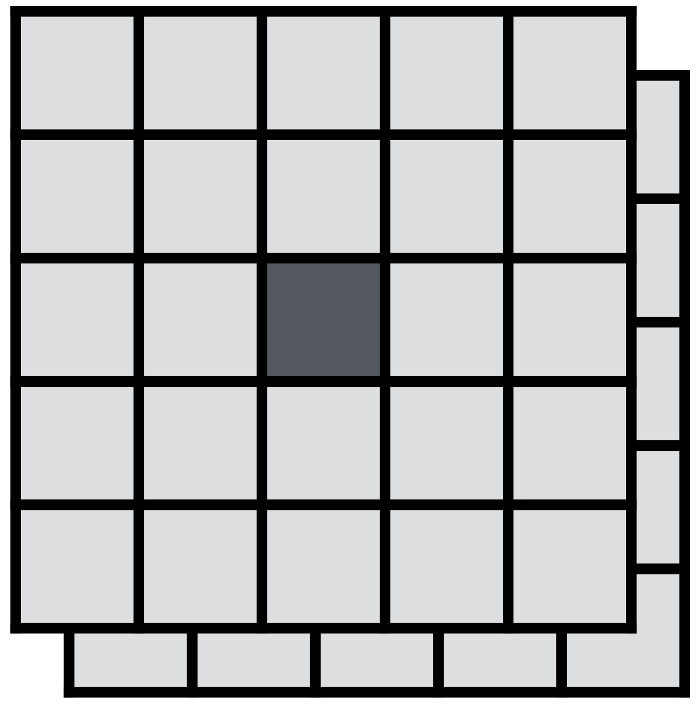
- **1D MapOverlap**
  - Regular Vector

  - Matrix RowWise `instance.setOverlapMode(Overlap::RowWise) // default`
  - Matrix ColWise `instance.setOverlapMode(Overlap::ColWise)`





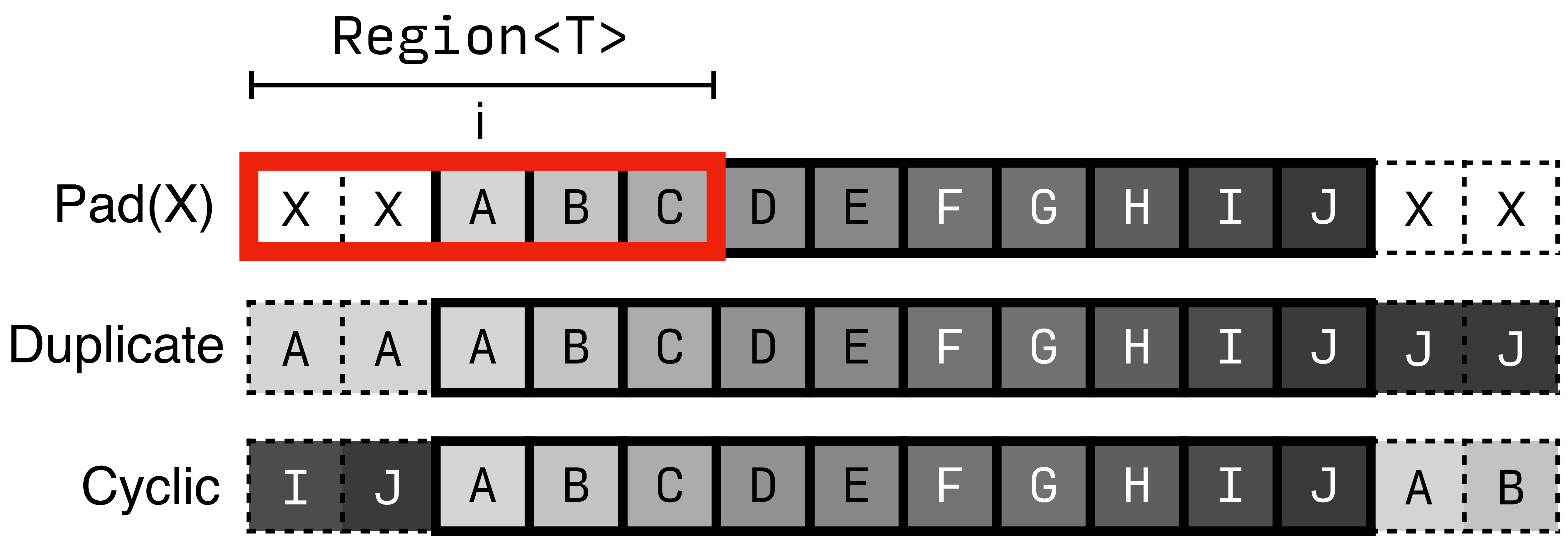
- **2D MapOverlap** on Matrix
- **3D MapOverlap** on Tensor3
- **4D MapOverlap** on Tensor4



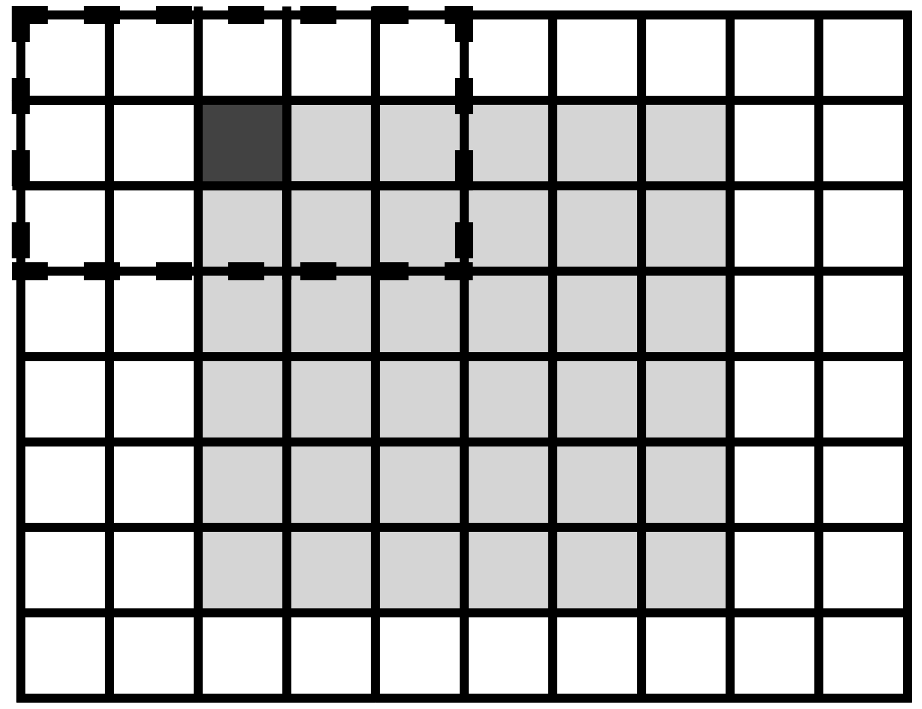
MapOverlap Edge Mode

- `instance.setOverlap(i [, j [, k [, l]]])`
  - Set overlap radius in each dimension
- `instance.setEdgeMode(mode)`
  - `Edge::Pad`
    - `instance.setPad(pad)` – set padding value
  - `Edge::Duplicate` (default for 1D)
  - `Edge::Cyclic`
  - `Edge::None` (default for other dimensions)

MapOverlap Edge Mode



None: Updates only ”safe” elements



MapOverlap 1D Example

```
float conv(skepu::Region1D<float> r, int scale)
{
   return (r(-2)*4 + r(-1)*2 + r(0) + r(1)*2 + r(2)*4) / scale;
}

Vector<float> convolution(Vector<float> &v)
{
   auto convol = MapOverlap(conv);
   Vector<float> result(v.size());
   convol.setOverlap(2);
   return convol(result, v, 10);
}
```

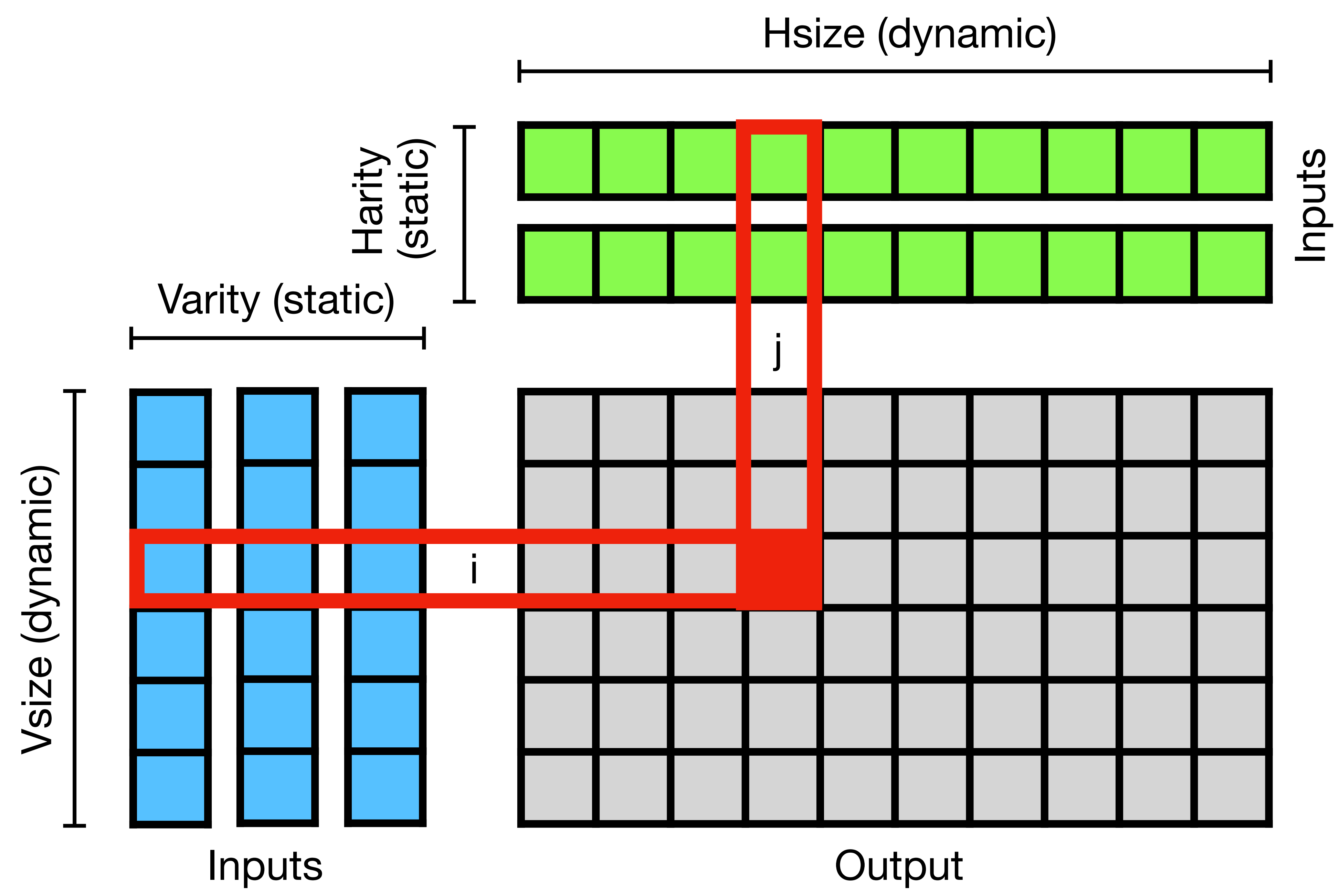
MapOverlap 2D

```
float over_2d(skepu::Region2D<float> r, const skepu::Mat<float> stencil)
{
  float res = 0;
  for (int i = -r.oi; i <= r.oi; ++i)
    for (int j = -r.oj; j <= r.oj; ++j)
      res += r(i, j) * stencil(i + r.oi, j + r.oj);
  return res;
}
```

