# SkePU

https://skepu.github.io

## Introduction & Tutorial

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

LINKÖPING UNIVERSITY

exa2pro
Enhancing Programmability and boosting Performance Portability For Exascale Computing Systems

# Further Reading



- For the tutorial specifically, see the Getting Started Guide
- Currently the best information resource on SkePU is August's licentiate thesis (2020)
  - http://liu.diva-portal.org/smash/record.jsf?pid=diva2:1472256
- See also the user guide on https://skepu.github.io for more concrete instructions on e.g. installation.

# Overview



- Introduction and history
- SkePU interface
- Smart containers
- Skeletons in depth
  - Map
  - Reduce
  - MapReduce
  - Scan
  - MapOverlap
  - MapPairs + MapPairsReduce
- Demonstration, Outlook, Discussion…

# Skeleton Programming

# Skeleton Programming :: Motivation



**Programming parallel systems is hard!**

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

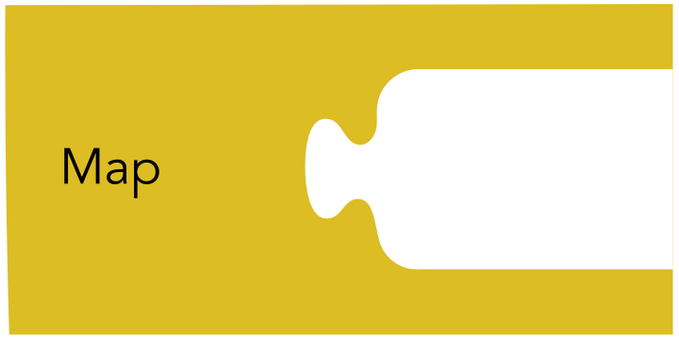
**Skeleton programming (algorithmic skeletons)**

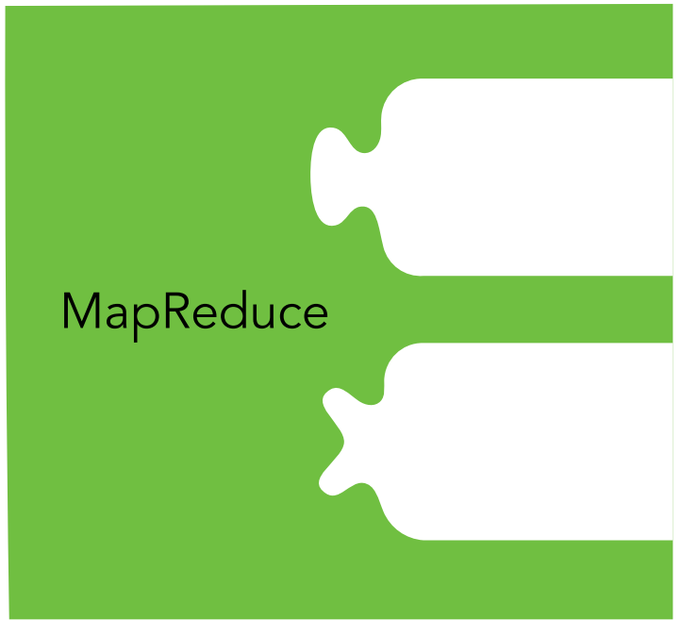
- A high-level parallel programming concept
- Inspired by functional programming
- Generic computational patterns
- Abstracts architecture-specific issues

# Skeletons

Parametrizable higher-order constructs

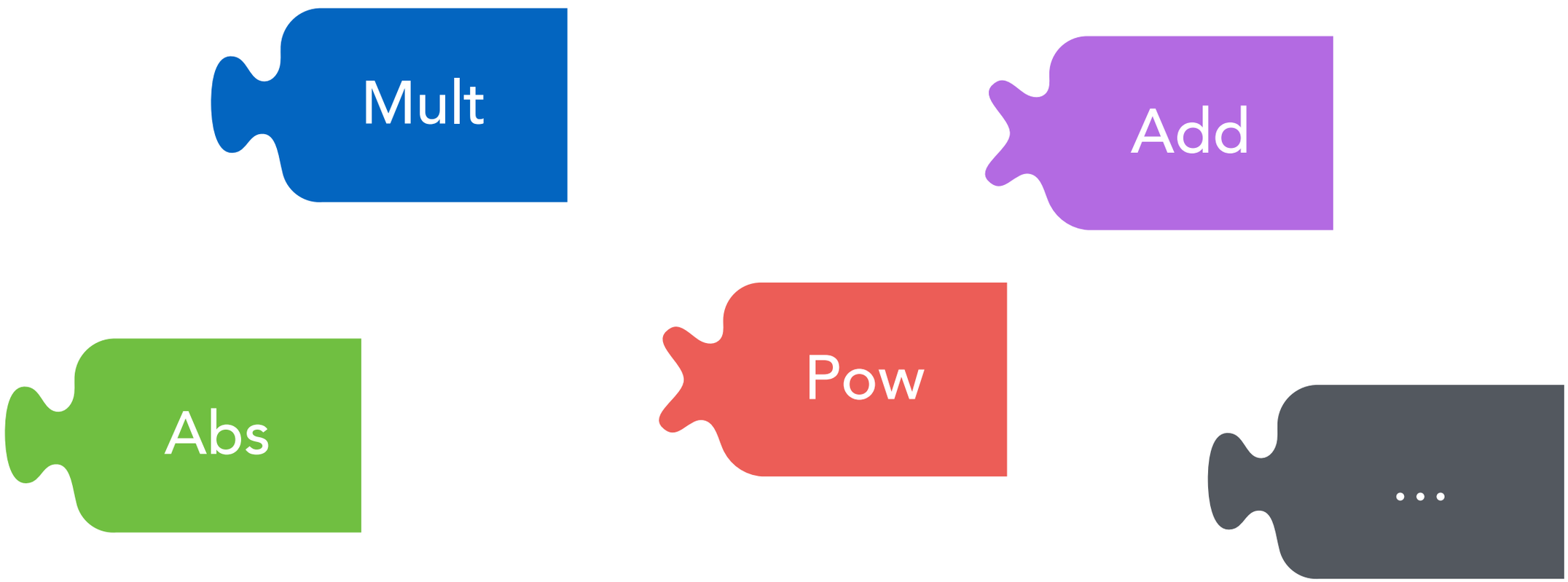
- Map
- Reduce
- MapReduce
- Scan
- and others



