SkePU-3 User Guide

August Ernstsson, Johan Ahlqvist, Christoph Kessler

November 25, 2025

Contents

Chapter 1

SkePU Programming and Installation

This chapter gives installation guidelines and a high-level introduction to programming with SkePU[1]

1.1 Introduction

SkePU is a skeleton programming framework for multicore and multi-GPU systems with a C++11 interface. It includes data-parallel skeletons such as `Map` and `Reduce` generalized to a flexible programming interface. SkePU emphasizes and improves on flexibility, type-safety and syntactic clarity over its predecessor, while retaining efficient parallel algorithms and smart data movement for high-performance and energy-efficient computation.

SkePU is structured around a source-to-source translator (precompiler) built on top of Clang libraries, and thus requires the LLVM and Clang source when building the compiler driver.

All user-facing types and functions in the SkePU API are defined in the `skepu` namespace. Nested namespaces are not part of the API and should be considered implementation-specific. The `skepu::` qualifier is implicit for all symbols in this document.

1.2 License

SkePU is distributed as open source and licensed under a modified BSD 4-clause license.

The copyright belongs to the individual contributors.

1.3 Authors and Maintainers

The original SkePU (v1) was created in 2010 by Johan Enmyren and Christoph Kessler [7]. Over the years, a number of people have contributed to SkePU 1, including Usman Dastgeer [3]. The major revision SkePU 2 was designed by August Ernstsson, Lu Li and Christoph Kessler [8]. The major revision towards SkePU 3 was designed by August Ernstsson, Christoph Kessler, Johan Ahlqvist, and Suejb Memeti with input from partners in the EXA2PRO project [9, 10].

August Ernstsson[2] is the current maintainer of SkePU.

1.4 Dependencies and Requirements

SkePU is fundamentally structured around C++11 features and thus requires a mature C++11 compiler. It has been tested with relatively recent versions of Clang and GCC, and NVCC version 9.

[1] The version of SkePU documented here is the forthcoming official release of SkePU 3.3 of late 2025.

[2] august.ernstsson@liu.se

It also uses the STL, including C++11 additions. It has been tested with `libstdc++` and `libc++`. SkePU does not depend on other libraries.

SkePU requires the LLVM and Clang source when building the source-to-source translator. The translator produces valid C++11, OpenCL and/or CUDA source code and can thus be used on a separate system than the target if necessary ("cross-precompilation").

The StarPU MPI backend requires a recent GCC compiler, an OpenMP library, an MPI library (tested with OpenMPI version 2.1), and StarPU built from the master branch.

1.5 Example

We will introduce the SkePU syntax with an example, see Listing 1.1.

The function `ppmcc` calculates a statistical test on two vectors `x` and `y`. It creates three skeleton instances: `sum` as a reduction over ordinary floatingpoint addition; `sumSquare` as a fused map and reduce computing the sum of squares of an operand data-container; and `dotProduct` as another `MapReduce` instantiation with ordinary multiplication and addition, realizing a dot product. After their construction, these skeleton instances can be invoked on SkePU data-container operands like ordinary hand-written C++ functions.

1.6 Installation

This section explains how to build and install the clang-based version of the SkePU pre-compiler, SkePU-tool.

There are three steps to do when building skepu-tool from source:

- Getting the source
- Build skepu-tool
- Install skepu-tool

Getting the source

The public SkePU source code repository is available on GitHub: `https://github.com/skepu/skepu.git`. Clone the main repository to your local system with `git clone`.

The main SkePU repository includes Git submodules.

- LLVM: This is an external dependency to the LLVM project. It is used only for building the SkePU precompiler tool. This submodule repository is patched as part of the initialization process of SkePU, and should not be touched by a SkePU user, or even a SkePU contributor who does not need to make in-depth modifications to the precompiler.
- skepu-headers: This is an internal SkePU submodule which holds the SkePU run-time system, i.e. the template header library. Most SkePU contributors would make changes in this submodule.

Enter the cloned main repository and fetch the submodules by running `git submodule update --init`. This process can take a while, as the LLVM repository is large.

Listing 1.1: Example SkePU 3 user function: A linear congruential generator.

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

// Unary user function
float square(float a)
{
        return a * a;
}

// Binary user function
float mult(float a, float b)
{
        return a * b;
}

// User function template
template<typename T>
T plus(T a, T b)
{
        return a + b;
}

// Function computing PPMCC
float ppmcc(skepu::Vector<float> &x, skepu::Vector<float> &y)
{
        // Instance of Reduce skeleton
        auto sum = skepu::Reduce(plus<float>);

        // Instance of MapReduce skeleton
        auto sumSquare = skepu::MapReduce<1>(square, plus<float>);

        // Instance with lambda syntax
        auto dotProduct = skepu::MapReduce<2>(
                [] (float a, float b) { return a * b; },
                [] (float a, float b) { return a + b; }
        );

        size_t N = x.size();
        float sumX = sum(x);
        float sumY = sum(y);

        return (N * dotProduct(x, y) - sumX * sumY)
                / sqrt((N * sumSquare(x) - pow(sumX, 2))
             * (N * sumSquare(y) - pow(sumY, 2)));
}

int main()
{
        const size_t size = 100;

        // Vector operands
        skepu::Vector<float> x(size), y(size);
        x.randomize(1, 3);
        y.randomize(2, 4);

        std::cout << "X:␣" << x << "\n";
        std::cout << "Y:␣" << y << "\n";

        float res = ppmcc(x, y);
```

Building skepu-tool

SkePU-tool uses a CMake-based build procedure. The CMake scripts requires CMake version 3.13 or later. Best practice is to create an out of source build folder. In the following code snippet, we will use <src>/build. The following commands will build skepu-tool:

```
mkdir build && cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
make
```

A couple of notable build options:

Option name	Default value	Description
SKEPU_ENABLE_TESTING	OFF (Release) ON (Debug)	Enables the test suite for skepu-tool.
SKEPU_BUILD_EXAMPLES	OFF (Release) ON (Debug)	Enables building skepu examples

For more build options, run `cmake -LAH`.

1.7 Usage

The source-to-source translator tool `skepu-tool` accepts as arguments:

- input file path: `-name <filename>`,
- output directory: `-dir <directory>`,
- output file name: `<filename>` (without file extension),
- any combination of backends to be generated: `-cuda -opencl -openmp`.

A complete list of supported flags, and further instructions, can be found by running `skepu-tool -help` on the clang based skepu-tool.

Note that code for the sequential backend is always generated.

SkePU programs (source files) are written as if a sequential implementation—without source translation—was targeted. In fact, such an implementation exists and is automatically selected if non-transformed source files are compiled directly. Make sure to `#include` header `skepu`, which contains all of the SkePU library[3].

We recommend to take a look at the included example SkePU programs and `Makefile`s to get an idea of how everything works in practice.

Include directories

The Clang based `skepu-tool` uses Clang libraries and will perform an actual parse to be able to properly analyze and transform the source code; still, it is not a fully-featured compiler as you would get with a pre-configured package of, e.g., Clang or GCC. This has consequences when it comes to locating platform and system-specific include directories, as these have to be specified explicitly.

By adding the `--` token to the arguments list, you signal that any remaining arguments should be passed directly to the underlying Clang engine. These arguments are formatted as standard Clang arguments. The required arguments are as follows:

- `-std=c++11`;

[3] Almost everything in SkePU is templates, so there is no penalty from including skeletons etc., which are not used.

- include path to Clang's compiler-specific C++ headers, `-I <path_to_skepu>/lib/SkePU/clang-headers`, where the path is the root of the Clang sources (typically in the `tools` directory in the LLVM tree);
- include path to the SkePU source tree: `-I <path_to_skepu>/include`;
- include path(s) to the C++ standard library, platform-specific;
- additional flags as necessary for the particular application, as if it was being compiled.

Debugging

Standard debuggers can be used with SkePU. Per default, SkePU does not use or require exceptions, and reports internal fatal errors to `stderr` and terminates. For facilitating debugging, defining the `SKEPU_ENABLE_EXCEPTIONS` macro will instead cause SkePU to report these errors by throwing exceptions. This should *not* be used for error recovery in release builds, as the internal state of SkePU is not consistent after an error. (The types of errors reported this way are mostly related to GPU management.)

1.7.1 StarPU MPI

To use the StarPU MPI backend, precompile the source with the flags `-openmp -starpu-mpi`. If only the Map skeleton is used in a program, one can also enable the cuda backend. Do not forget to add the link flags and include flags that is needed to compile StarPU codes.

When using the SkePU StarPU MPI backend, do not forget to make sure the code is safe to execute on multiple ranks at the same time. The `skepu::external` function can be used to make a region of code safe. Writing to file is one example where multiple nodes cannot execute the same region at the same time.

1.7.2 Clang skepu-tool and CMake

The clang skepu-tool offers a CMake function to automatically configure the precompilation step. The syntax is as follows:

```
skepu_add_executable(<name> [EXCLUDE_FROM_ALL]
  [[[CUDA] [OpenCL] [OpenMP]] | [MPI]]
  SKEPUSRC ssrc1 [ssrc2 ...]
  [SRC src1 [src2 ...])
```

The function is a wrapper around `add_executable` that will generate precompilation targets for the SkePU sources listed as argument. Any include directories added via `target_include_directory` or `target_link_library` will propagate to the precompilation targets as well. The MPI backend option to `skepu_add_executable` implies OpenMP.

To be able to use the function, add `find_package(SkePU)` to the CMakeLists.txt file. If SkePU is not installed in a path that CMake is aware of, one adds the argument `HINTS <skepu_prefix>` to find_package.

1.8 Limitations

The following section details limitations as of 2020-05-20.

1.8.1 SkePU general

C++ purists, please do not use `const` with SkePU. SkePU is not `const`-correct, due to technical issues.

Known issues with the SkePU headers (in parts possibly outdated, to be revised):

- MapOverlap (all) code generation is disabled for OpenCL (Section 1.11.5).
- MapOverlap (3D, 4D) code generation is disabled for CUDA (Section 1.11.5).
- MapPairsReduce code generation is disabled for CUDA and OpenCL (Section 1.11.8).
- Multi-valued return (Section 1.12) is not implemented for MapOverlap.
- Multi-valued return is disabled for CUDA and OpenCL.
- Scan is missing OpenMP backend selection parameters (thread count, scheduling mode).

1.8.2 StarPU-MPI skeleton backends

The StarPU MPI backend is not very well tested yet. The following list are known issues:

- The data-containers are not fully compatible with the normal version of SkePU.
- The skeletons `MapPool` and `Call` are missing.
- Only the Map, MapPairs, and Reduce skeletons have support for CUDA and applications that use other skeletons cannot have CUDA and StarPU-MPI enabled at the same time.

1.8.3 Clang SkePU tool

In addition, not all combinations of skeleton features are implemented/tested for this release.

1.9 Definitions

Please read through this section once to familiarize yourself with the terms used in this document. It can then be used as a reference, as the terms defined here are typeset in *italics* at first mention in each section.

Skeleton A generic program construct providing a computational pattern (such as map, reduce, stencil, scan etc.), configurable in side-effect-free, problem-specific user code, and operating on data-containers, e.g., vector or matrix operands. A skeleton exposes a sequential-looking interface and encapsulates all implementation details such as parallelism, accelerator usage, memory management, data transfers etc.

The skeletons in SkePU are all data-parallel, i.e., the computation graph is determined by the dependence structure of container parameters and not dependent e.g. on the value of individual elements in a data container.

Data-container An object of some SkePU data-container class, i.e., vector, matrix, or higher-dimensional tensors. Data containers are homogeneous, i.e., contain objects of the same *scalar* (non-container) type. In this document, the term "container" refers exclusively to SkePU data-containers (as opposed to, e.g., raw data pointers or STL vectors); in particular, it has nothing to do with operating system containers.

Scalar The type of elements in a *data-container*. May be a fundamental type such as `float`, `double` or `int` or a compound struct type satisfying certain rules. (Note that the compound types are still referred to as scalar types when in containers.)

User function An operation performed repeatedly (perhaps in parallel) in a *skeleton instance*. A user function in SkePU should not contain side effects, with the exception of writing to *random access arguments*.

Skeleton instance An object of some skeleton type instantiated with one or more *user functions*, and callable with `(...)`like an ordinary hand-written C++ function. May include state, such as

- a *backend specification*,
- an *execution plan*, and
- *skeleton*-specific parameters, such as the initial value for a reduction or the overlap sizes for a stencil.

Skeleton (instance) invocation The process of applying a *skeleton instance* to a set of parameters. Performs some computation as specified by the instance's *skeleton* type and *user function*.

Output argument For the *skeletons* which return a *container*, this container is passed as the first argument in a *skeleton invocation*. If the skeleton instead returns a *scalar*, no argument is passed and the value is instead the evaluated value of the invocation expression (i.e., the return value).

Element-wise parameter/argument A *container* argument to a *skeleton instance*, elements of which, during *skeleton invocation*, is passed to the corresponding *user function* parameter as a scalar value. Iterators into containers can also be used for these parameters to narrow down the range of accessed elements.

Random access parameter/argument A *container* argument to a *skeleton instance*, which, during *skeleton invocation*, is passed as a proxy-container object to the corresponding *user function* parameter so that arbitrary elements can be accessed by the user function.

Uniform parameter/argument A *scalar* argument to a *skeleton invocation*, passed unaltered to each user function call.

Backend The compute units and/or programming interface to use when executing a skeleton

Backend specification An object of type `BackendSpec`. Encodes a *backend* (e.g., OpenMP) along with backend-specific parameters for execution (e.g., number of threads) for use by a *skeleton instance*. Overrides *execution plan*s when selecting backends.

Tuning The process of training a *skeleton instance* on differently sized input data to determine the optimal *backend* in each case.

Execution plan Generated during *tuning* and stored in a *skeleton instance*. Helps select the proper *backend* for a certain input size.

Source-to-source translator / precompiler (`skepu-tool`) Clang-based tool which transforms SkePU programs for parallel execution. Accepts C++11 code as input and produces C++11 / CUDA / OpenCL / OpenMP code as output. Built by user from Clang sources, patched with SkePU-provided extensions.

Host compiler User-provided C++11/CUDA compiler which performs the final build of a SkePU program, producing an executable. Can also be used on raw (non-precompiled) SkePU source for a sequential executable.

1.10 Fundamental SkePU concepts

1.10.1 The basic Map skeleton structure and its derivatives

Map is a term widely used in programming interfaces, sequential as well as parallel, as a name for a construct that transforms a set of values to another set of values in accordance with some transformation (mapping) function f. This function is typically a pure function, deterministic and without side effects, which aids the compiler or interpreter in automatic program translation and optimization. In a statically typed language like C++, the types of the domain and image are fixed but typically they can be different from each other.

SkePU borrows the *map* label for its `Map` skeleton. While `Map` is and does everything mentioned in the preceding paragraph, its versatility and importance in SkePU greatly exceeds that of typical map constructs. `Map` is the *fundamental building block* of the SkePU programming interface: it is the default building block for encoding data parallel computations unless a particularly specific pattern is needed, and in those cases, the vast majority of skeleton patterns in SkePU are directly based upon the foundations of `Map`. Indeed, the names tell the story: `MapReduce`, `MapPairs`, `MapPairsReduce`, `MapPool` and `MapOverlap` are all either specialized variations of `Map` or fusions with another pattern. The important role played by `Map` means that understanding the syntax, capabilities, and limitations of this skeleton is of utmost importance for anyone interested in using or otherwise learning SkePU.

1.10.2 Freely accessible containers inside user functions

Map patterns often only concern themselves with providing a single element from the input data set as argument to the mapping operator. To perform a computation with a non-trivial dependency pattern, the operators can be defined as *lexical closures* which capture the enclosing scope, allowing the use of any free variables inside the operator.

The multi-backend nature of SkePU makes such constructions impractical from an implementation standpoint.[4] The backend environments can have different programming models and the memory spaces are typically separate from the C++ domain perceived by the SkePU user. SkePU therefore require that any auxiliary data structures—limited to smart containers and scalar values—are declared as *bound variables* in the user function signature. There are particular rules for how these objects are declared and passed, discussed in Section 1.10.3.

Proxy Container Objects in User Functions SkePU smart containers are C++ objects of intricate class templates, and cannot be made available in a backend execution context. Therefore, smart containers as bound variables in user functions are encoded as *proxy containers*.

Listing 1.2 illustrates the use of auxiliary smart containers in the matrix-vector multiplication skeleton instance `mvmult`[5] and Figure **??** illustrates how using proxies bring entire container data sets into the user function. This is a `Map` instance with no element-wise inputs, which is a surprisingly powerful construct enabled by the SkePU design principles presented in Section 1.10.3.

[4] SkePU user functions may be defined as lambda expressions, which can act as lexical closures in C++, but SkePU treats them strictly as "syntactic sugar". See Section **??** for further discussion.

[5] Note that this is not the preferred way to encode matrix-vector multiplication since SkePU 3, with the introduction of the `MatRow` proxy container. A better way is shown in Listing **??**.

Listing 1.2: Matrix-vector multiply in the SkePU 2 style, without MatRow.

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

skepu::Vector<float> y(height), x(width);
skepu::Matrix<float> A(height, width);
auto mvmult = skepu::Map(mvmult_f);
mvmult(y, A, x);
```