MapPairs

MapPairs

- Generalized cartesian product / outer product of vectors
- Always results in a Matrix
- Syntactically, Map extended with another variadic elwise parameter group



MapPairs Example

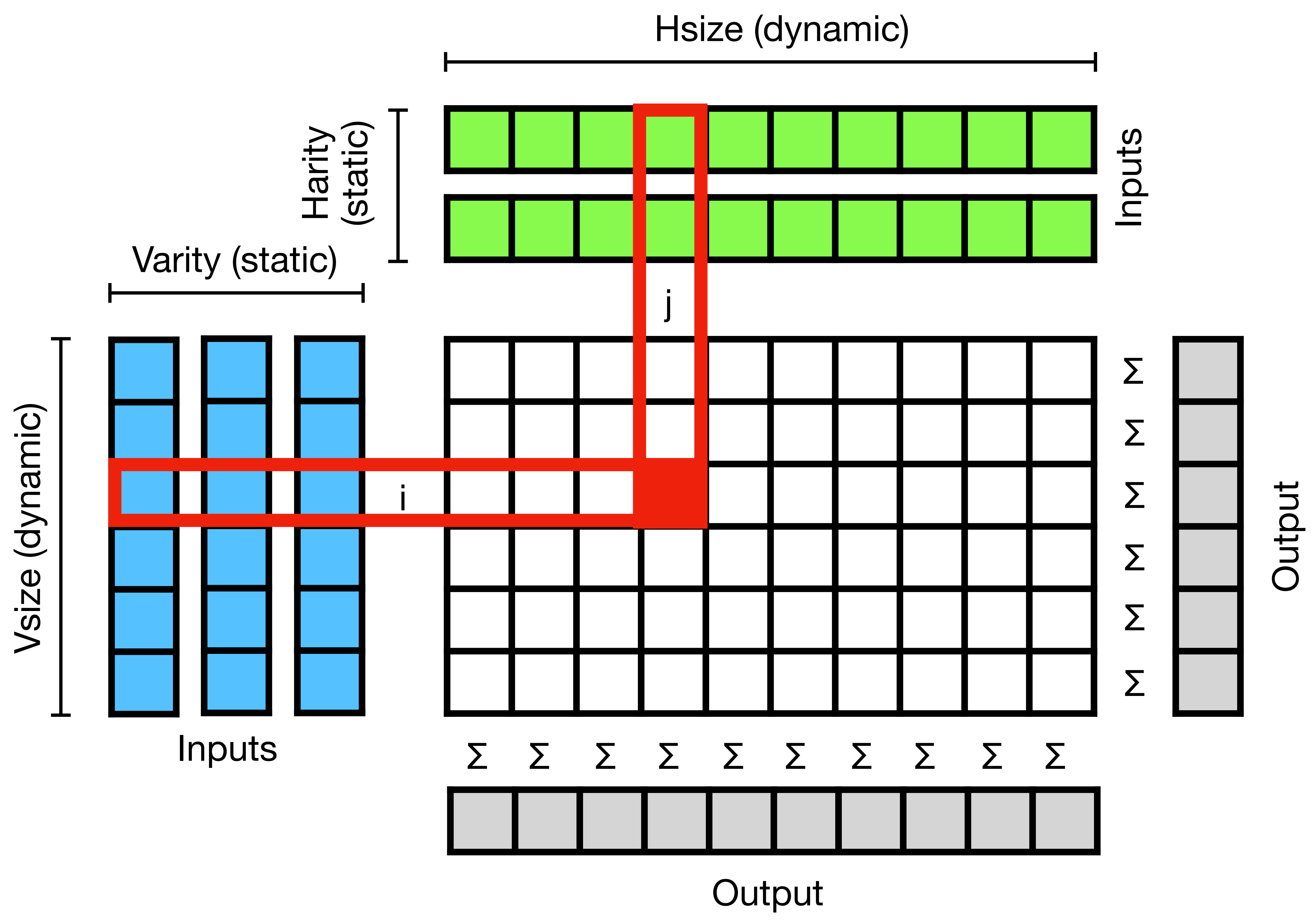
```
float ger_uf(float x, float y, float alpha)
{
  return alpha * x * y;
}

void outer_product(
  float alpha,
  Vector<float>& x,
  Vector<float>& y,
  Matrix<float>& A,
)
{
  auto skel = skepu::MapPairs<1, 1>(ger_uf);
  skel(A, x, y, alpha);
}
```

MapPairsReduce

MapPairsReduce

- MapPairs chained with reduction along rows or columns of the matrix
- Efficient: matrix need not be allocated in full



Variadic Return

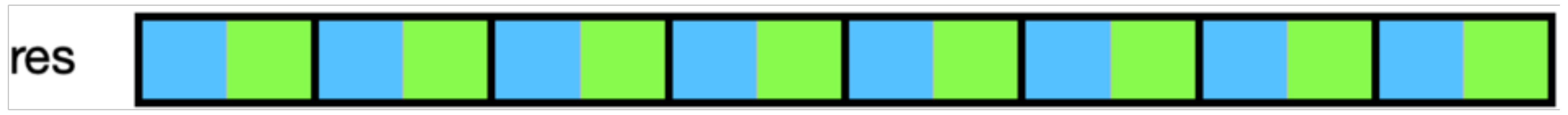
Variadic Return

- Map*** skeletons optionally can return tuples of elements from the user function

- Compared to custom struct types, this stores results in several disjoint arrays

```
skel( resA, resB, inputs... );
```


```
skel( res, inputs... );
```



Variadic Return

```
skepu::multiple<int, float>
test_f(skepu::Index1D index, int a, int b,
skepu::Vec<float> c, int d)
{
  return skepu::ret(a * b, (float)a / b);
}

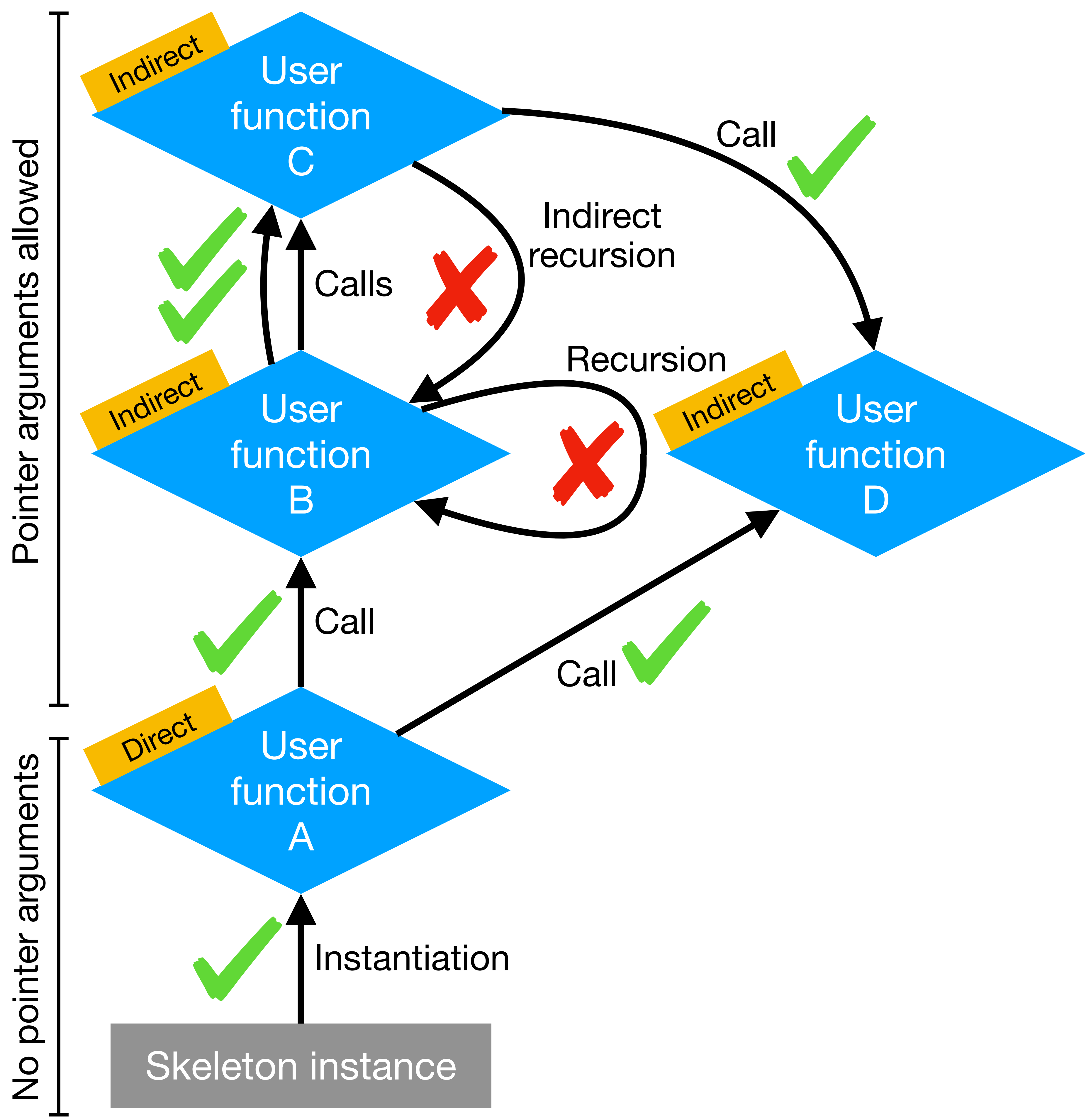
// …

auto test = skepu::Map<2>(test_f);
test(r1, r2, v1, v2, e, 10);
```