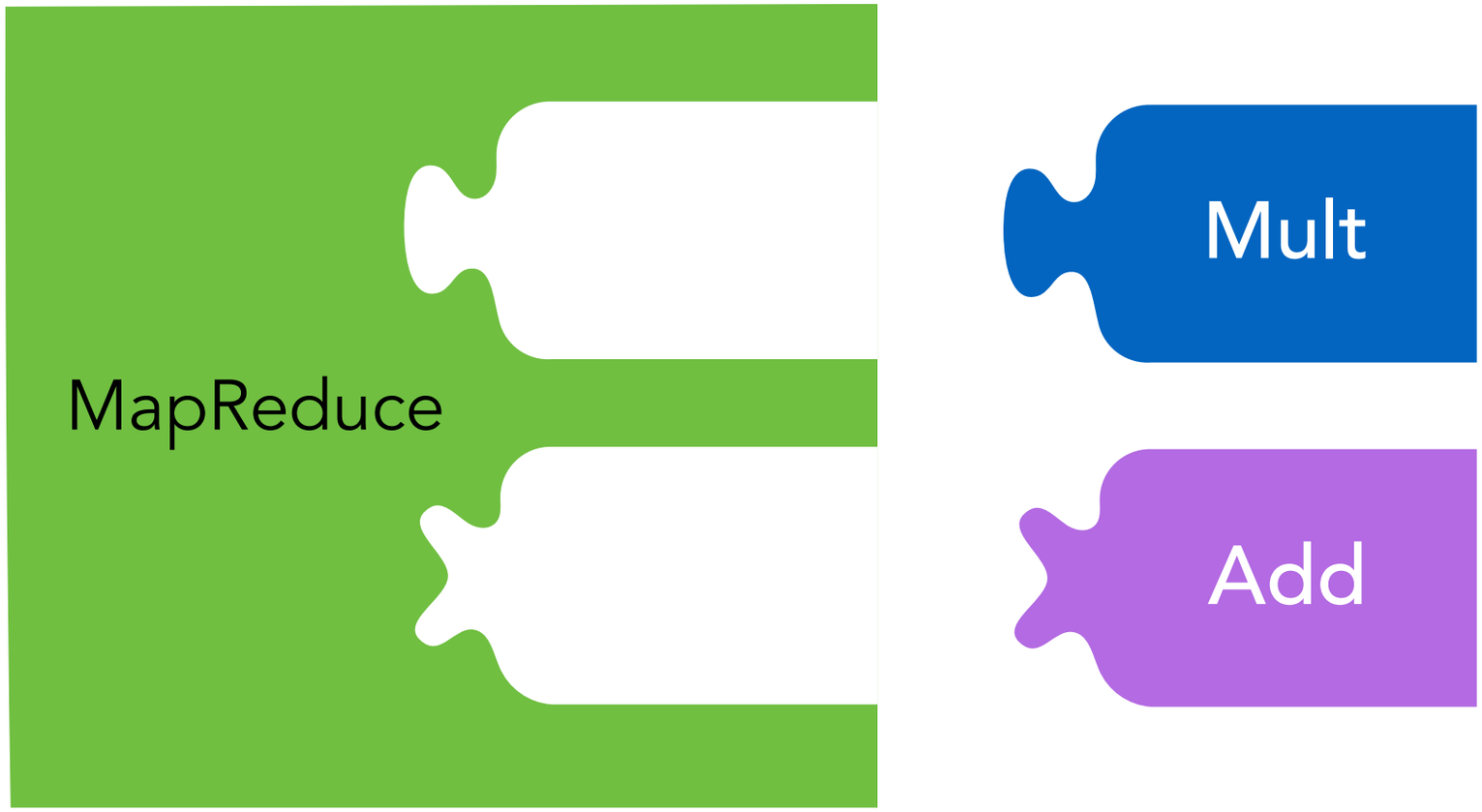
# User functions

User-defined operators



# Skeleton parametrization example

## Dot product operation

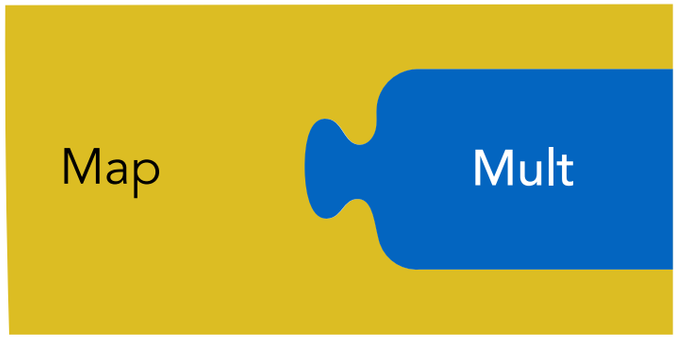


# Skeleton Programming :: Map



## Sequential algorithm

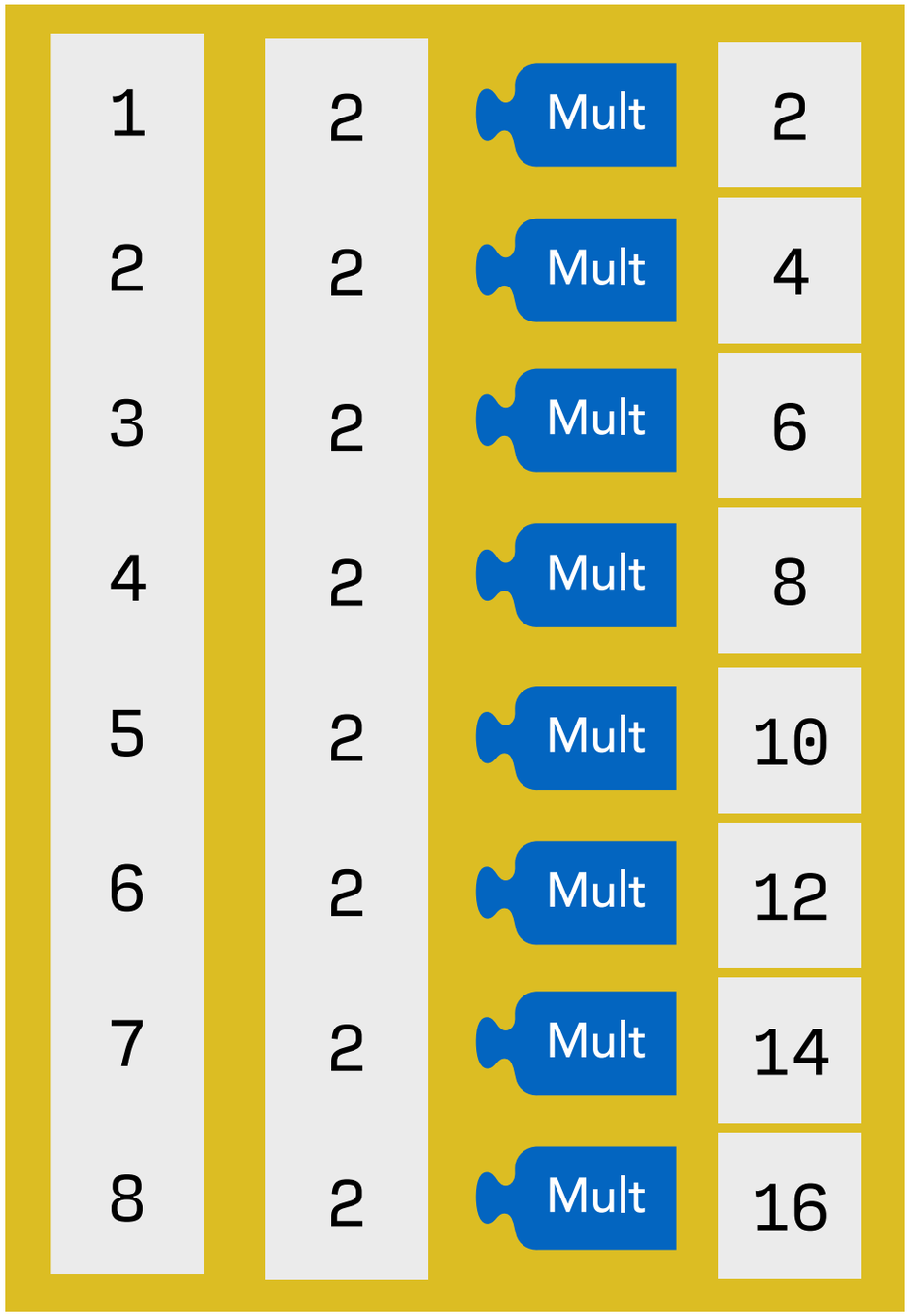
| | | | |
|---|---|---|---|
| 1 | 2 | Mult | 2 |
| 2 | 2 | Mult | 4 |
| 3 | 2 | Mult | 6 |
| 4 | 2 | Mult | 8 |
| 5 | 2 | Mult | 10 |
| 6 | 2 | Mult | 12 |
| 7 | 2 | Mult | 14 |
| 8 | 2 | Mult | 16 |