1.10.3 Multi-variadic type signatures

The central aspect of `Map` which gives it its high flexibility and expressive power compared to many other skeleton programming frameworks is the *multi-variadic interface*.[6] The underlying C++-11 features which enable this generational leap[7] are designed to be used by framework engineers, and the significant complexity of implementation is elegantly hidden beneath the framework boundaries. For the SkePU user, it means that using the map construct is very easy for trivial computations but enables great adaptivity for more involved situations.

A `Map` skeleton instance and the corresponding user function are *four-way variadic*. Arguments of a call to the instance are effectively grouped into four sets:

- output arguments (see Section 1.10.4),
- element-wise input arguments,
- random-access input arguments, and
- uniform input arguments.

The size (henceforth *arity*) of each group is flexible and up to the user to choose based on the use-case at hand. The only restriction is that there has to be at least one output argument.[8] All `Map`-like skeletons in SkePU use the output container to determine the *degree of parallelism*: each element corresponds to an invocation of the user function and is an independent task that could be mapped and scheduled for execution as a unit. It does not matter how each group is ordered internally, but the relative order of each group must be taken: outputs come first, followed by element-wise containers (if any), followed by random-access containers (if any), and finally uniform scalars (if any).

Element-wise ("elwise") parameters in a user function are scalar values (or user types, see Section **??**), with the corresponding arguments in a skeleton invocation being SkePU containers. Each element of the container is uniquely mapped to the parameter of a single user function invocation,

[6] Along with type-safety, flexibility was the main contribution of the original SkePU 2 design [15], and prompted the complete API redesign from SkePU 1. The original impetus for this change was that the SkePU 1 model of having separate skeletons for unary, binary, and ternary Map is not ideal neither from a user nor maintainer perspective in a high-level parallel programming framework.

[7] Mainly variadic templates and advances in template meta-programming: the same techniques behind the implementation of, e.g., `std::tuple` from the C++ standard library.

[8] `Call` is much like a `Map` with no return value or element-wise arguments.

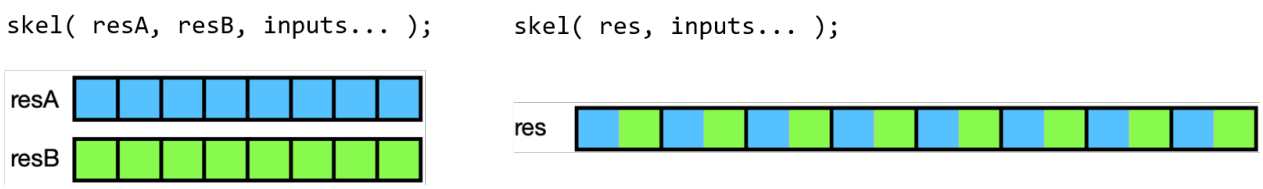

Figure 1.1: Difference in return value storage between using multi-valued return (left) and single-value (by manually managed array-of-struct) return (right).

in a data-parallel fashion. Random access parameters and arguments are both containers (but expressed slightly differently, as explained later) and all elements are accessible from within a single user function invocation.

In user function definitions, the function signature encodes the outputs as the return type of the function and the rest of the arguments come within the parentheses. Extra care has to be observed when crafting a user function, since SkePU uses the function signature when determining the type information for a skeleton instance. Because random-access container arguments are represented as *container proxy types* (see Sections 1.10.2) in the user function signature, the four groupings have natural separations in the type system. SkePU uses template meta-programming and the pre-compiler to analyze the types in the function header and construct the internal groupings. Figure **??** contains an illustration of how the parameter groups bring data from the arguments into the user function in different ways.

Astute readers may notice that the random-access container group is allowed to be empty, in which case the distinction between where the element-wise arguments end and the uniform scalars begin is unavailable. A `Map` instance definition can optionally contain an explicit template argument, as in `auto instance = skepu::Map<N>(...);` where `N` denotes the element-wise arity, and if not present in the construct, SkePU will make a best-guess deduction based on the parameter list (the *formal arguments*) of the user function. Skeleton instances are fully statically typed, so if the deduced arity differs from the *actual arguments* at the skeleton invocation site, a compile-time error occurs.[9]

1.10.4 Multi-valued return

SkePU 3 introduced tuple-like return functionality for cases where a single skeleton instance requires multiple (element-wise) output containers. This way, multiple return values can be computed by the same user function, operating on the inputs in one sweep, potentially improving data locality compared to two separate skeleton invocations after each other. Although the values are returned in a tuple-like manner, the output containers are completely separate objects (see Figure 1.1). This distinguishes this new feature from the use of custom structs ("user types", see Section **??**) as return values, as those are stored in array-of-records format.

To use this feature, we specify the return type in the user function signature as `skepu::multiple<T, [U, ...]>`, i.e., analogous to `std::tuple`. Then, at the site of the `return` statement, we construct this compound object by `skepu::ret(expr, [expr, ...])`.

Listing 1.3 shows an example of a user function utilizing multi-valued return.

The skeleton instance declaration and invocation follows the syntax of ordinary `Map`, but instead of supplying one output container as the first argument, specify several of the correct types and order, as in Listing 1.4.