Nested User Functions

Nested User-functions

- User functions may call other functions
- The callee will be processed as its own user function by the precompiler
- Some restrictions are relaxed over the uf-uf barrier: pointers may be allowed



SkePU Standard Library

Standard Library Overview

NB: Very new and subject to changes!

- SkePU "standard" library is a collection of **supporting APIs** to further simplify SkePU programming by eliminating the need for common boilerplate code.

- Some parts have framework integration, but most is implemented at **user-level** and can be used as reference for SkePU application programming.

* random	* complex	blas.hpp	filter.hpp	io.hpp	benchmark .hpp	util.hpp
skepu::Random<N> skepu::PRNG Deterministic PRNG	skepu::complex<float> skepu::complex<double> Complex number type, functions, and operator overloads	skepu::blas::axpy skepu::blas::gemv skepu::blas::gemm … Dense level 1-3 BLAS implementation	skepu::filter::blur skepu::filter::edge skepu::filter::downsample … Image filtering routines RGB + grayscale modes	skepu::io::cout skepu::io::cin skepu::io::cerr I/O built on top of skepu::external	Lambda expression-based time measurement Statistical processing Automated backend alternation and measurement	skepu::util::add skepu::util::mul skepu::util::identity … Common user functions

* Language/precompiler integration

SkePU BLAS

- SkePU-accelerated BLAS implementation
  - User functions and multi-backend code generation ”behind the scenes”
- Full coverage of level 1 BLAS
- Coverage of dense parts of level 2 + level 3
- Slightly modernized C++ function signatures
  - Otherwise argument-compatibility with CBLAS

SkePU BLAS — Example

- **Conjugate gradient** computation in SkePU-BLAS

```cpp
#include <skepu>
#include <skepu-lib/blas.hpp>

template<typename T>
void conjugate_gradient(skepu::Matrix<T> BLAS_CONST& A, skepu::Vector<T> BLAS_CONST& b, skepu::Vector<T> &x)
{
	size_t N = b.size();
	assert(A.size_i() == N && A.size_j() == N && x.size() == N);
	skepu::Vector<T> p(N), r(N), Ap(N);

	// Set up initial r and p
	skepu::blas::copy(N, b, 1, r, 1);
	skepu::blas::gemv(skepu::blas::Op::NoTrans, N, N, -1.f, A, N, x, 1, 1.f, r, 1);
	skepu::blas::copy(N, r, 1, p, 1); // p := r

	float rTr = skepu::blas::dot(N, r, 1, r, 1); // rTr = r * r

	for (size_t k = 0; k < N; ++k)
	{
		// Compute alpha
		skepu::blas::gemv(skepu::blas::Op::NoTrans, N, N, 1.f, A, N, p, 1, 0.f, Ap, 1); // Ap := A * p
		float tmp = skepu::blas::dot(N, p, 1, Ap, 1); // tmp := p * Ap = p * A * p
		float alpha = rTr / tmp;

		// Update x
		skepu::blas::axpy(N,  alpha,  p, 1, x, 1); // x := x + alpha * p

		// Update r
		skepu::blas::axpy(N, -alpha, Ap, 1, r, 1); // r := r - alpha * Ap

		// Compute beta
		float rTr_new = skepu::blas::dot(N, r, 1, r, 1); // rTr_new := r * r
		float beta = rTr_new / rTr;

		// Early exit condition
		if (sqrt(rTr_new) < 1e-10f)
			return;

		// Update p
		skepu::blas::scal(N, beta, p, 1); // p := beta * p
		skepu::blas::axpy(N, 1.f, r, 1, p, 1); // p := r + p

		rTr = rTr_new;
	}
}
```

SkePU Deterministic PRNG

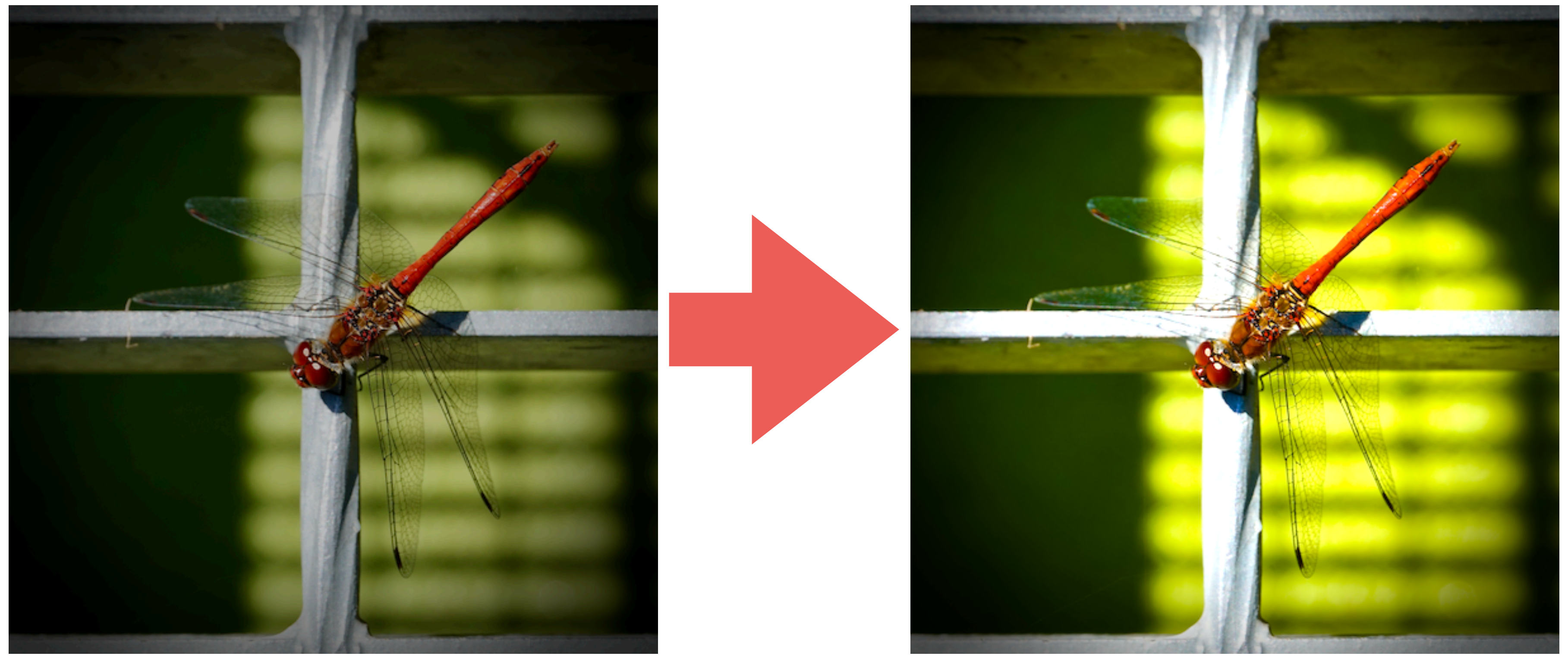
- `skepu::Random<N>` as user function parameter
  - Always first in parameter list
  - Maximum of one such parameter may be present
  - `.get()` extracts integers in [0, MAX)
  - `.getNormalized()` extracts real numbers in [0, 1)
- `skepu::PRNG`
  - Outside user functions
  - Represents individual PRNG streams to bind to skeleton instances

- Monte-Carlo π **calculation**

```
#include <iostream>
#include <skepu>

int monte_carlo_sample(skepu::Random<2> &random)
{
    float x = random.getNormalized();
    float y = random.getNormalized();
    // check if (x,y) is inside region:
    return ((x*x + y*y) < 1) ? 1 : 0;
}

int add(int lhs, int rhs) { return lhs + rhs; }

int main(int argc, char *argv[])
{
    auto montecarlo = skepu::MapReduce<0>(monte_carlo_sample, add);

    skepu::PRNG prng(0);      // seed
    montecarlo.setPRNG(prng);

    const size_t samples = atoi(argv[1]);
    montecarlo.setDefaultSize(samples);

    double pi = (double)montecarlo() / samples * 4;
    std::cout << pi << "\n";
}
```

Image Filtering

- SkePU is well-suited for image processing with its **Map** and **MapOverlap** skeletons and **GPU** backends
- Image processing is also useful for **visualizing** unrelated HPC applications, e.g. heat diffusion
- Standard library includes **image filtering** primitives
  - Data types for **pixels** in various formats
    - Grayscale, RGB, RGBA, HSV, HSVA
  - **Filter kernel** user functions, e.g.
    - Pixel conversions
    - Channel adjustments (brightness, contrast, …)
    - Convolutions (blurs, edge detection, …)



```
#include <skepu>
#include <skepu-lib/filter.hpp>

skepu::Matrix<skepu::filter::RGBPixel> image;
// (Load image from file here)

auto lighten  = skepu::Map<1>(skepu::filter::lighten_rgb);
auto saturate = skepu::Map<1>(skepu::filter::saturate_rgb);

lighten(image, image, 0.5);  // In-place update
saturate(image, image, 0.5); // In-place update
```