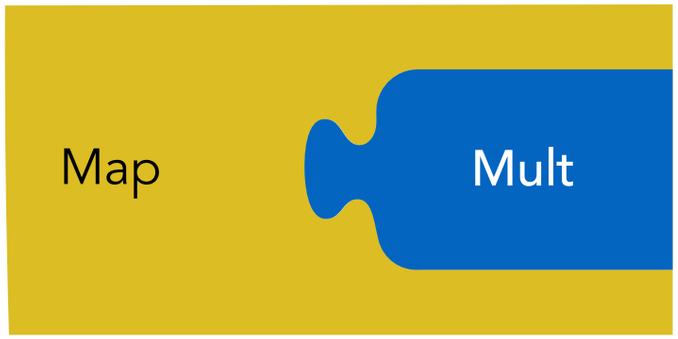


# Skeleton Programming :: Map



## Parallel algorithm

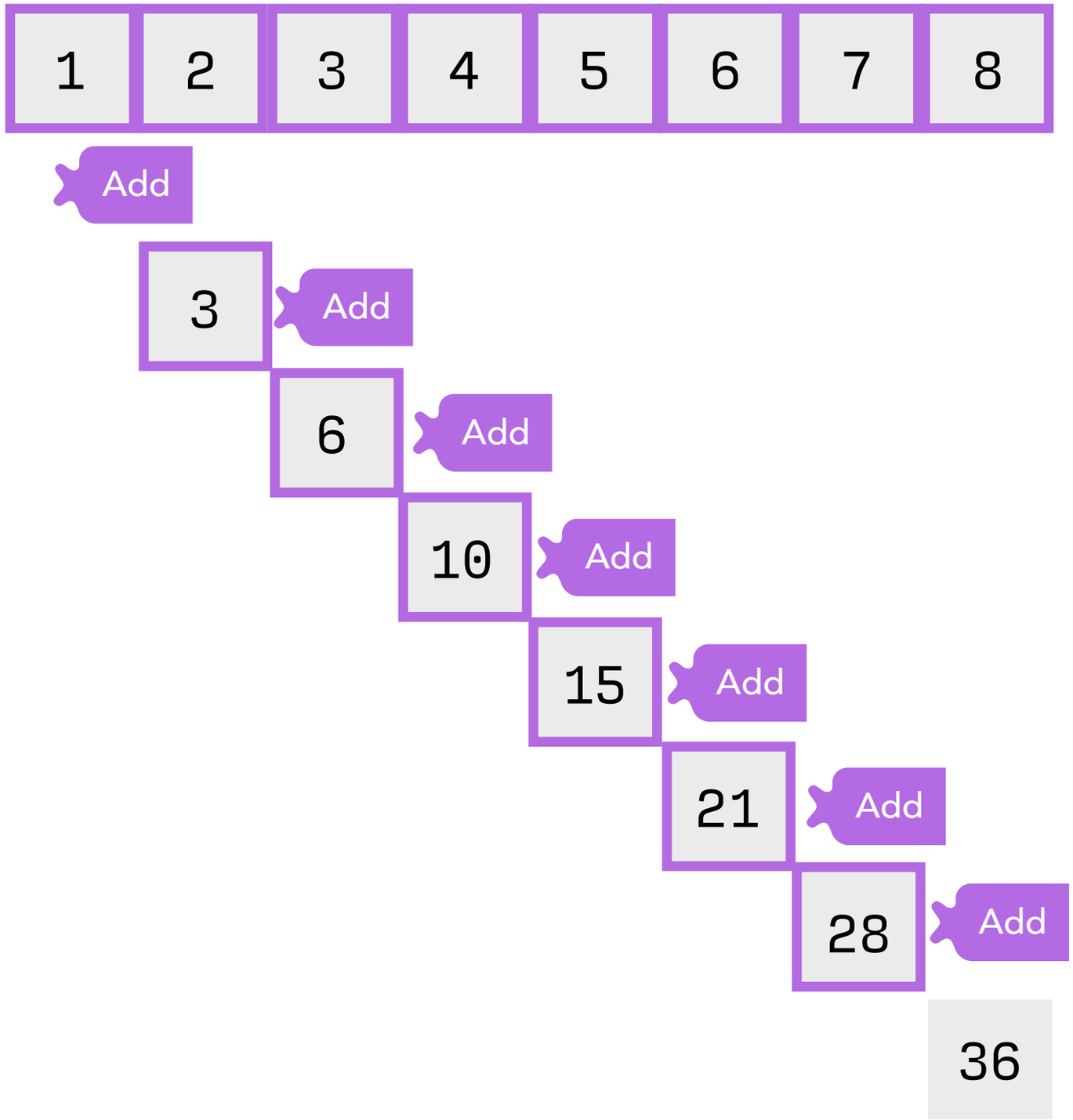


# Skeleton Programming :: Reduce



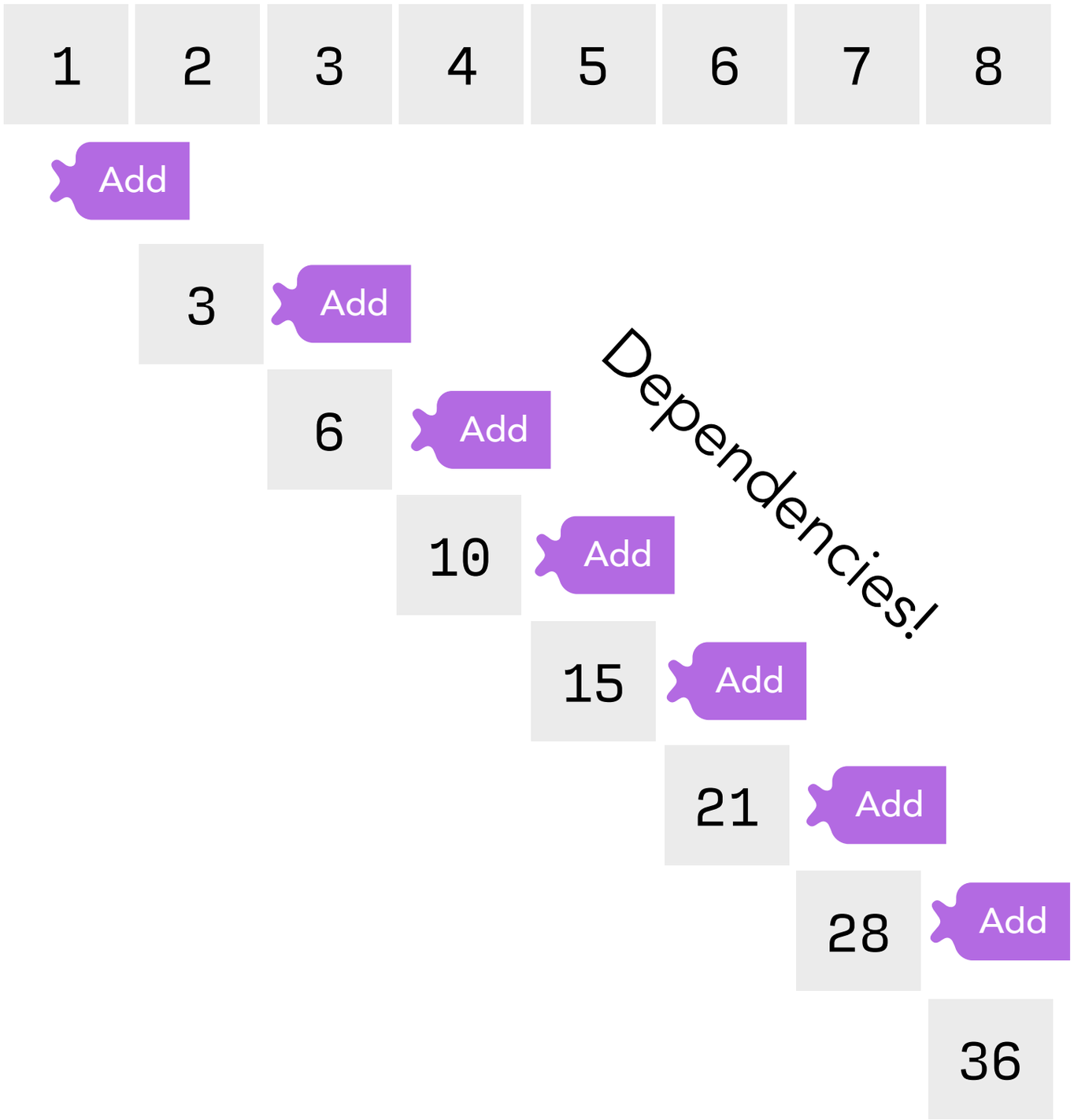
## Sequential algorithm




# Skeleton Programming :: Reduce



## Parallel algorithm?

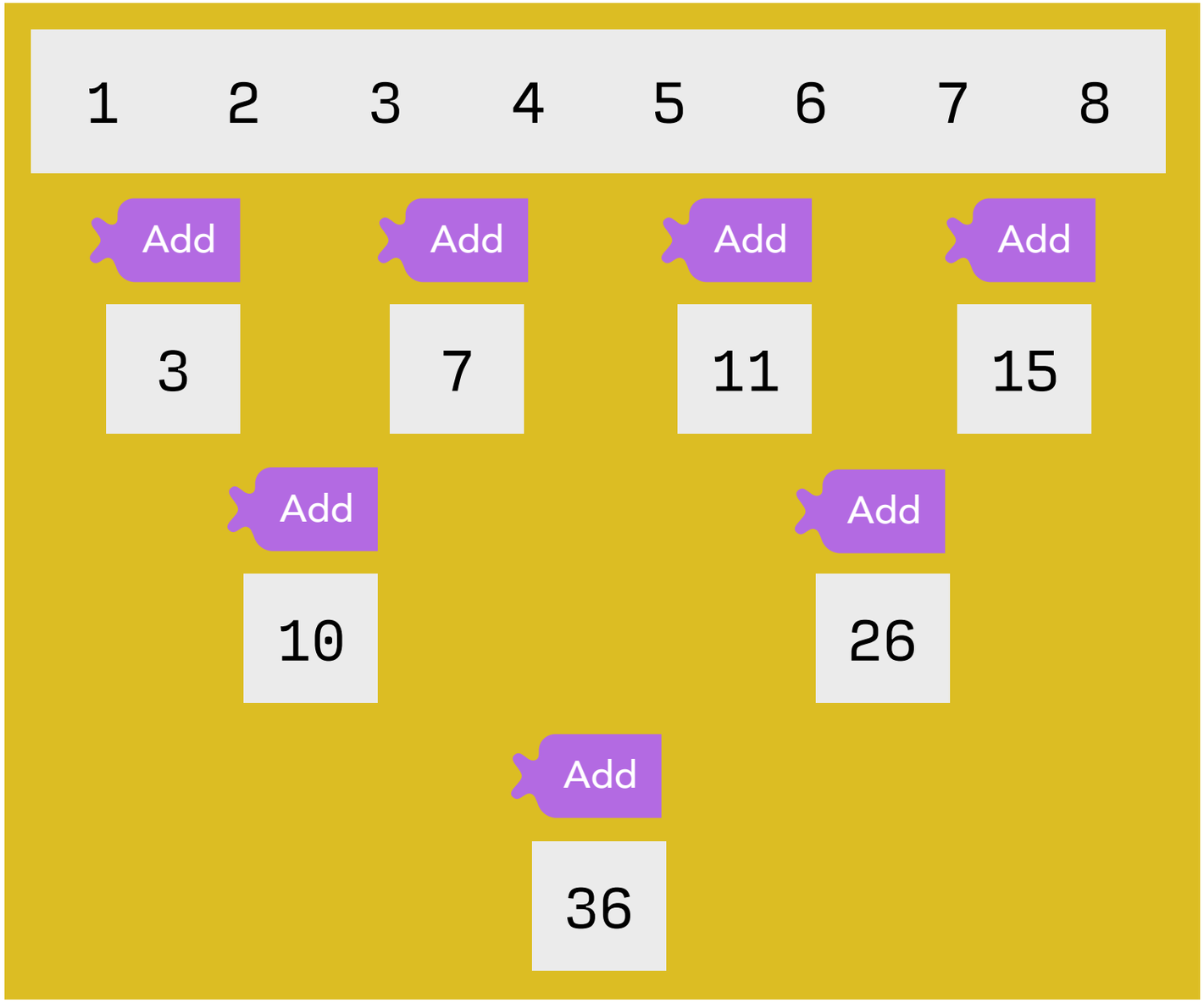



# Skeleton Programming :: Reduce



## Parallel algorithm

(assuming associativity)