[9]The pre-compiler has access to the entire AST and can in principle look at both skeleton instantiation and skeleton invocations for arity deduction; however, SkePU is designed and implemented (Chapter **??**) such that programs are semantically sound C++ programs also without the pre-compiler.

Listing 1.3: User function with multi-valued return.

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

Listing 1.4: Using multi-valued return with Map in SkePU 3.

```
skepu::Vector<int> v1(size), v2(size), r1(size);
skepu::Vector<float> e(1);

auto multi = skepu::Map<2>(multi_f);

multi(r1, r2, v1, v2, e, 10);
```

Multi-valued return statements are available in the skeletons which follow the typical map pattern: `Map`, `MapPairs`, and `MapOverlap`.

1.10.5 Index-dependent computations

Another feature of Map-derived SkePU skeletons is the option to access the index for the currently processed container element to the user function. This is handled automatically, deduced from the user function signature. An index parameter's type is one out of four types: `IndexND` where `N` is the dimensionality of the index, as shown in the type declarations in Listing 1.5.[10]

The Mandelbrot fractal generation in the SkePU example programs collection (Section 1.19) is a typical example of a computation where the user function is reliant on the current index into the resulting `Matrix` container.

1.11 Skeletons

SkePU provides a number of skeletons which represent different *data-parallel* patterns, SkePU 3.3 (2025) encompasses currently nine different skeletons:

- `Map`,
- `Reduce`,
- `MapReduce`,

[10] The `Index`*N*`D` feature replaces the dedicated `Generate` skeleton of SkePU 1, allowing for a commonly seen pattern—calling `Generate` to generate a vector of consecutive indices and then pass this vector to MapArray—to be implemented in one single Map call.

Listing 1.5: Index types corresponding to each smart container.

```
struct Index1D { size_t i; };
struct Index2D { size_t row, col; }; // note!
struct Index3D { size_t i, j, k; };
struct Index4D { size_t i, j, k, l; };
```

Table 1.1: Skeleton Feature Matrix (adapted from Ernstsson [9])

Feature **Skeleton:**	`Map`	`MapPairs`	`MapOverlap`	`MapPool`	`Reduce`	`Scan`	`MapReduce`	`MapPairsReduce`	`Call`
Elwise dimension in	1–4	1	1–4	1–4	1–4**	1	1–2	1	0
Elwise dimension out	Same as *in*	2	Same as *in*	Same as *in*	0–1	1	0	1	0
Indexed	Yes	Yes	Yes	Yes	–	–	Yes	Yes	–
Multi-return	Yes	Yes	Yes	Yes	–	–	Yes	Yes	–
Elwise parameters	Variadic	Variadic x2	1***		*	*	Variadic	Variadic x2	–
Full proxy parameters	Variadic	Variadic	Variadic	Variadic	–	–	Variadic	Variadic	Variadic
Uniform parameters	Variadic	Variadic	Variadic	Variadic	–	–	Variadic	Variadic	Variadic
Region/pool proxy	–	–	`Region`*X*`D`	`Pool`*X*`D`	–	–	–	–	–
MatRow/MatCol proxy	Yes	Yes	–	–	–	–	Yes	Yes	–
PRNG stream arg.	Yes	Yes	–	–	–	–	Yes	Yes	–
Element strides	Variadic 1D	–	Dimensional	Dimensional	–	–	–	–	–

Footnotes:
* Parameters to the user functions can be raw elements from the container or partial results, depending on evaluation
** Dimensions higher than 2 are linearized in the current implementation.
*** A `Region`*X*`D` or `Pool`*X*`D` proxy object providing access to elements surrounding the current index is supplied.

- `Scan`,
- `MapOverlap`,
- `MapPool`,
- `MapPairs`,
- `MapPairsReduce`, and
- `Call`.

The skeletons can be loosely ordered into three groups: the map-based `Map`, `MapPairs`, and `MapOverlap`, being element-wise transformations of data; `Reduce` and `Scan`, two forms of data accumulation patterns with internal dependency structures; and explicit *fusions* of a map-based skeleton in sequence with some form of reduction in `MapReduce` and `MapPairsReduce`. `Call` is a pseudo-skeleton and does not fit into any grouping.

Table 1.1 summarizes skeleton attributes and features to show similarities and differences between them.

Each skeleton except for `Call` encodes a computational pattern which is efficiently parallelized. In general, the skeletons are differentiated enough to make selection obvious for each use case. However, there is some overlap; for example, MapReduce is an efficient combination of Map and Reduce in sequence. This makes Reduce a special case of MapReduce.

Most of the skeletons are very flexible in how they can be used. All but Reduce and Scan are variadic, and some have different behaviors for one- and two-dimensional computations.

Skeletons in SkePU are instantiated by calling factory functions named after the skeletons, returning a ready-to-use skeleton *instance*. The type of this instance is implementation-defined and can only be declared as `auto`. This has the consequence of an instance not being possible to declare before definition, passed as function arguments, etc., which is important to consider when architecting applications based on SkePU[11].

SkePU guarantees, however, that a skeleton instance supports a basic set of operations (a "concept" in C++ parlance).

`instance(`*`args...`*`)` *Invokes* the instance with the arguments. Specific rules for the argument list applies to each skeleton.

[11] We are considering different solutions to work around this restriction, please contact the SkePU maintainers if this is important for you.

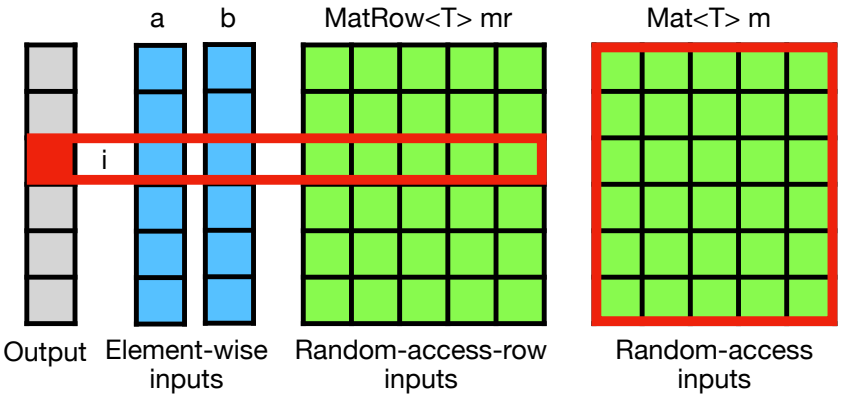

Figure 1.2: Illustrative diagram of the operand access scopes in the Map skeleton.

`instance.tune()` Performs *tuning* on the instance.

`instance.setBackend(`*`backendspec`*`)` Sets a *backend specification* to be used by the instance and overrides the automatic choice.

`instance.resetBackend()` Clears a backend specification set by `setBackend`.

`instance.setExecPlan(`*`plan`*`)` Sets the execution plan manually. The plan should be heap-allocated, and ownership of it is immediately transferred to the instance and cannot be de-referenced by the caller anymore.

1.11.1 Map

The fundamental property of `Map` is that it represents a set of computations without dependencies. The amount of such computations matches the size of the *element-wise* container arguments in the *application* of a `Map` *instance*. Each such computation is a call to (application of) the user function associated with the `Map` instance, with the element-wise parameters taken from a certain position in the inputs. The return value of the user function is directed to the matching position in the output container.

`Map` can additionally accept any number of *random access* container arguments and *uniform* scalar arguments.

When *invoking* a `Map` skeleton, the output container (required) is passed as the first argument, followed by element-wise containers all of a size and format which matches the output container. After this comes all random-access container arguments in a group, and then all uniform scalars. The user function signature matches this grouping, but without a parameter for the output (this is the return value) and the element-wise parameters being scalar types instead. The return value is the output container, by reference.