Utilities

- SkePU standard library includes frequently seen user functions in the `skepu::util` namespace
  - Arithmetic operations
  - Identity mapping, type conversions
  - Max, min, …

```
#include <skepu>
#include <skepu-lib/util.hpp>

skepu::Vector<float> a(N), b(N);

auto dotproduct = skepu::MapReduce(skepu::util::mul<float>, skepu::util::add<float>);

float res = dotproduct(a, b);
```

SkePU in

Current Research

Lazy Skeleton Evaluation

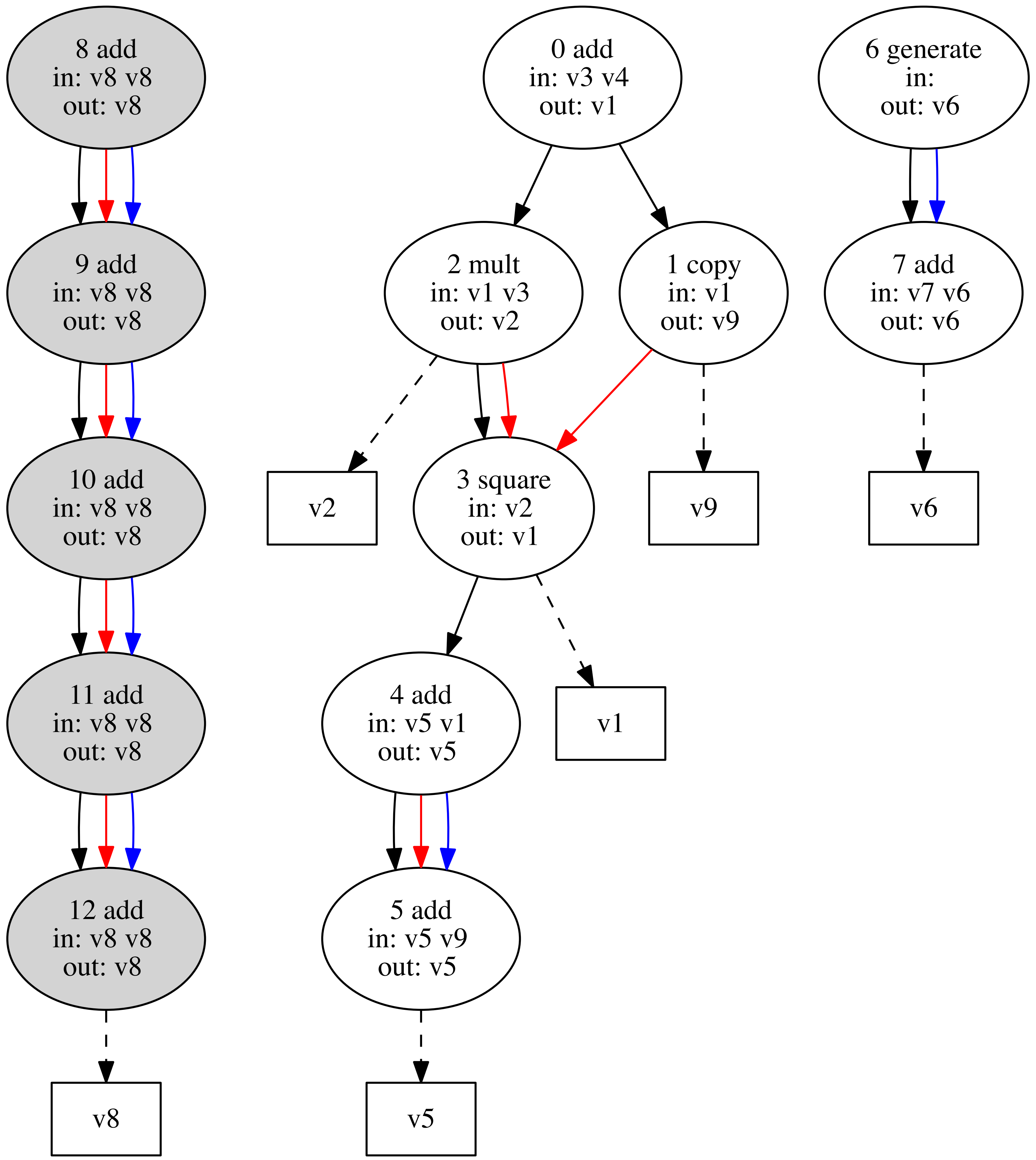
Ernstsson, Kessler, ”Extending smart containers for data locality-aware skeleton programming”, *Concurrency and Computation Practice and Experience,* 2019; 31:e5003 https://doi.org/10.1002/cpe.5003

Lazy Skeletons

- SkePU extended with **lazy evaluation** of the Map skeleton
- No change in syntax, lineage graph built automatically at run-time
- Existing smart containers extended with some of the RDD semantics
- Non-Map operations cause lineage to be evaluated
  - Possibly out of order, while satisfying dependencies
- Run-time dataflow information can assist backend selection

Lazy Skeletons

```
add(v1, v3, v4);

copy(v9, v1);

mult(v2, v1, v3);

square(v1, v2);

add(v5, v5, v1);

add(v5, v5, v9);

add(v6, v7, generate(v6, 5.f));

for (int i = 0; i < 5; i++)
   add(v8, v8, v8);
```