# SkePU

# 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

# Features



- ✓ Efficient parallel algorithms
- ✓ Memory management and data movement
- ✓ Automatic backend selection and tuning

# History



- **SkePU 1**, released **2010**
  - C++ template-based interface (limited arity)
  - Multi-backend using macro-based code generation
- **SkePU 2**, released **2016**
  - C++11 *variadic* template interface (flexible arity)
  - Multi-backend using source-to-source precompiler
- **SkePU 3**, preview released **2020**, in development
  - Expanding skeleton set, container set, and expressivity
  - Cluster backend with StarPU-MPI
  - Updated syntax for e.g. MapOverlap
  - Refined consistency model

# 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 2010-2020 (see website)

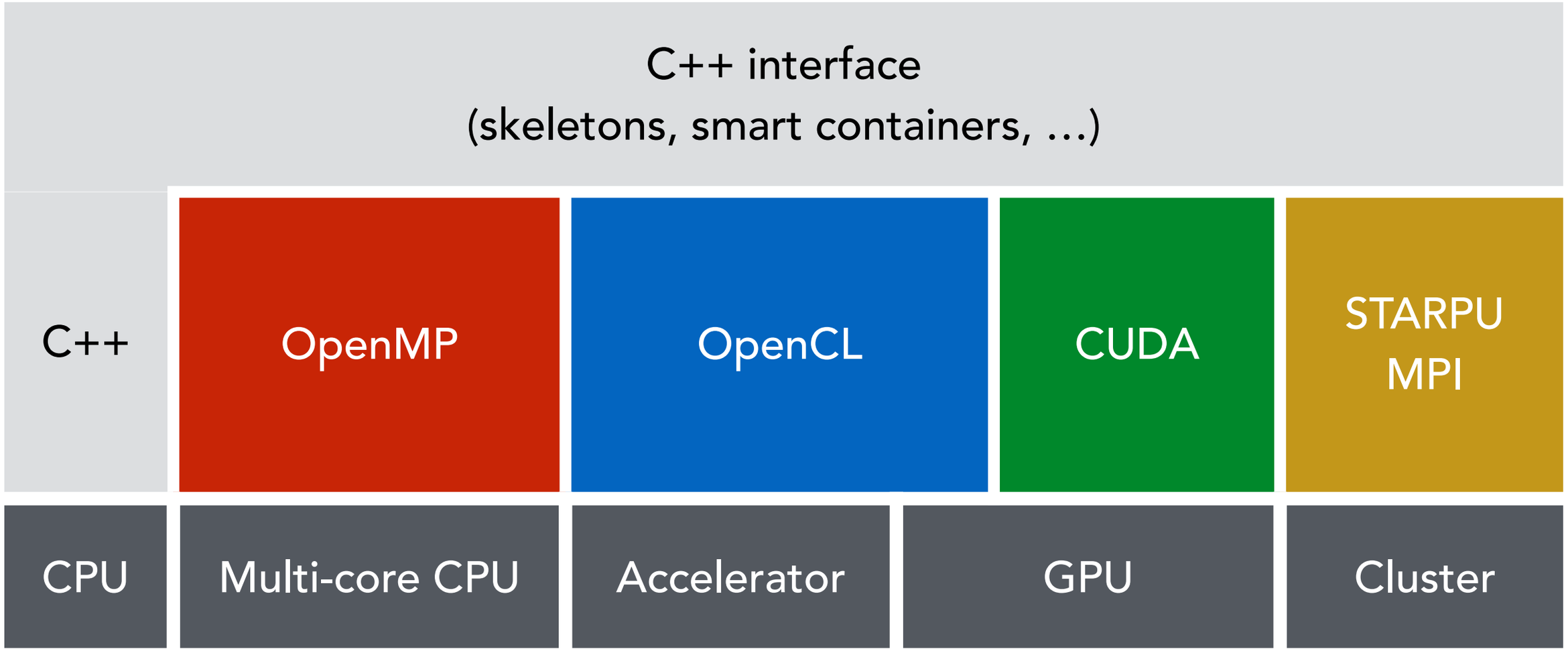
# Skeletons



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

# Backend Architecture





# Smart Containers



## Vector

| i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 |

## Matrix

| i \ j | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 | 4 |
| 1 | 5 | 6 | 7 | 8 | 9 |
| 2 | 10 | 11 | 12 | 13 | 14 |
| 3 | 15 | 16 | 17 | 18 | 19 |
| 4 | 20 | 21 | 22 | 23 | 24 |

## Tensor3

| i | 0 | | | 1 | | |
|---|---|---|---|---|---|---|
| j \ k | 0 | 1 | 2 | 0 | 1 | 2 |
| 0 | 0 | 1 | 2 | 9 | 10 | 11 |
| 1 | 3 | 4 | 5 | 12 | 13 | 14 |
| 2 | 6 | 7 | 8 | 15 | 16 | 17 |

## Tensor4

| j | 0 | | | 1 | | |
|---|---|---|---|---|---|---|
| i | l | 0 | 1 | 2 | 0 | 1 | 2 |
| 0 | 0 | 0 | 1 | 2 | 9 | 10 | 11 |
| | 1 | 3 | 4 | 5 | 12 | 13 | 14 |
| | 2 | 6 | 7 | 8 | 15 | 16 | 17 |
| 1 | 0 | 18 | 19 | 20 | 27 | 28 | 29 |
| | 1 | 21 | 22 | 23 | 30 | 31 | 32 |
| | 2 | 24 | 25 | 26 | 33 | 34 | 35 |

# 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

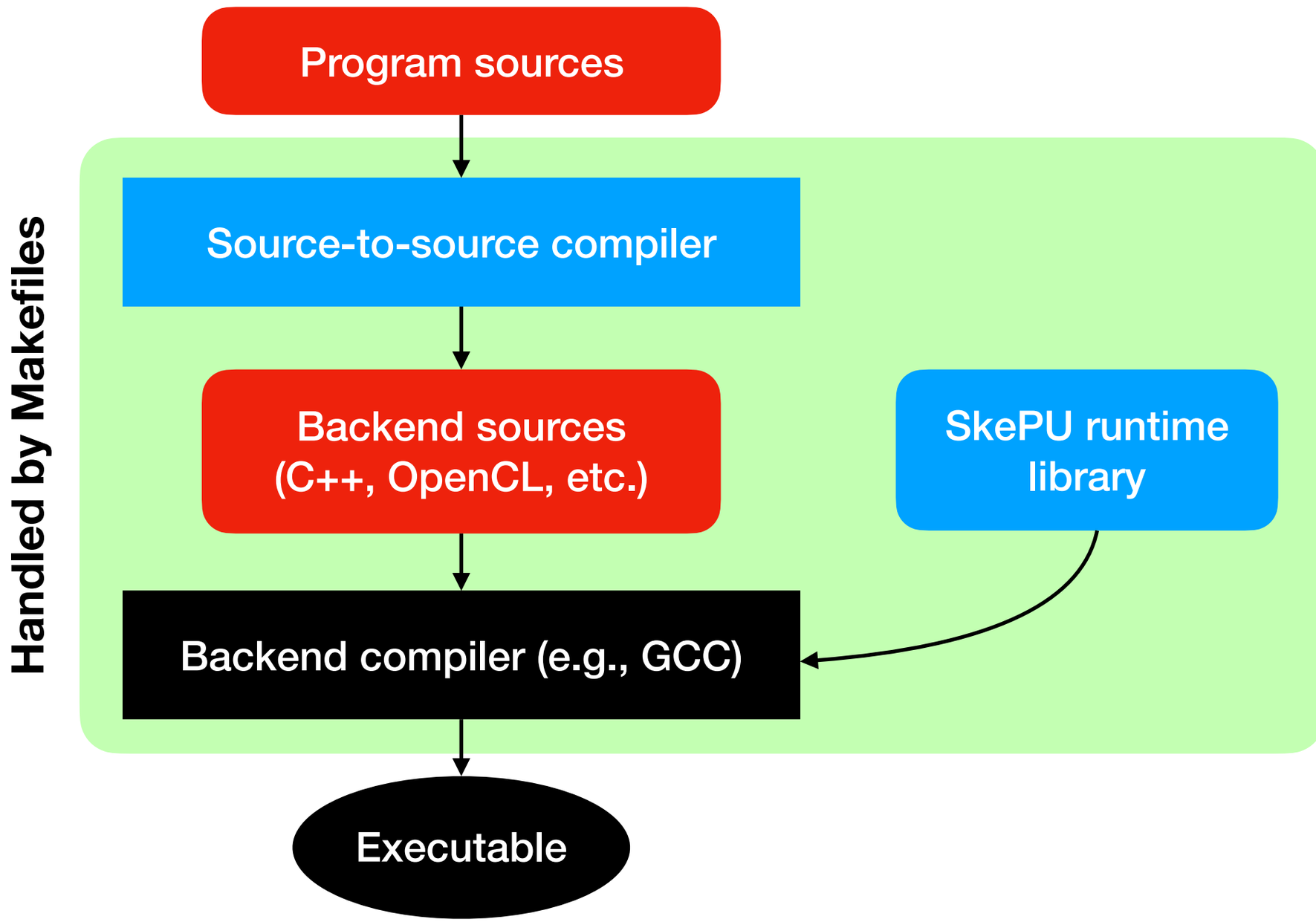
# Using SkePU

# Source-to-Source Translation




# Compilation Architecture





# Installation



- Two main installation options:
  - Docker image, see Getting Started Guide
  - Clone SkePU repository, build with CMake