An example can be found in Listing 1.6.

OpenMP Scheduling Modes

The OpenMP backend in SkePU 3 has changed. It is now possible to control the scheduling mode, as the implementation uses the `runtime` option for OpenMP loop scheduling. The options are static scheduling (default), dynamic scheduling, guided dynamic scheduling, or letting the OpenMP runtime decide. Examples can be found in Listing 1.7.

Listing 1.6: Example usage of the `Map` skeleton.

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

int sum(int a, int b)
{
  return a + b;
}

int main()
{
  skepu::Vector<int> v1{1, 2, 3, 4}, v2{4, 3, 2, 1};
  skepu::Vector<int> result(v1.size());

  auto vsum = skepu::Map(sum);
  vsum(result, v1, v2);

  skepu::io::cout << result << "\n"; // Prints 5 5 5 5
  return 0;
}
```

Listing 1.7: Examples of explicit backend specifications with backend-specific settings.

```
skepu::BackendSpec spec{...};

// OpenMP
spec.setType(skepu::Backend::Type::OpenMP);
spec.setSchedulingMode(skepu::Backend::Scheduling::Static);
spec.setSchedulingMode(skepu::Backend::Scheduling::Dynamic);
spec.setSchedulingMode(skepu::Backend::Scheduling::Guided);
spec.setSchedulingMode(skepu::Backend::Scheduling::Auto);
spec.setCPUChunkSize(/*int*/);

// CUDA + OpenCL
spec.setType(skepu::Backend::Type::CUDA);
spec.setType(skepu::Backend::Type::OpenCL);
spec.setDevices(/*int*/); // number of GPUs to use (default: all)
spec.setGPUThreads(/*int*/);
spec.setGPUBlocks(/*int*/);

// Hybrid
spec.setType(skepu::Backend::Type::Hybrid);
spec.setCPUPartitionRatio(/*float*/); // CPU fraction, range [0, 1]
```

Listing 1.8: Example usage of the Reduce skeleton.

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

int minimum(int a, int b)
{
  return (a < b) ? a : b;
}

int main()
{
  skepu::Vector<int> v{5, 2, 4, 1, 3};

  auto vmin = skepu::Reduce(minimum);
  vmin.setStartValue(v(0)); // start value defaults to 0 otherwise
  int result = vmin(v);

  skepu::io::cout << result << "\n"; // Prints 1
  return 0;
}
```

1.11.2 Reduce

Reduce performs a standard reduction. Two modes are available: 1D reduction on vectors or matrices and 2D reduction on matrices only. An instance of the former type accepts a vector or a matrix, producing a scalar respectively a vector, while the latter only works on matrices. For matrix reductions, the primary direction can be controlled with a parameter on the instance.

The reduction is allowed to be implemented in a tree pattern, so the user function(s) should be associative.

`instance.setReduceMode(`*`mode`*`)` Sets the reduce mode for matrix reductions. The accepted values are `ReduceMode::RowWise` (default) or `ReduceMode::ColWise`.

`instance.setStartValue(`*`value`*`)` Sets the start value for reductions. Defaults to a default-constructed object, which is `0` for built-in numeric types.

An example can be found in Listing 1.8.

Pending Revisions to Reduce Skeleton

The `Reduce` skeleton and the reduce step of `MapReduce` is seeing some changes in SkePU 3.

Reduce modes will be revised to not always trigger data rearrangement such as transposition (sublinear extra memory complexity).

A define is available to enable the old behavior up to a set container size, `-DSKEPU_REDUCE2DCOL_TRANSPOSE_SIZE_MAX [n]`.

The revisions to the `MapReduce` skeleton includes the availability of an additional reduce mode: not only reduction over the entire container span, but also reduction over the innermost dimension (row-wise for matrices).

1.11.3 MapReduce

`MapReduce` is a combination of `Map` and `Reduce` in sequence and offers the most features of both, for example, only 1D reductions are supported.

Listing 1.9: Reduction from matrix to vector, and from vector to scalar.

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

int sum(int a, int b)
{
  return a + b;
}

int main()
{
  skepu::Matrix<int> m(5, 5, 1); // Constructs a 5x5 matrix, each element
     set to 1
  skepu::Vector<int> v(5); // Constructs a vector of length 5

  auto summer = skepu::Reduce(sum);
  summer(v, m);

  skepu::io::cout << v << "\n"; // Prints 5.... 5 5 5 5 5

  summer.setStartValue(0);
  int result = summer(v);

  skepu::io::cout << result << "\n"; // Prints 25

  return 0;
}
```

Listing 1.10: Reduction from matrix to scalar.

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

int sum(int a, int b)
{
  return a + b;
}

int main()
{
  skepu::Matrix<int> m(5, 5, 1); // Constructs a 5x5 matrix, each element
     set to 1

  auto summer = skepu::Reduce(sum, sum);
  summer.setStartValue(0);
  int result = summer(m);

  skepu::io::cout << result << "\n"; // Prints 25

  return 0;
}
```

Listing 1.11: Difference between colwise and rowwise reduction on a matrix.

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

int sum(int a, int b)
{
  return a + b;
}

int main()
{
  skepu::Matrix<int> m(5, 5); // Constructs a 5x5 matrix
  skepu::Vector<int> v(5); // Constructs a vector of length 5

  skepu::external([&]{
    int val = 1;
    for (int row = 0; row < m.total_rows(); ++row)
    {
      for (int col = 0; col < m.total_cols(); ++col)
        m(row, col) = val;
      val += 1;
    }
  }, skepu::write(m));

  skepu::io::cout << m << "\n";
  // Prints:
  // Matrix: (5 X 5)
  // 1 1 1 1 1
  // 2 2 2 2 2
  // 3 3 3 3 3
  // 4 4 4 4 4
  // 5 5 5 5 5

  auto summer = skepu::Reduce(sum);
  summer.setReduceMode(skepu::ReduceMode::RowWise);
  summer(v, m);

  skepu::io::cout << v << "\n"; // Prints 5.... 5 10 15 20 25

  summer.setReduceMode(skepu::ReduceMode::ColWise);
  summer(v, m);

  skepu::io::cout << v << "\n"; // Prints 5.... 15 15 15 15 15

  return 0;
}
```

Listing 1.12: Example usage of the `MapReduce` skeleton.

```
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);
}
```

An instance is created from two user functions, one for mapping and one for reducing. The reduce function should be associative.

`instance.setStartValue(`*`value`*`)` Sets the initial value for reduction. Defaults to a default-constructed object, which is `0` for built-in numeric types.

An example use of `MapReduce`, a dot product computation, can be found in Listing 1.12. The elementwise-access operand arity specifier `<2>` is not really necessary here, as the SkePU implementation can infer it automatically from the signatures of the user functions used at instantiation of the `MapReduce` skeleton.

1.11.4 Scan

`Scan` performs a generalized prefix sum operation, either inclusive or exclusive.

When *invoking* a `Scan` skeleton, the output container is passed as the first argument, followed by a single input container of equal size to the first argument. The return value is the output container, by reference.

`instance.setScanMode(`*`mode`*`)` Sets the scan mode. The accepted values are `ScanMode::Inclusive` (default) or `ScanMode::Exclusive`.

`instance.setStartValue(`*`value`*`)` Sets the start value for exclusive scan. Defaults to a default-constructed object, which is `0` for built-in numeric types.

An example can be found in Listing 1.13.

1.11.5 MapOverlap

`MapOverlap` is a stencil operation. It is similar to `Map`, but instead of a single input element, a region of input elements is available in the user function, see Fig. 1.3. The region is (for CPU backends) passed as a pointer to the center element, see Fig. 1.3.

A `MapOverlap` instance can either be one-dimensional, working on vectors or matrices (the latter with separable filter computations in 1D only) or multi-dimensional for matrices, 3D and 4D tensors. The type is set per-instance and deduced from the user function.

Listing 1.13: Example usage of the Scan skeleton.

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

int sum(int a, int b)
{
  return a + b;
}

int main()
{
  skepu::Vector<int> v{1, 1, 1, 1, 1};
  skepu::Vector<int> result(v.size());

  auto vscan = skepu::Scan(sum);
  vscan.setScanMode(skepu::ScanMode::Inclusive);
  vscan(result, v);

  skepu::io::cout << result << "\n"; // Prints 1 2 3 4 5

  vscan.setScanMode(skepu::ScanMode::Exclusive);
  vscan(result, v);

  skepu::io::cout << result << "\n"; // Prints 0 1 2 3 4
}
```