SkePU lineage graph from program

SkePU 2 with Lazy Skeletons

Compile-time

SkePU precompiler
with transformations
Static dataflow information

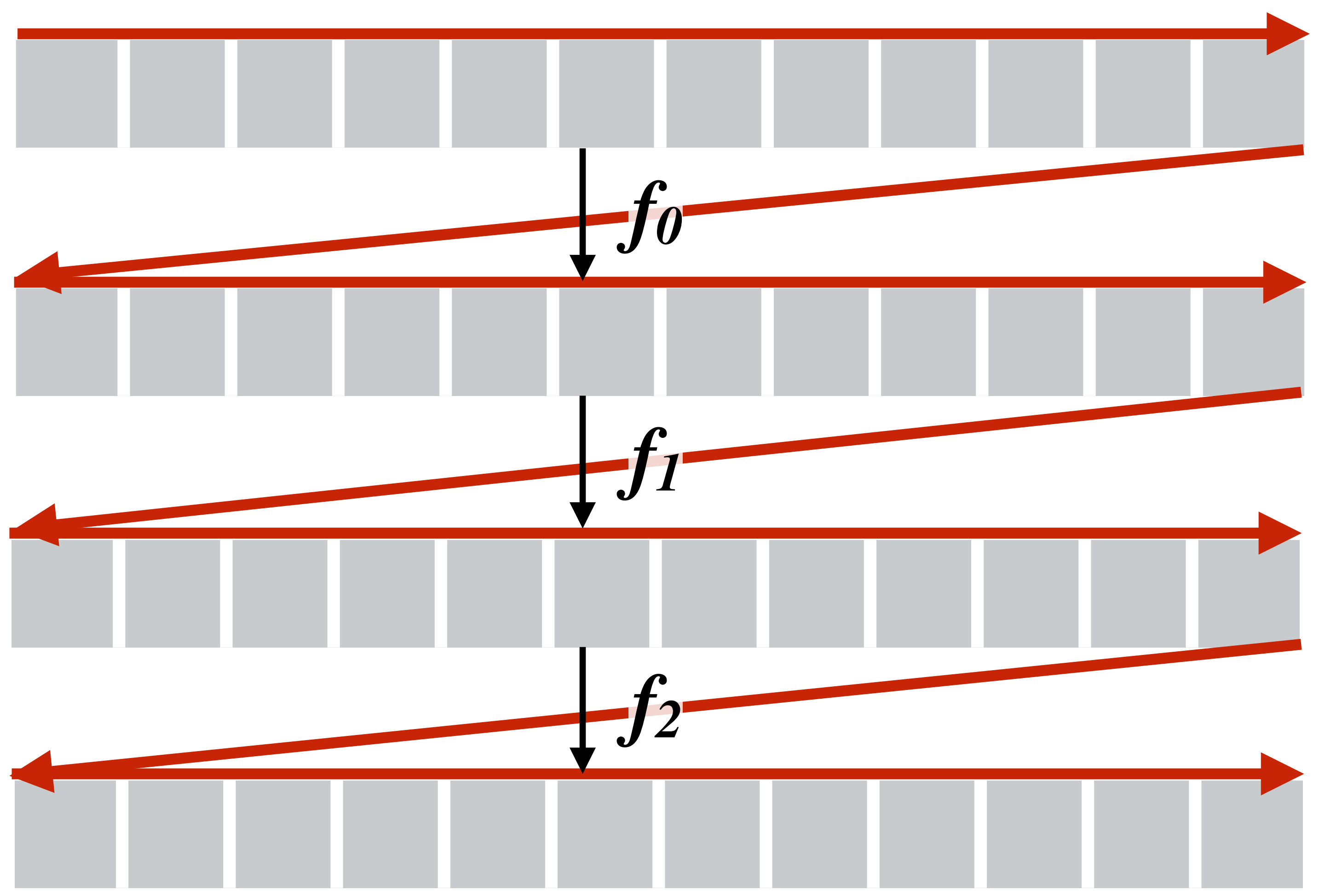
Run-time

Auto-tuning
Smart containers
Dynamic dataflow information

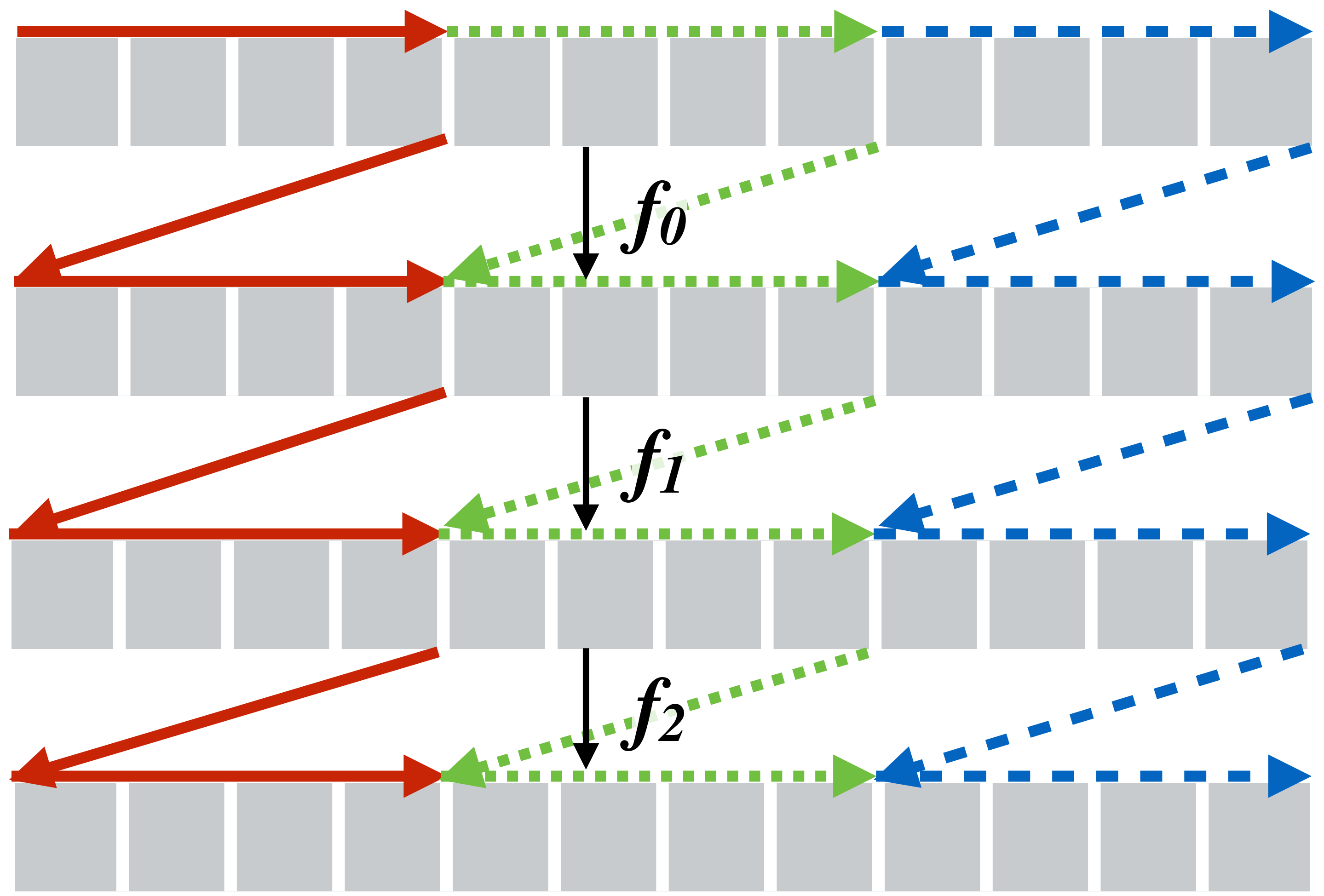
Lazy Skeletons: Tiling

- Lineages evaluated by **tiling** computations
  - Efficient cache utilization on CPU
  - Using accelerators with limited memory space
  - Any situation where cached data is important for performance

Standard SkePU



Lazy SkePU



Multi-Variant User Functions

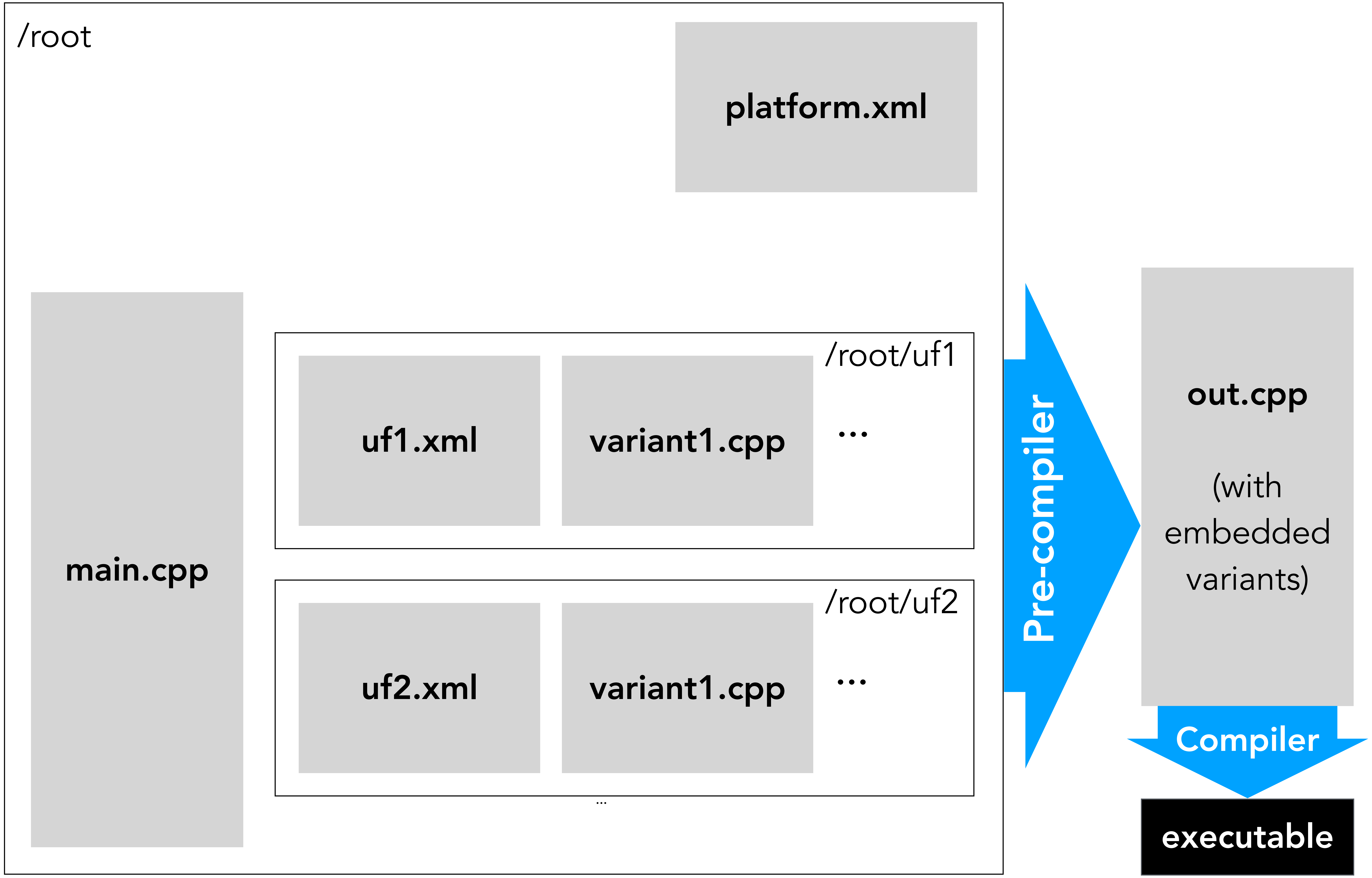
Enhancing Programmability and boosting Performance Portability For Exascale Computing Systems

Ernstsson, Kessler, "Multi-Variant User Functions for Platform-Aware Skeleton Programming"
Parallel Computing: Technology Trends, IOS PRESS , 2020, Vol. 36, pp. 475-484
http://doi.org/10.3233/APC200074

Multi-variant User Functions

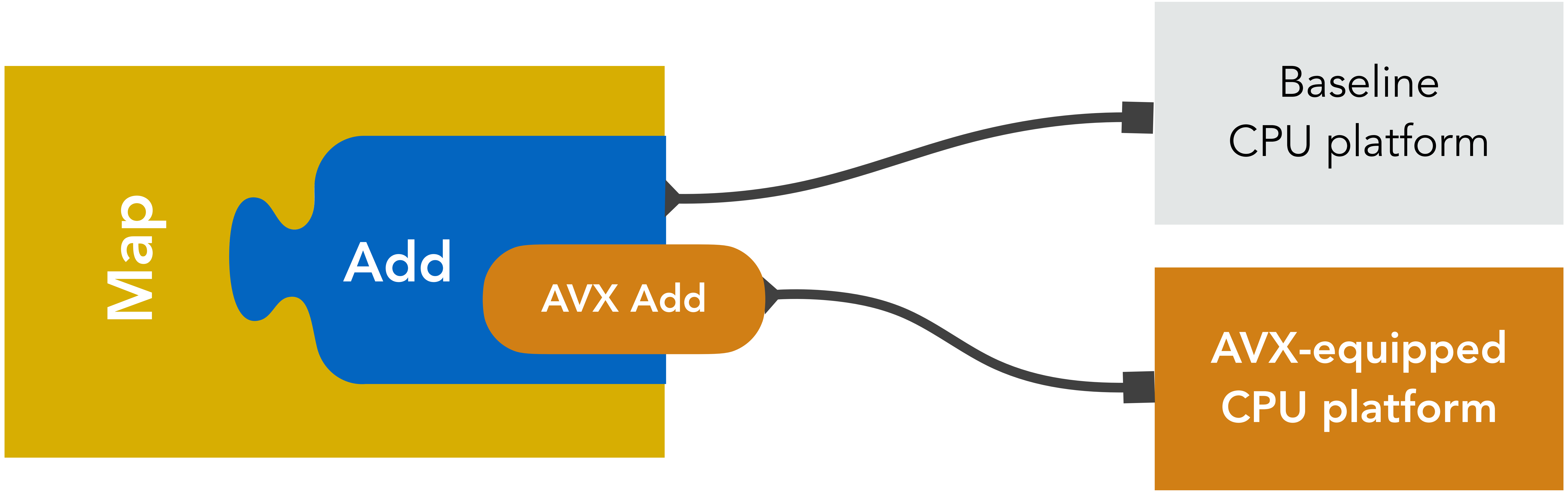
- Single-source, in-line user function definitions are **easy to use**
- **But**: might not always be **performance optimal** approach across backends!
- **Idea**: Leverage multi-variant component approach in Exa2Pro also on skeleton/SkePU level
  - Let each user function have multiple *implementation* variants conforming to the same *interface*
    - Algorithmic variants
    - Hardware/ISA optimizations (vectorize, crypto, …)
    - External library acceleration, …
  - Each variant only selectable when the target system **supports** it
    => platform-aware (XPDL) source-to-source compilation