See website: https://skepu.github.io

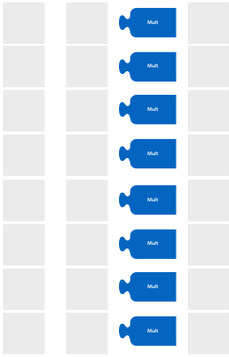
# Syntax



```
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



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

- 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

# 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 mvmult(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 most SkePU backends. SkePU will allow certain such functions to be called from a user function.
  **Examples**: sin(x), pow(x, e)

# Lambda Syntax



```
auto vec_sum = Map<2>([](int a, int b)
{
   return a + b;
});

// ...

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

# Compilation options

# Compilation



- Precompiler options
  - `skepu-tool -name` **`map_precompiled`** **`map.cpp`** `-dir` **`bin -opencl`** `-- [Clang flags]`
  - Handled by Makefiles
-

# Backend Specification



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

# Smart Containers & Consistency Model

# Smart Containers

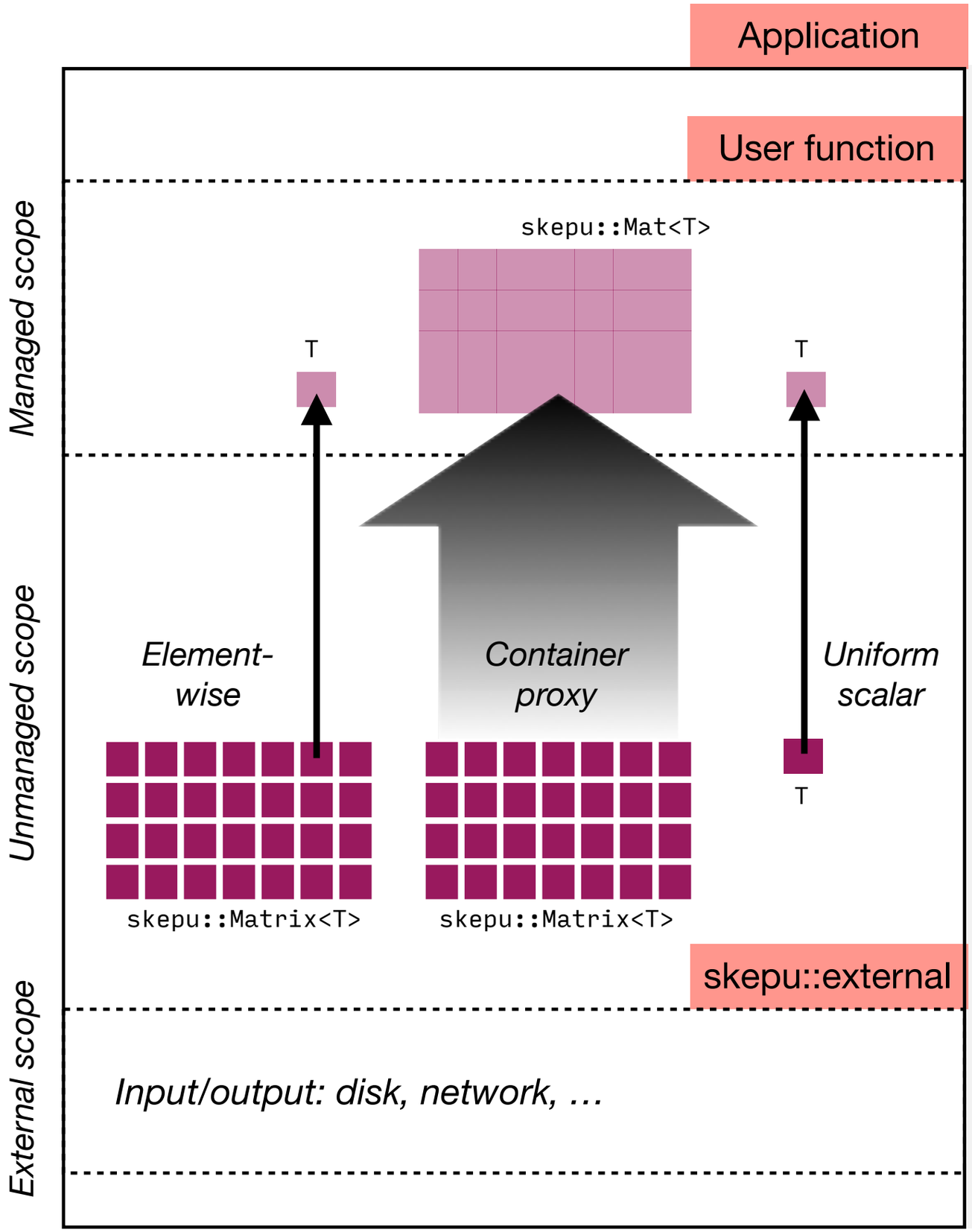


- SkePU provides three different ”scopes” with different syntax and consistency rules
- Inside a user function: **Managed scope**
  - C-style syntax allowed, no global synchronization (compare with GPU kernel code)
- Outside user function: **Unmanaged scope**
  - Regular single-threaded C++-land with 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.

**Changed in SkePU 3**

# External Construct






```
#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);
});
```

# Smart Containers: Unmanaged Scope



- `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)

# Skeletons

# Skeleton Features

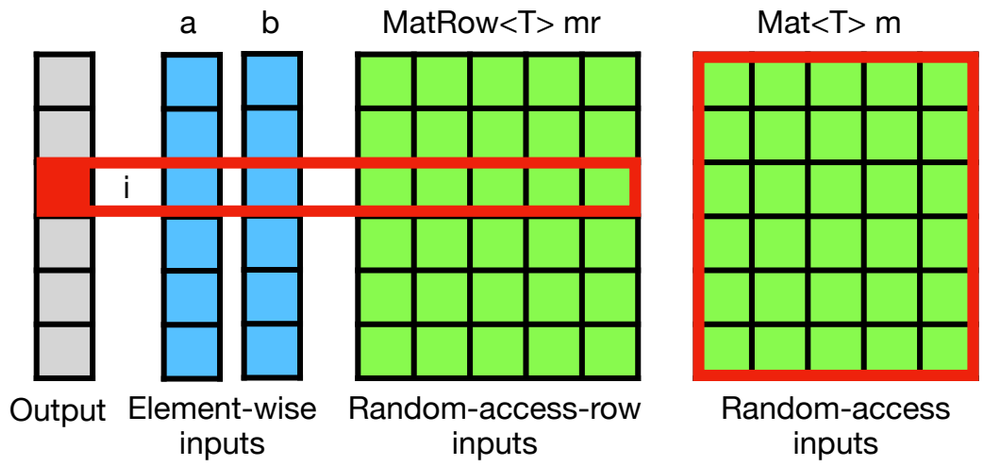


| Feature \ Skeleton | Map | MapPairs | MapOverlap | Reduce | Scan | MapReduce | MapPairsReduce | Call |
|---|---|---|---|---|---|---|---|---|
| **Elwise dimension in** | 1–4 | 1 | 1–4 | 1–4** | 1 | 1–2 | 1 | 0 |
| **Elwise dimension out** | Same as *in* | 2 | Same as *in* | 0–1 | 1 | 0 | 1 | 0 |
| **Indexed** | 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 | Variadic |
| **Uniform parameters** | Variadic | 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 – Iterators



- Optionally, use iterators with Map (and most other skeletons)

- `mapper(r.begin(), r.end(), v1.begin(), v2.begin());`

# Map Example



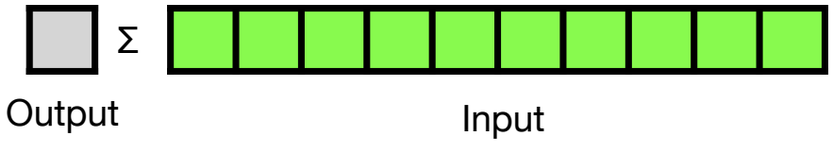
```
float sum(float a, float b)
{
   return a + b;
}

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);
}
```