`MapOverlap` represents a computational pattern with as many names as there are application domains. It is known as a *convolution* in signal processing, *stencil filter* in image processing, *window function* in statistics, and so on. The SkePU name of *MapOverlap* indicates that it is another variant of the archetypal map pattern, which would typically indicate that there is a degree of parallelism equal to the number of elements in the result container. This is almost true, but not quite: the number of user function invocations—and therefore schedulable tasks—follows this metric, but the "overlap" part of the name reveals that these tasks are not independent. In a `MapOverlap` user function, not only a single element-wise mapped element from an input container is accessible, but also a *region* of surrounding elements. The individual access regions in the input data-container for neighbored output element positions *overlap* each other, which therefore gives rise to read-after-write dependencies between user function invocations and in general creates a more complex dependency structure between input and output container elements.

In SkePU, the surrounding region is always a *hyper-rectangle*, i.e., a regular multi-dimensional box in up to four dimensions. The side length of the hyper-rectangle can vary in each dimension, and is defined by a *overlap radius*, which is the number of included elements away from the center element. Therefore, the total amount of elements included in the overlap region is $\prod_i^D (1 + 2o_i)$, where o_i is the overlap radius for dimension i and D is the number of dimensions of the `MapOverlap` instance as determined from its user function. See also Figure 1.4 for illustration.

A `MapOverlap` example showing a one-dimensional convolution is shown in Listing 1.14.

MapOverlap skeleton instances in SkePU can be of several different types:

- **1D MapOverlap** on
  - vector containers or
  - matrix containers with

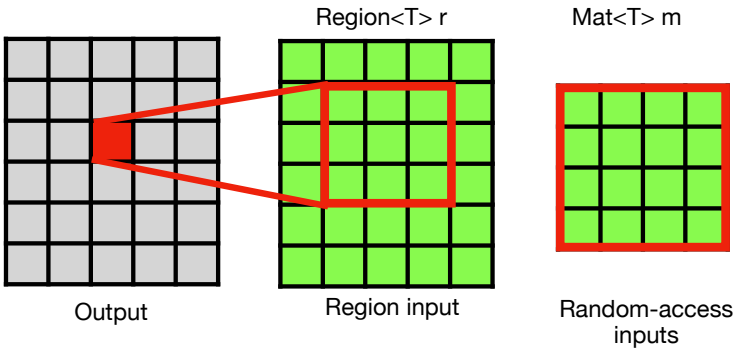

Figure 1.3: Data access in a `MapOverlap` skeleton instance operating on 2D data-containers. When calculating an output element (left), the user function has read random-access to all elements within the `Region2D` data-container proxy object (middle). Note that `Region2D` differs from the `Mat` proxy object (right) which gives read random-access to all elements in the input data-container.

  - row-wise overlap,
  - column-wise overlap,
  - row-wise overlap followed by column-wise overlap (two passes), or
  - column-wise overlap followed by row-wise overlap (two passes).

- **2D MapOverlap**
  On matrix containers.
- **3D MapOverlap**
  On three-dimensional tensor containers.
- **4D MapOverlap**
  On four-dimensional tensor containers.

The dimensionality of a MapOverlap instance is determined by the N in the `Region`N`D<T>` type used for the element-wise argument in the user function. These are compiler-known types and dictate which internal implementation variant of the `MapOverlap` skeleton to use for code generation. Note that the dimensionality of the `MapOverlap` pattern encoded in the skeleton instance does not necessarily match the dimension of the smart data-containers the instance is applied on. In principle, there could be a `MapOverlap` variant for any overlap dimension smaller than or equal to the dimension of the element-wise container input. However, for practical reasons, only the combinations listed above are implemented in SkePU.

In SkePU 3, a `MapOverlap` user function accepts a *data-container proxy* argument of type `Region`N`D`, where N is the data-container's dimensionality. It provides read access to all input data-container elements "under the stencil", indexed relatively to the stencil's center element which matches the position of the corresponding element in the calculated output data-container being calculated by applying the stencil. The syntax for a stencil computation using `MapOverlap`, namely 1D convolution with a 1D five-point stencil, can be seen in Listing 1.14.

Edge handling modes

When MapOverlap user functions are evaluated near the edges of the input container, the overlapping region may reach outside the bounds of the input. The expected behavior of out-of-bounds overlap regions are application-dependent, but to avoid invalid memory accesses, the implementing framework must do something to handle these scenarios. SkePU approaches this problem in several ways. There are a total of four options for edge handling, three of which are proper edge-handling modes:

Listing 1.14: Example usage of the `MapOverlap` skeleton: 1D convolution with a 1D five-point stencil.

```
float conv ( skepu::Region1D<float> r, const skepu::Vec<float> stencil )
{
 float res = 0;
 for (int i = -r.oi; i <= r.oi; ++i)
    res += r(i) * stencil(i + r.oi);
 return res;
}

skepu::Vector<float> convolution ( skepu::Vector<float> &v )
{
 auto convol = skepu::MapOverlap( conv );
 Vector<float> stencil {1, 2, 4, 2, 1};  // stencil coefficients
 Vector<float> result(v.size());         // output data-container
 convol.setOverlap(2);
 return convol ( result, v, stencil );
}
```

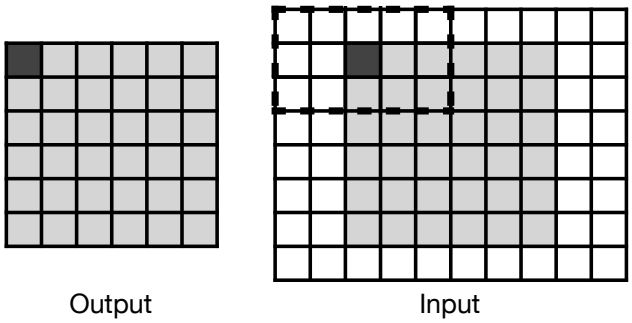

Figure 1.4: Expected input and output container sizes when edge element synthesis is disabled, here in 2D `MapOverlap`.

- no edge handling (default for 2D, 3D, and 4D `MapOverlap`),
- fixed padding with a user-set value,
- duplicate padding of the value closest to the edge (default for 1D `MapOverlap`), or
- cyclic (toroidal) padding.

If the "no edge handling" option is specified, SkePU requires that the *size* of the input container is larger than the size of the output container, to ensure that all user function evaluations correspond to a well-defined overlap region. Figure 1.4 illustrates this restriction: the overlap radius in this example is 2 in the x-axis and 1 in the y-axis, and the output[12] container size is 6×6 elements. The input container is therefore expected to be of size $6 + 2 * 2 = 10$ in the horizontal dimension and $6 + 2 * 1 = 8$ in the vertical dimension.