Tool Flow


/root
platform.xml
main.cpp
/root/uf1
uf1.xml
variant1.cpp
...
/root/uf2
uf2.xml
variant1.cpp
...
...
Pre-compiler
out.cpp
(with embedded variants)
Compiler
executable

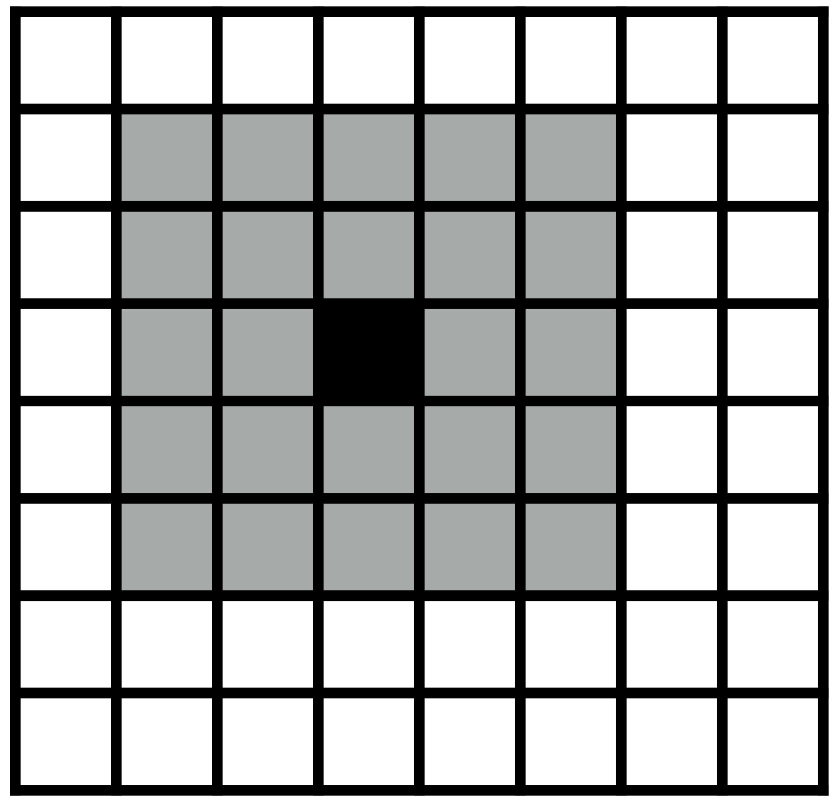
Explicit Vectorization

- Data-parallel algorithms using Map skeleton
- Archetype of skeleton programming: element-wise vector sum
- To increase the complexity, also do element-wise multiplication of complex numbers



Median Filtering

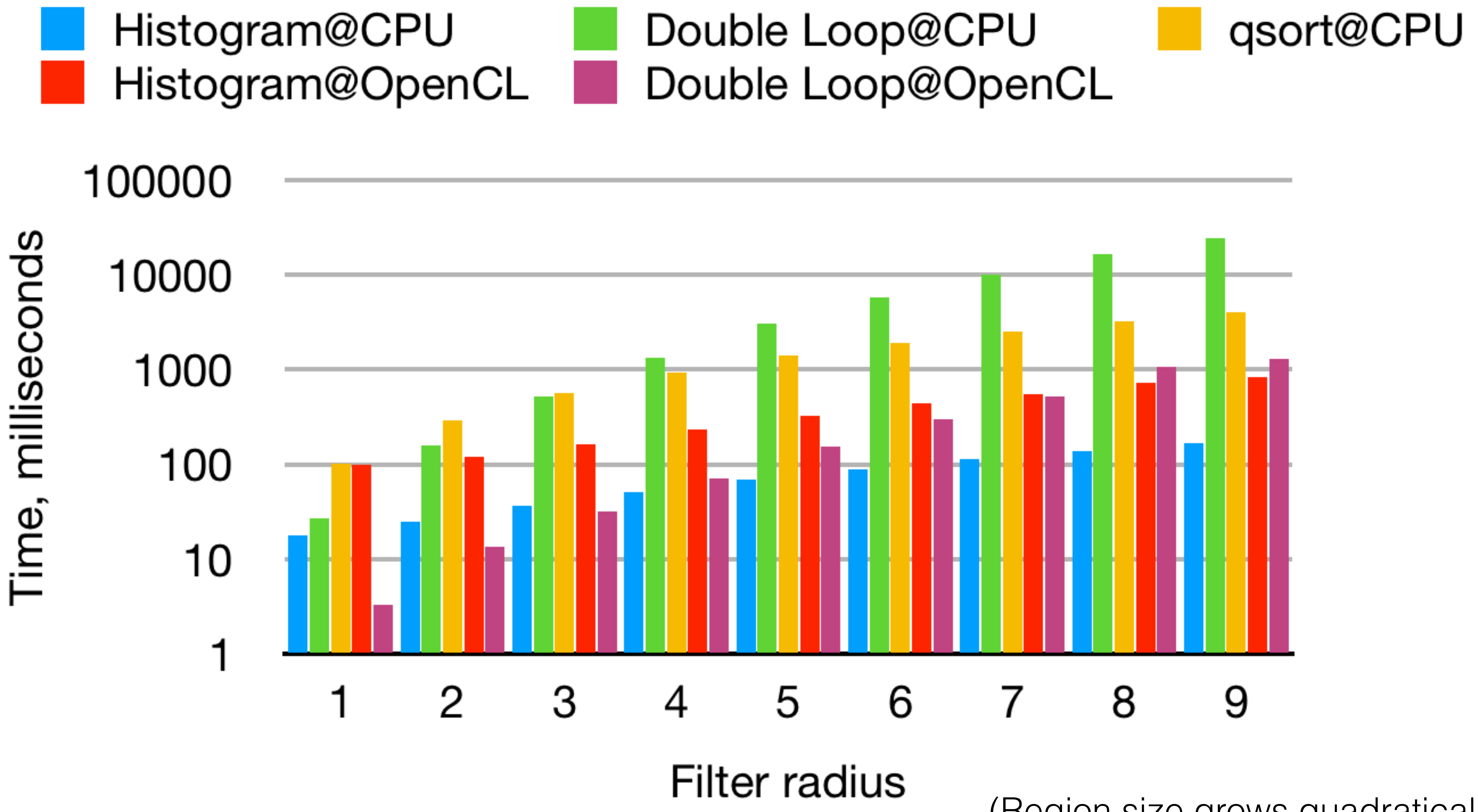
- Median filtering: a blurring **image filter** with MapOverlap
- Data-parallel in the ”outer” dimensions
- **Multiple algorithmic** approaches in the ”inner" dimensions
  - Naive nested-loop comparison
  - Sorting
  - Histogram (note: data independent)



Variant	Time complexity	Memory complexity	Dependencies
Double loop	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	None
Histogram	$\mathcal{O}(n + \lvert D \rvert)$	$\mathcal{O}(\lvert D \rvert)$	None
`qsort`	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	C standard library

Median Filtering Results

Variant	Time complexity	Memory complexity	Dependencies
Double loop	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	None
Histogram	$\mathcal{O}(n + \|D\|)$	$\mathcal{O}(\|D\|)$	None
`qsort`	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	C standard library



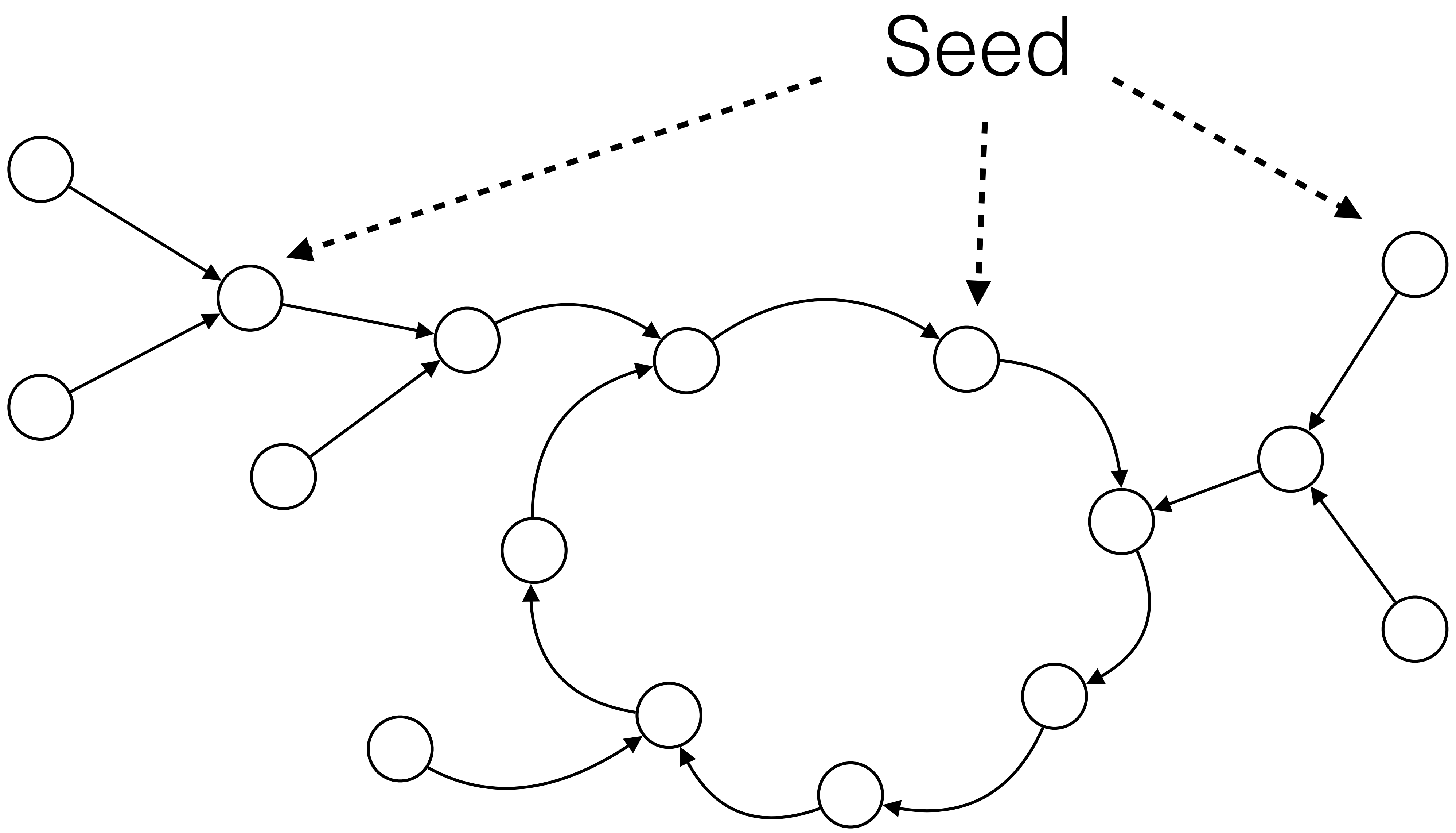
(Region size grows quadratically)