# Reduce

# Reduce Modes



- **1D Reduce**



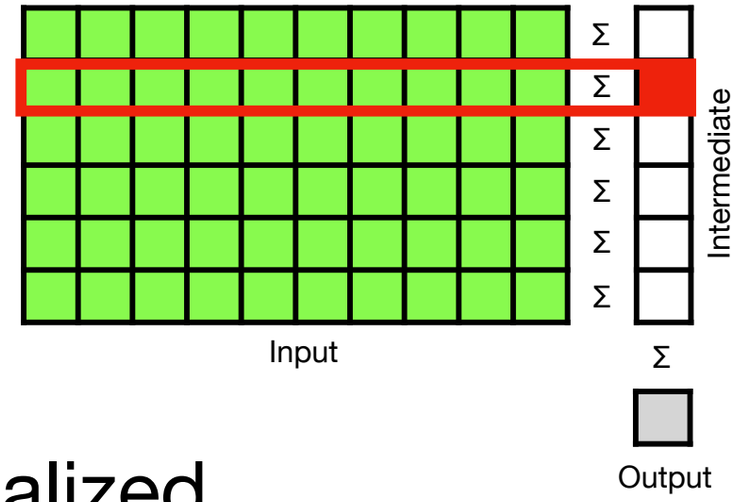

  - Regular Vector
  - Matrix RowWise (returns Vector)
  - Matrix ColWise (returns Vector)

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

- **”2D” Reduce**

```
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);
```

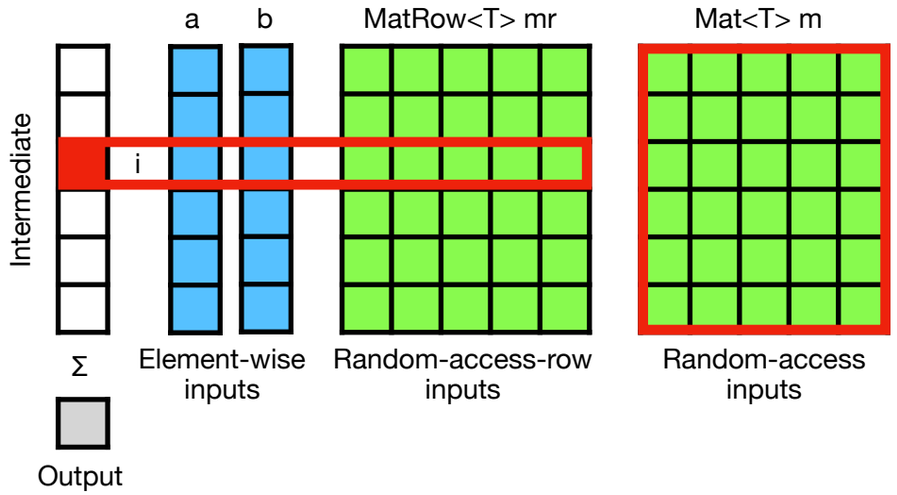
# MapReduce

# 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

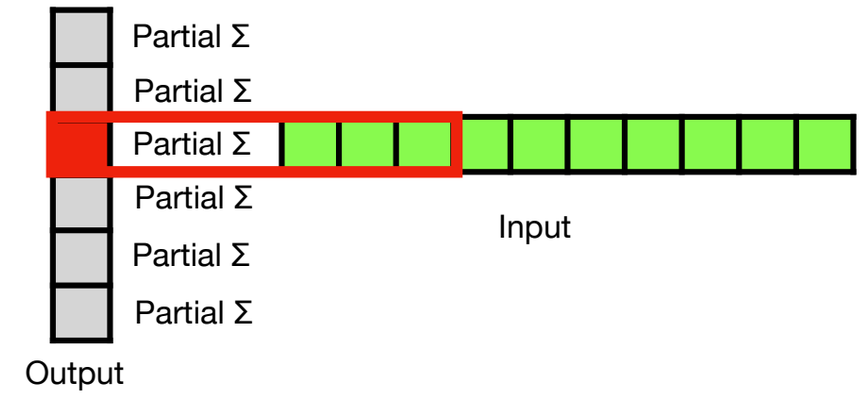


```
float add(float a, float b)
{
  return a + b;
}

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



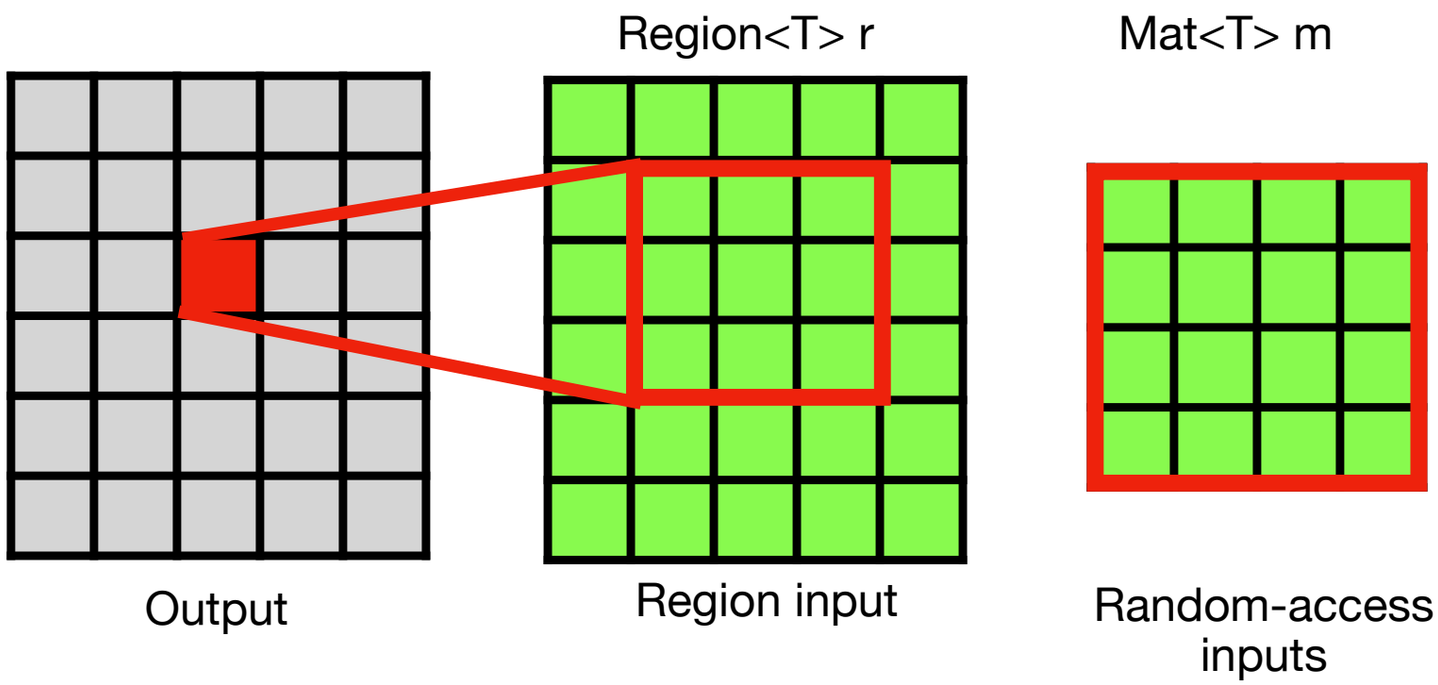
```
float max_f(float a, float b)
{
  return (a > b) ? a : b;
}

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

# MapOverlap

# MapOverlap Modes



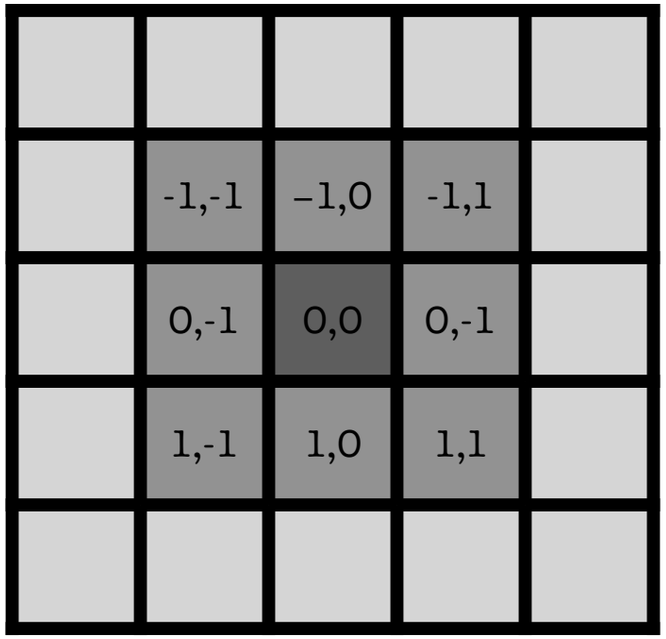
- Region of the input container accessible in user function
- In addition to optional full random-access parameters