In all other modes, the output container will be of equal size to the input container, and in cases where the overlap region intersects the container boundaries, SkePU synthesizes virtual elements for out-of-bounds accesses. The properties of each mode is visualized in Figure 1.5.

Synthesis of out-of-bounds elements adds some run-time overhead, but auxiliary memory usage is kept low: proportional to the overlap region size, not to the input data size. Depending on various

[12]Recall that SkePU always parallelizes skeletons on the output container range.

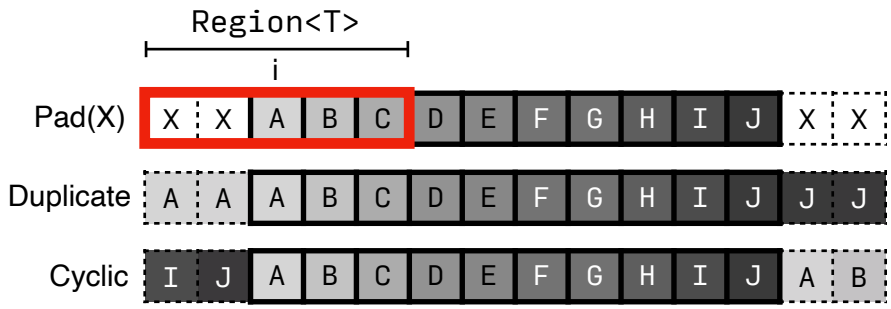

Figure 1.5: Edge handling modes of 1D `MapOverlap`.

aspects of the skeleton instance at hand (especially container type), elements in the region may be either pre-allocated or synthesized lazily upon access.

MapOverlap API

The parameter list for a user function to `MapOverlap` is important. It always starts with an `int`, which is the overlap radius in the x-direction. 2D `MapOverlap` also has another `int`, which will bind to the y-direction overlap radius. The presence of this parameter is used to deduce that an instance is for 2D. A `size_t` parameter follows, this is the stride. The next parameter is a pointer to of the contained type, pointing to the center of the overlap region. *Random-access* container and *uniform* scalar arguments follow just as in `Map` and `MapReduce`.

`instance.setOverlap(`*`radius`*`)` Sets the overlap radius for all available dimensions.

`instance.setOverlap(`*`i_radius, j_radius`*`)` For 2D `MapOverlap` only. Sets the overlap for i and j directions. **Note**: i corresponds to the y-axis and j corresponds to the x-axis.

`instance.setOverlap(`*`i_radius, j_radius, k_radius`*`)` For 3D `MapOverlap` only. Sets the overlap for i, j, and k directions.

`instance.setOverlap(`*`x_radius, y_radius, k_radius, l_radius`*`)` For 4D `MapOverlap` only. Sets the overlap for i, j, k, and l directions.

`instance.getOverlap()` Returns the overlap radius: a single value for 1D `MapOverlap`, a `std::pair` (i, j) for 2D `MapOverlap`, and `std::tuple`s for 3D and 4D.

`instance.setEdgeMode(`*`mode`*`)` Sets the mode to use for out-of-bounds accesses in the overlap region. Allowed values are `Edge::Pad` for a user-supplied constant value, `Edge::Cyclic` for cyclic access, or `Edge::Duplicate` (default) which duplicates the closest element.

`instance.setOverlapMode(`*`mode`*`)` For 1D MapOverlap: Sets the mode to use for operations on matrices. Allowed values are `Overlap::RowWise` (default), `Overlap::ColWise`, `Overlap::RowColWise`, or `Overlap::ColRowWise`. The latter two are for separable 2D operations, implemented as two passes of 1D MapOverlap.

`instance.setPad(`*`pad`*`)` Sets the value to use for out-of-bounds accesses in the overlap region when using `Edge::Pad` overlap mode. Defaults to a default-constructed object, which is `0` for built-in numeric types.

`instance.setupdateMode(`*`mode`*`)` Sets the mode of update to use for a single invocation to this instance. Options: `UpdateMode::Normal`, `UpdateMode::RedBlack`, `UpdateMode::Red`, `UpdateMode::Black`.

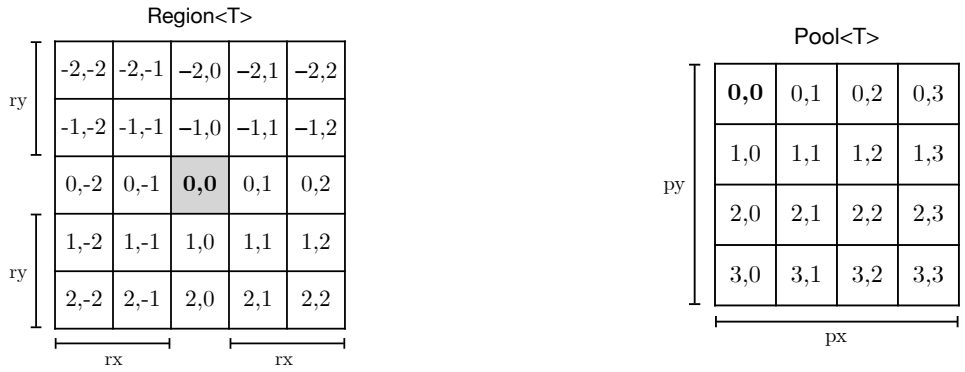

Figure 1.6: Left: A 2D `Region` proxy container object, for use with `MapOverlap`. The output container after convolution of an input container with this region object will, by default, have extents that are shorter by $2ry$ rows and $2rx$ columns, compared to the input container. — Right: A 2D `Pool` proxy container object, for use with `MapPool`. The output container after pooling will have by a factor of $py \times px$ fewer elements than the input container.

1.11.6 MapPool

The `MapPool` skeleton has been added to SkePU in 2025 in combination with the introduction of SkePU-DNN as part of the SkePU standard library [21]. It represents the *pooling* pattern, a partial, 2D blockwise reduction of a 2D, 3D or 4D input data container modeling an image (set) processed in convolutional neural networks. Pooling results in an output container of same dimensionality as the input container, but with accordingly lower extents in the two pooled dimensions. A common reduction operator used in CNN pooling layers is maximum ("max-pooling").

The block of elements on which the partial multidimensional reduction is applied is provided as a `Pool`X`D<>` proxy container object ($X \in \{1, 2, 3, 4\}$) passed to the user function of a `MapPool` instance. `Pool`X`D<>` is `MapPool`'s equivalent of the `Region`X`D<>` proxy container object used in combination with the `MapOverlap` skeleton, with the difference that `Region` objects always have odd extents and are indexed positively and negatively relative to the center point $(0, 0)$, while for `Pool` objects the origin $(0, 0)$ of intra-block indexing is in the left upper corner so that only positive indices occur, see also Figure 1.6 and the examples in Listings 1.15 (max-pooling) and 1.16 (grayscaling). Another difference is that the regions of neighbored output container elements overlap to a significant degree in the input container (by how much, depends on the `Region`'s extents), while pools of neighbored target container elements never overlap in the source container.

Like `Region<>`, a `Pool<>` proxy container object is generic in the coefficient element type it contains. The pool's elements refer to the corresponding input container elements being block-reduced to a single value.

... TBD: list of configuration options of MapPool instances

1.11.7 MapPairs

SkePU 3 added an additional top-level skeleton, `MapPairs`. This skeleton represents a Cartesian product-style pattern, operating on two distinct sets of element-wise container inputs. Each vector set may contain an arbitrary number of vector containers, similar to the variadicity of `Map`. All of the vectors in a set are expected to be of the same size. The arities in both directions are always present in the skeleton construction as explicit template arguments.