Deterministic Pseudo-Random Number Generation

exa2pro
Enhancing Programmability and boosting Performance Portability For Exascale Computing Systems

Sequential PRNG Basics

- Pseudo-random number generator (PRNG)
- **Finite** state automaton
  - `rand()` produces a pseudo-"random" number from its internal state *S(n)* and transitions to new state *S(n+1)*
  - Inherently **sequential**
  - Naturally **deterministic:** State *S(n)* depends only on state *S(0)* and *n*
  - Input: **seed** (determines initial state)



Parallel PRNG: A Naive Approach

```
void my_parallel_function(…)
{
   int rand_val = rand(); // library function
}
```

Pseudocode

- **Race conditions**
  - Corrupted state
  - Repeated outputs
- Loss of determinism

Bad statistical quality!

Ad-Hoc Solutions to Parallel PRNG

Global synchronization

```
void my_parallel_function(…)
{
   mutex_lock(…);
   int rand_val = rand(); // library function
   mutex_unlock(…);
}
```

Pseudocode

- Flexible

- Sequentialization
- Non-determinism
- Might not be supported (e.g. SkePU)
- Synchronization overhead

Pre-generated buffers

```
void my_parallel_function(int randvals[], …)
{
   int i = my_thread_id();
   int rand_val = randvals[i++];
}
```

Pseudocode

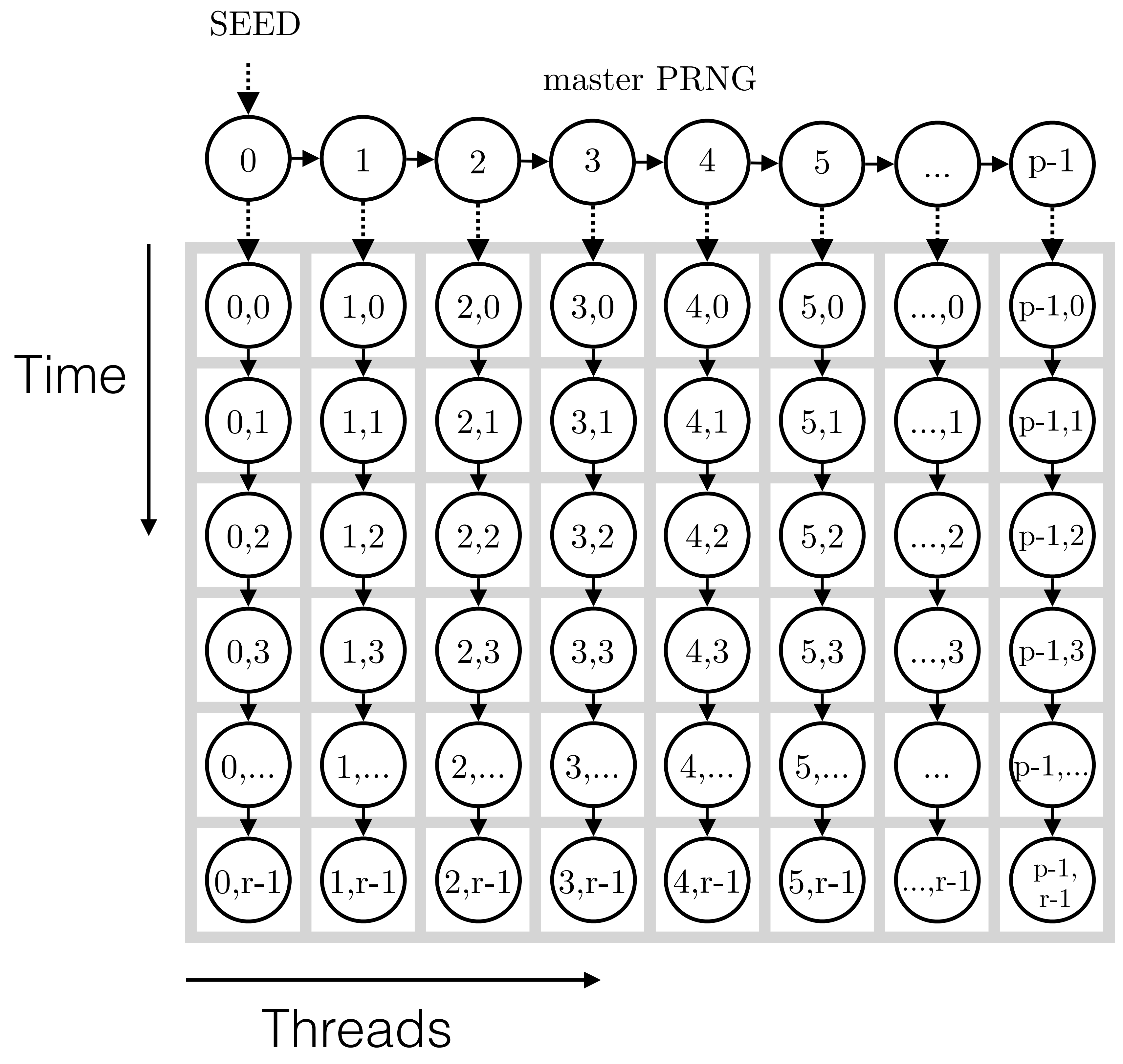
- Simple: works like any data buffer
- Widely supported

- Not really parallel
- Manual pre-allocation of numbers
- Data movement costs

… we can do better!

Stream Splitting

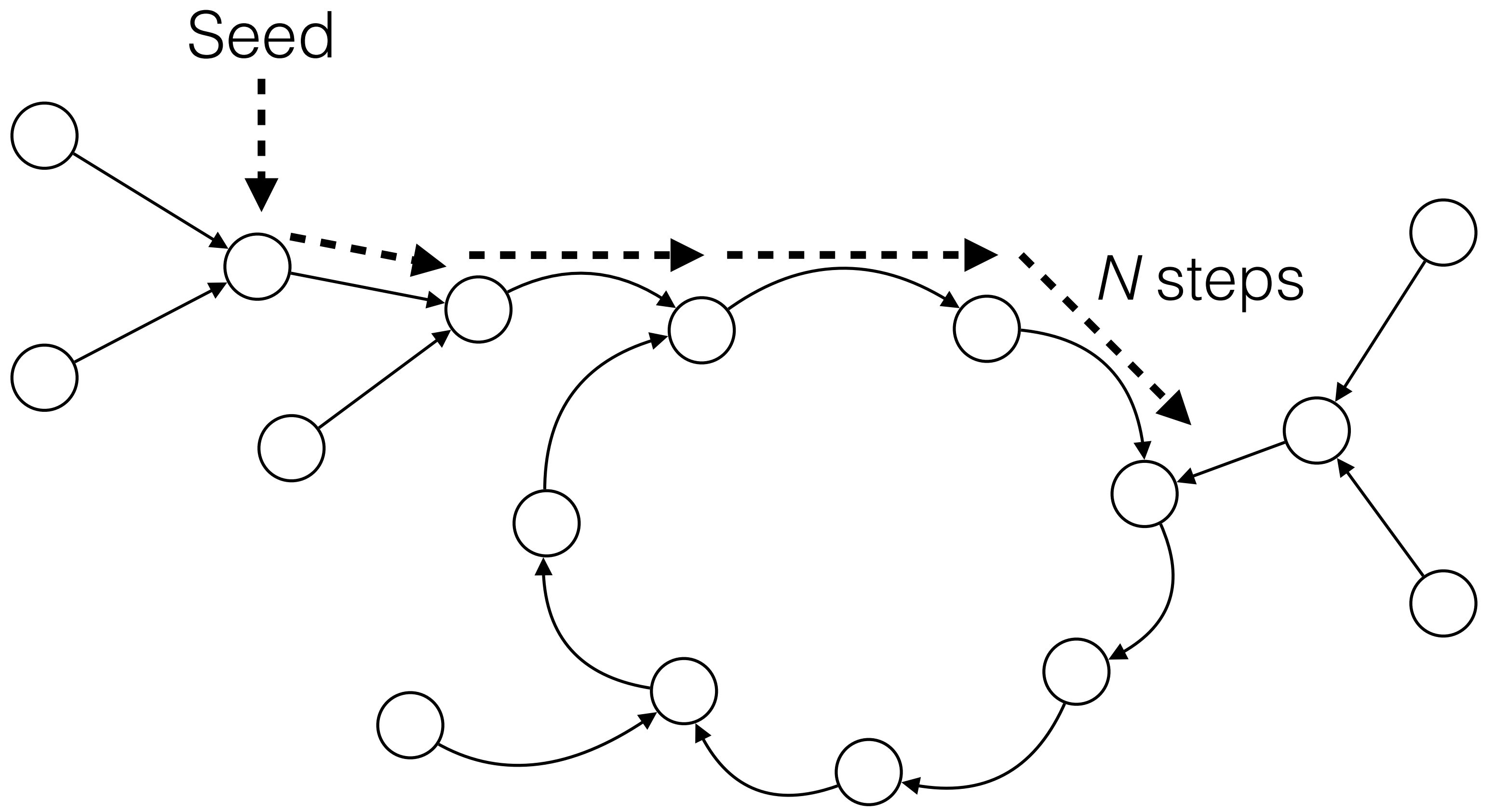
- Basic idea: Give each worker thread a **separate** independent PRNG state
  - No synchronization needed
  - PRNG generation in parallel – no bulk data movement
- Individual PRNGs are **independent**
  - Loss of original single-PRNG structure
  - Need *p* seeds
    - Hierarchical seeding: **master PRNG** producing seeds
- This was roughly the approach for the first SkePUized LQCD prototype