# MapOverlap Modes



- Region indexing is 0-centered in each dimension

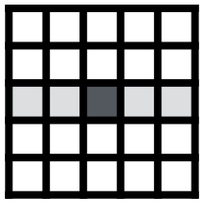


# MapOverlap Modes

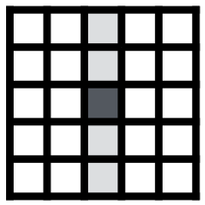


- **1D MapOverlap** `With 1D user function`
  - Regular Vector

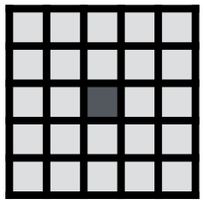
  - Matrix RowWise `instance.setOverlapMode(Overlap::RowWise) // default`



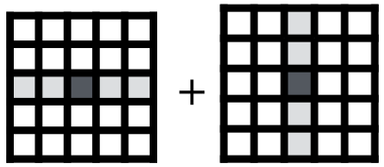
  - Matrix ColWise `instance.setOverlapMode(Overlap::ColWise)`



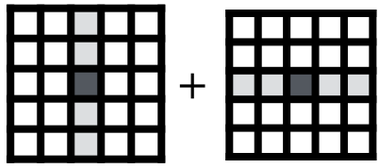
- **2D MapOverlap**
  - Regular Matrix



- **Separable MapOverlap (2D-with-1D)**
  - Matrix RolColWise `instance.setOverlapMode(Overlap::RowColWise)`




  - Matrix ColRowWise `instance.setOverlapMode(Overlap::ColRowWise)`



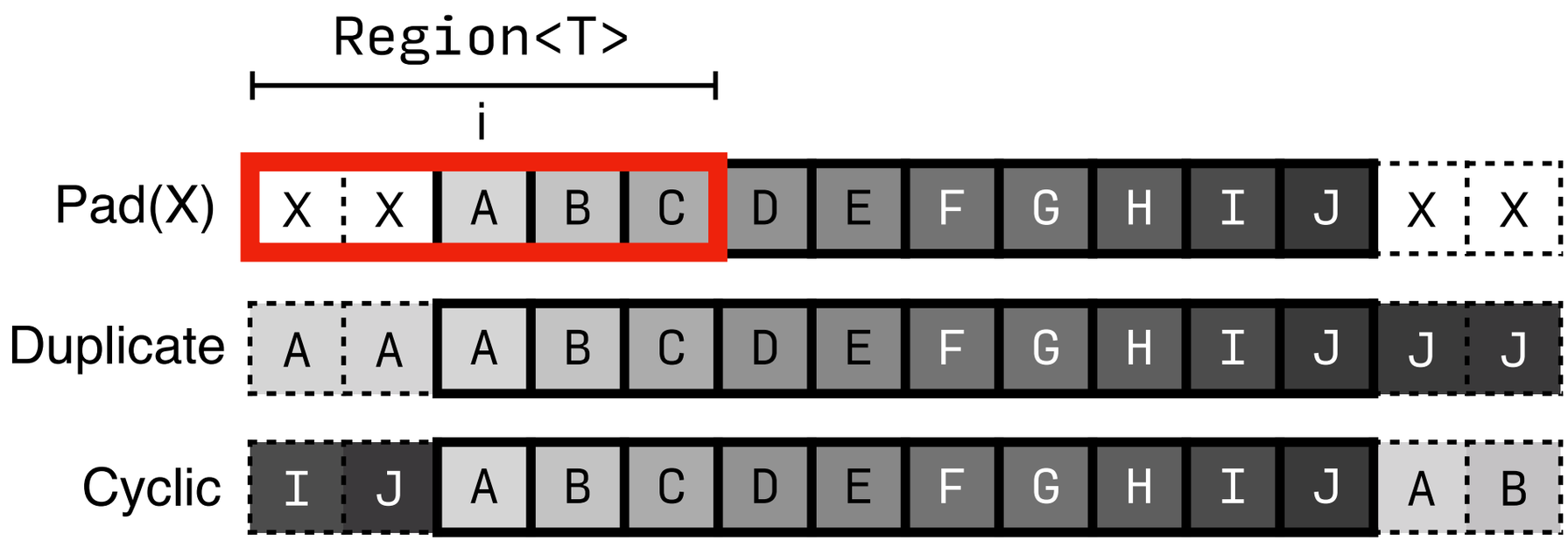
# MapOverlap Edge Mode



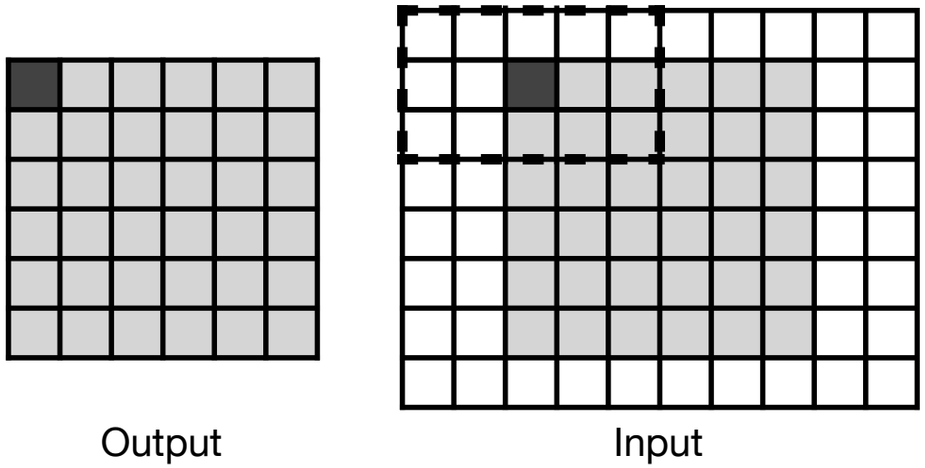
- `instance.setOverlap(x [, y, …])`
  - Set overlap radius
- `instance.setEdgeMode(mode)`
  - Edge::Pad
    - `instance.setPad(pad)` – set value
  - Edge::Duplicate (default for 1D)
  - Edge::Cyclic
  - Edge::None (default for other dimensions)