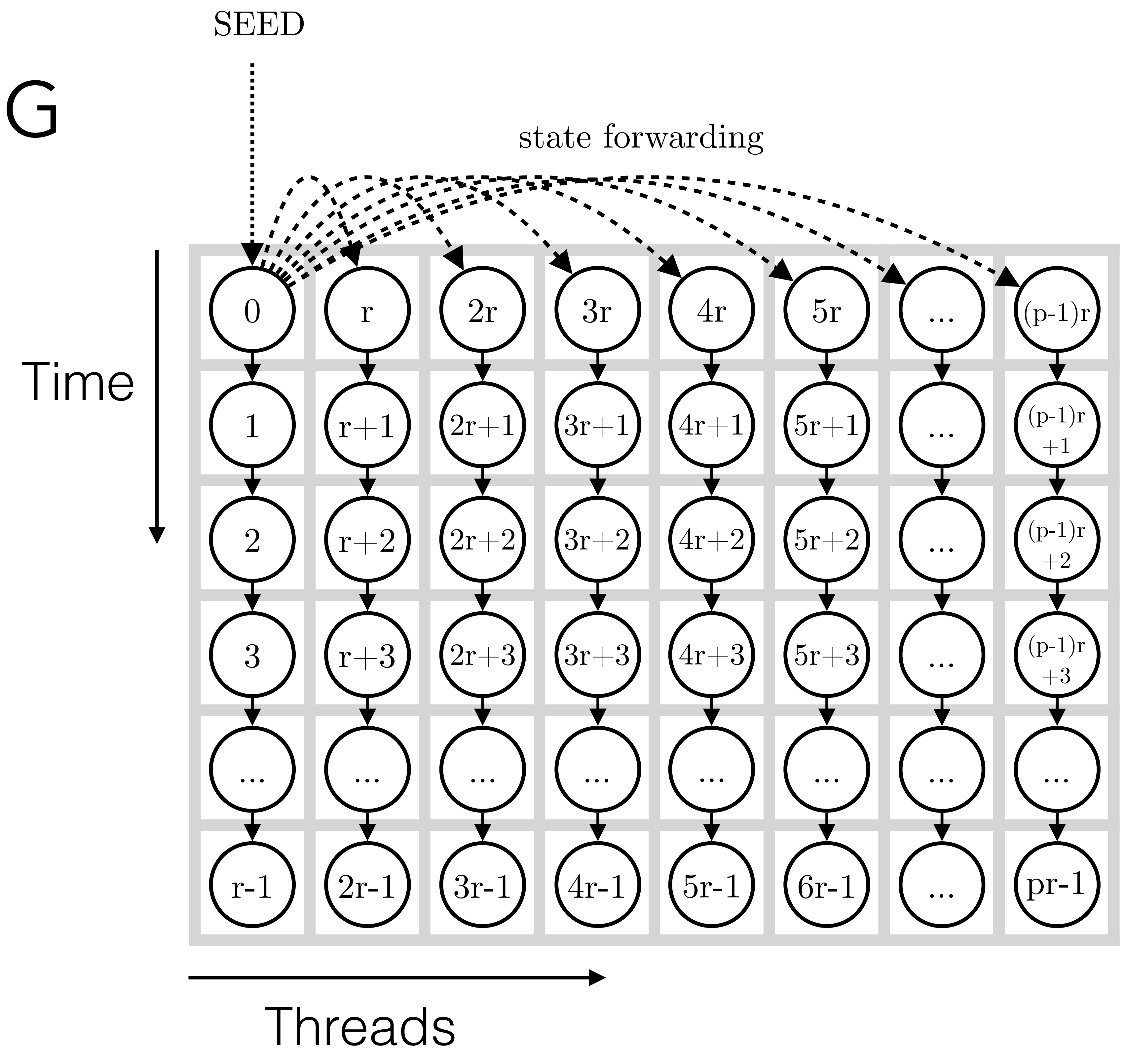


… we can go further!

State Forwarding

- **Goals**:
  - Retain the advantages of ”stream splitting” approach
  - Reclaim structure and determinism from sequential PRNG
- **Recall**: PRNG state automaton
  - We can **advance** the state *N* steps at any point!
  - As long as we know #calls to PRNG in each thread





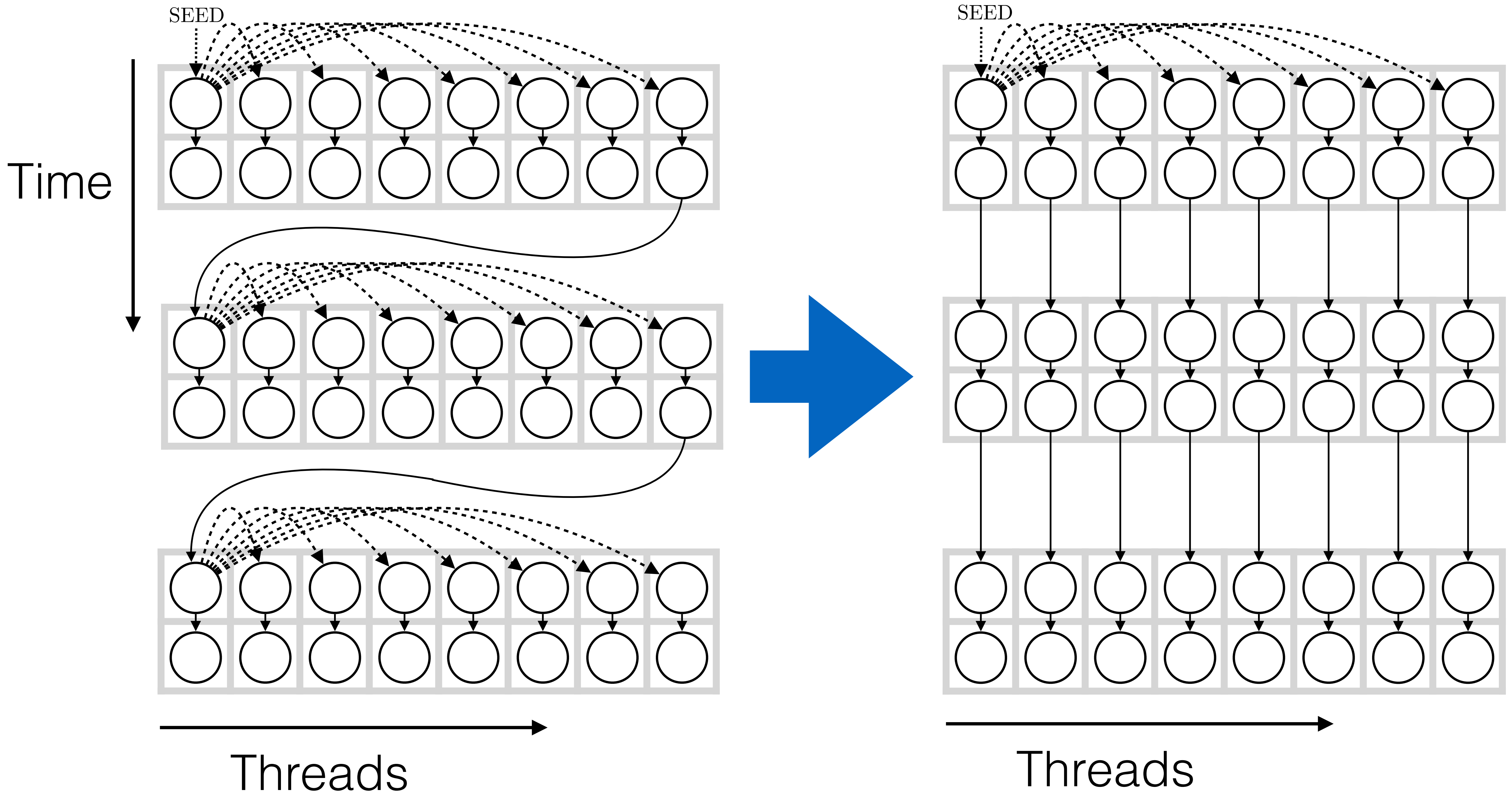
Pre-forwarding for Iterative Workloads

- Forwarding step incurs some overhead
- For **sequences** of skeleton calls, including **iterative** computations, we want to forward as few times as possible

```
for (int i = 0; i < N; ++i)
{
   my_skeleton_instance(…);
}
```

- **Pre-forward approach** for sequences of skeletons when program flow is known ahead of execution-time
  - Static analysis
  - Lineage building
  - Programmer annotation

Ernstsson, Kessler, HLPP 2017
https://doi.org/10.1002/cpe.5003



Behind the Scenes

Behind the Scenes

- SkePU is now **open-source on GitHub**: skepu.github.io
  - Permissive modified BSD-license
  - Systematic automated testing
  - Continuously improving user guide, tutorials, etc.
- SkePU is a key framework in the H2020 EXA2PRO research project
- SkePU is actively used in **teaching**
  (parallel programming courses at LiU)

Enhancing Programmability and boosting Performance Portability For Exascale Computing Systems