# MapOverlap Edge Mode






None



# 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> stencil {1, 2, 4, 2, 1};
   Vector<float> result(v.size());
   convol.setOverlap(2);
   return convol(result, v, stencil, 10);
}
```

# MapOverlap 2D



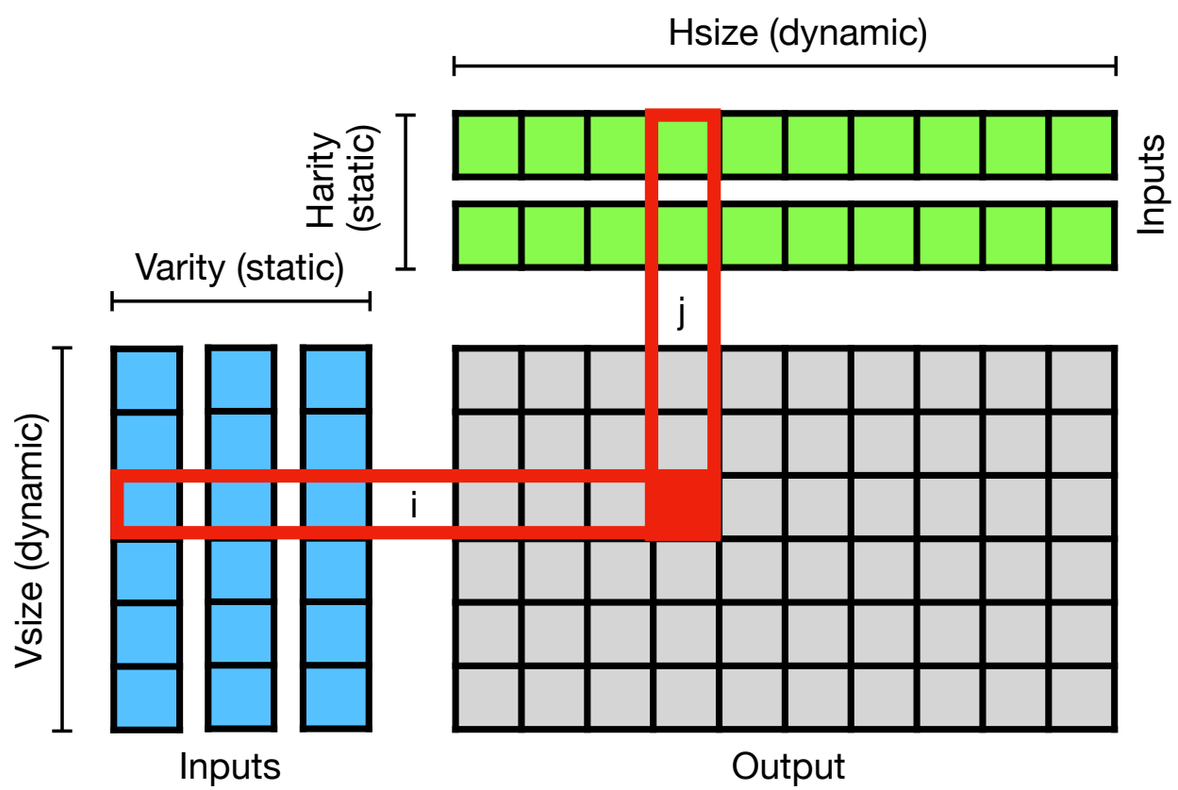
```
float over_2d(skepu::Region2D<float> r, const skepu::Mat<float> stencil)
{
   float res = 0;
   for (int i = -r.oi; i <= r.oi; ++i)
      for (int j = -r.oj; j <= r.oj; ++j)
         res += r(i, j) * stencil(i + r.oi, j + r.oj);
   return res;
}
```

# MapPairs

# MapPairs



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

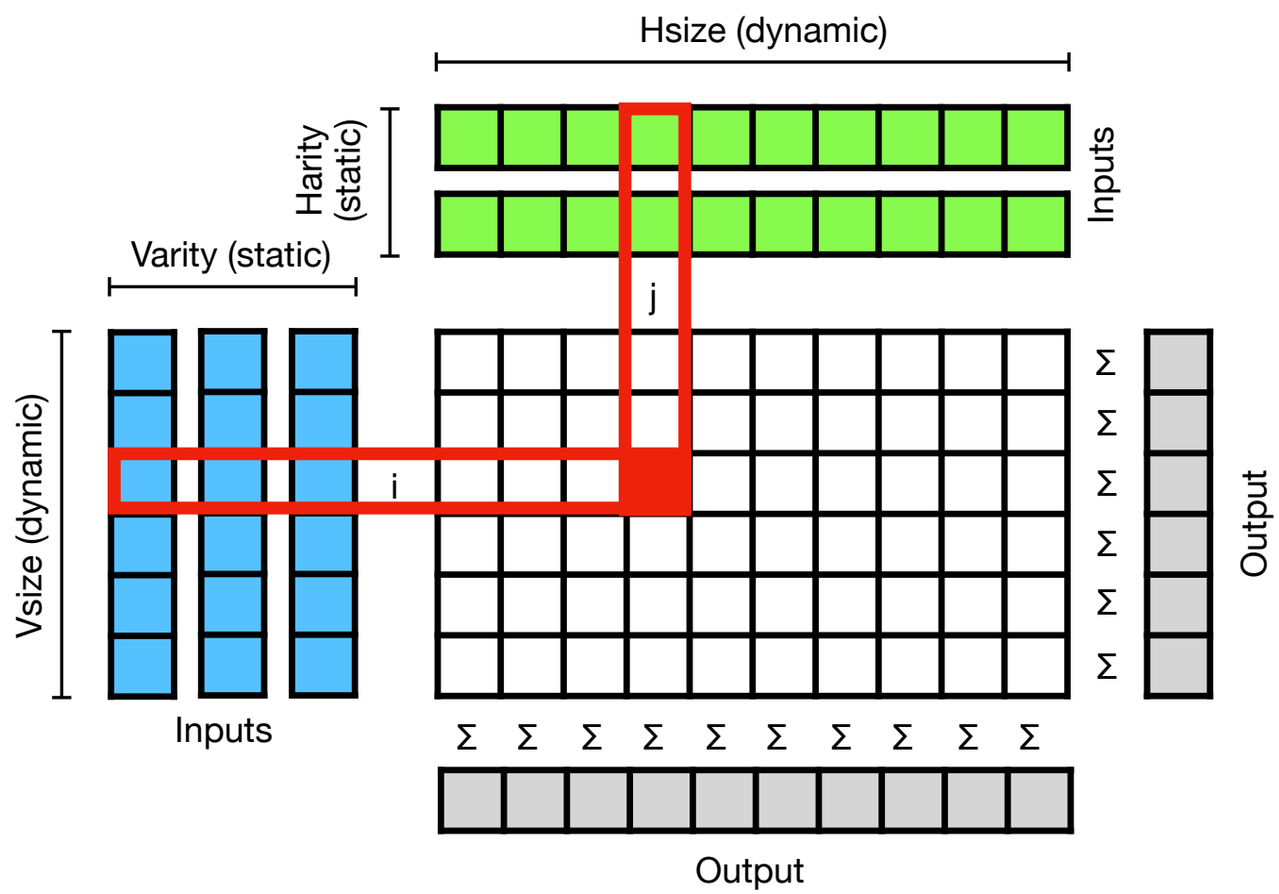


# MapPairsReduce

# MapPairsReduce



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



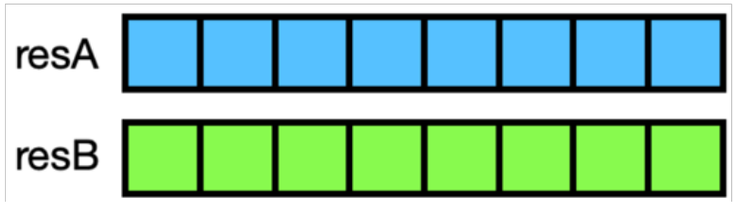
# Variadic Return

# 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



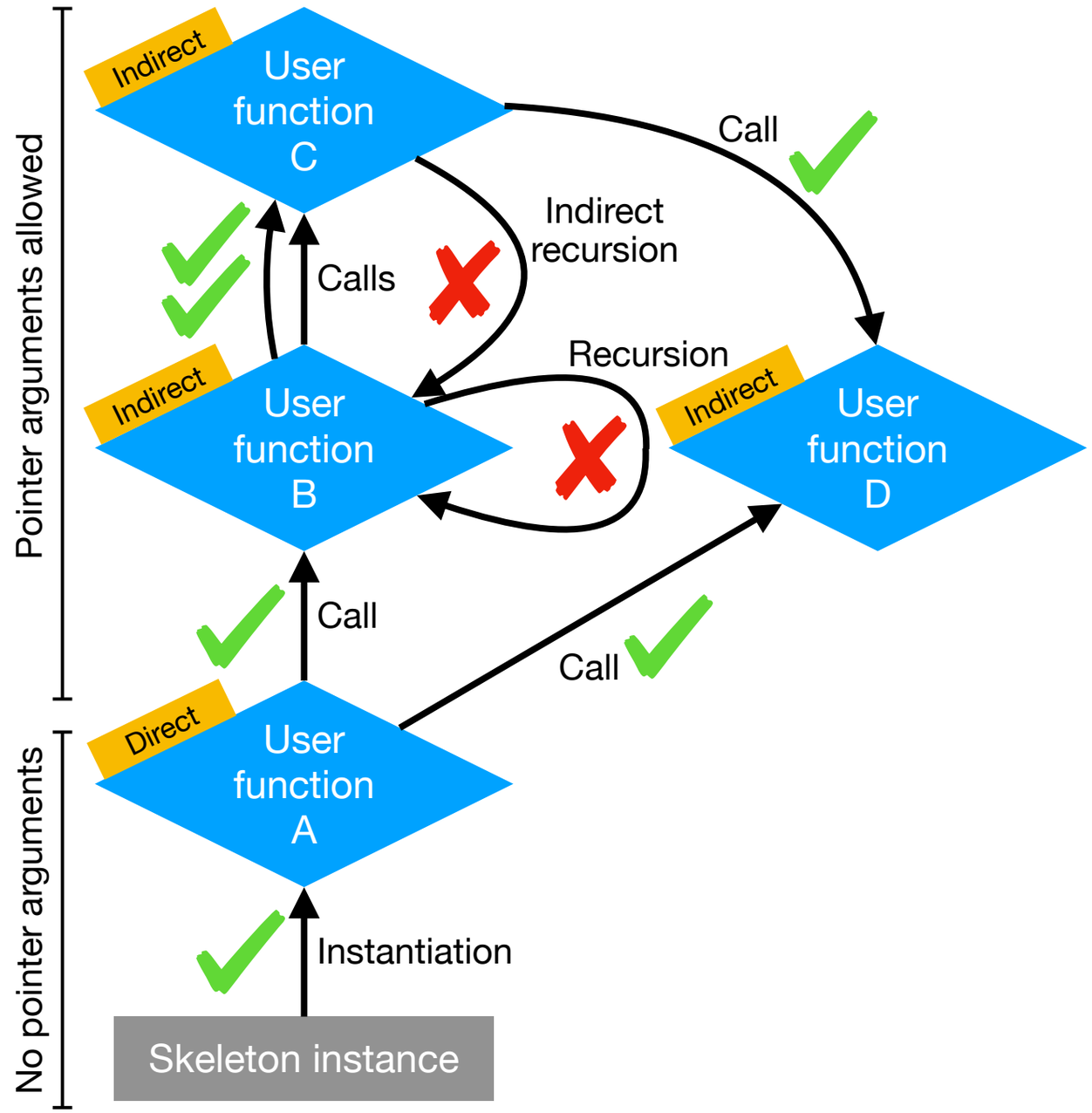
```
skepu::multiple<int, float>
test_f(
  skepu::Index1D index,
  int a, int b, skepu::Vec<float> c, int d)
{
  return skepu::ret(a * b, (float)a / b);
}

// …

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

# DEMO

# SkePU in

# Current Research

exa2pro
Enhancing Programmability and boosting Performance Portability For Exascale Computing Systems

# SkePU in **Teaching**

LINKÖPING UNIVERSITY