Each Cartesian combination of vector set indices generates one user function invocation, the result of which is an element in a `Matrix`. As in `Map`, there is an optional `Index2D` parameter in the user function signature to access this index.

Listing 1.15: 2×2 max-pooling computation with `MapPool` skeleton along dimensions 2 and 3 over 4D tensor input and output containers.

```
float max_pooling_uf(skepu::Pool4D<float> pool) {
        float maxval = pool(0,0,0,0);
        for (size_t j = 0; j < pool.sj; ++j)
                for (size_t k = 0; k < pool.sk; ++k) {
                        float val = pool(0, j, k, 0);
                        maxval = (maxval > val) ? maxval : val;
                }
        return maxval;
}
auto skel_pool_max = skepu::MapPool(max_pooling_uf);

// input, output are of type skepu::Tensor4<float>
skel_pool_max.setPoolSize(1, 2, 2, 1);
skel_pool_max(output, input);
```

Listing 1.16: Grayscaling of each image pixel from its RGB values stored in the innermost dimension (4) of a 4D tensor input container holding a batch of RGB images, using the `MapPool` skeleton and a user function taking a $(1 \times 1 \times 1 \times 3)$ `Pool4D<>` proxy container.

```
float gray_scale_kernel(skepu::Pool4D<float> pool) {
    return (0.2989f * pool(0,0,0,0) + // red channel
            0.5870f * pool(0,0,0,1) + // green ch.
            0.1140f * pool(0,0,0,2)); // blue ch.
}
auto grayscaler = skepu::MapPool(gray_scale_kernel);
grayscaler.setPoolSize(1,1,1,3);
```

Listing 1.17: Example usage of the MapPairs skeleton.

```
int mul(int a, int b) { return a * b }

void cartesian(size_t Vsize, size_t Hsize)
{
  auto pairs = skepu::MapPairs(mul);

  skepu::Vector<int> v1(Vsize, 3), h1(Hsize, 7);
  skepu::Matrix<int> res(Vsize, Hsize);
  pairs(res, v1, h1);
}
```

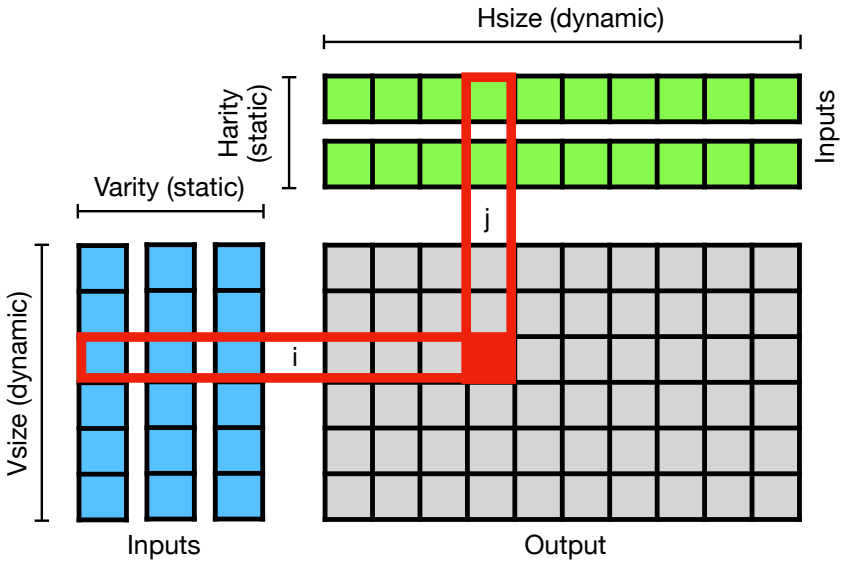

Figure 1.7: Illustrative diagram of the MapPairs skeleton.

Advanced and more flexible use of `MapPairs` can be carried out similarly to other SkePU skeletons. For instance, it retains flexibility of `Map` with regards to variadicity (5-way variadic, compared to `Map` being four-way):

- Resulting outputs (see Section 1.10.4),
- Element-wise-V ("vertical", column-aligned) input arguments,
- Element-wise-H ("horizontal", row-aligned) input arguments,
- Random-access input arguments,
- Uniform input arguments.

A `MapPairs` instance of higher arity looks like

```
auto pairs = skepu::MapPairs<3, 2>(...);.
```

This instance would accept three vertical and two horizontal input vectors.

1.11.8 MapPairsReduce

`MapPairsReduce` is the combination of a `MapPairs` followed by a row-wise or column-wise reduction over the generated matrix elements. It returns a `Vector` containing the row-wise or column-wise reduction, where the reduction dimension is specified as in 2D `Reduce`. Example usage of this skeleton can be seen in Listing 1.18.

Just like `MapReduce`, the skeleton is initialized with two user functions: one matching the format of a MapPairs user function, and one meeting the restrictions of a Reduce user function.

`MapPairsReduce` supports arity `<0,0>` and up. If the arity is 0 in a dimension, it will determine the size of the intermediate Matrix by the values given in a call to `setDefaultSize(<Hsize>, <Vsize>)` member function beforehand.

The intermediate `Matrix` is not guaranteed to be stored in memory at any point.

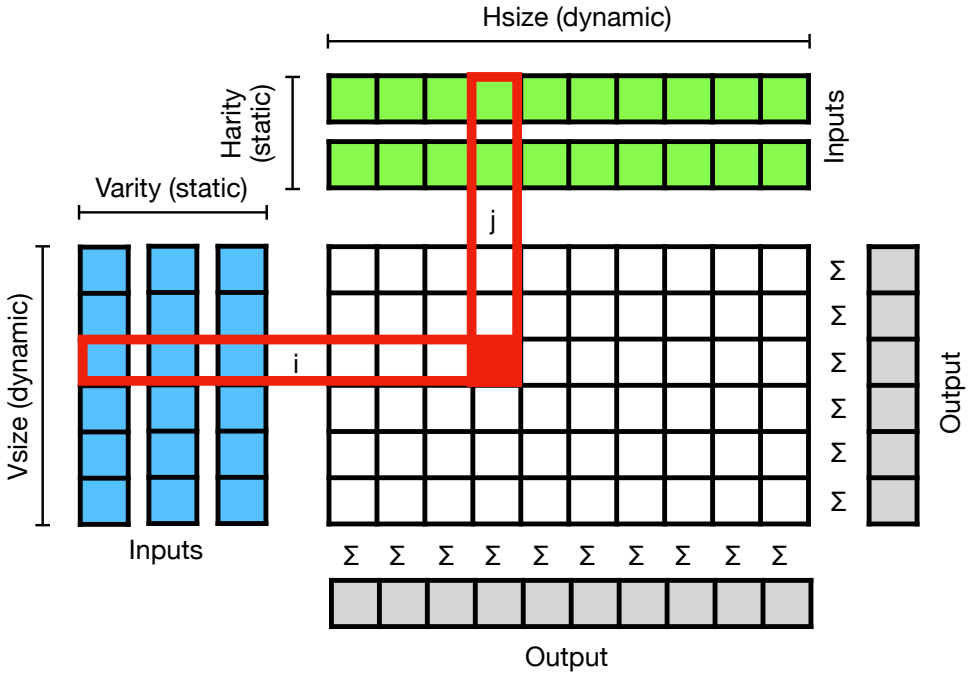

Figure 1.8: Illustrative diagram of the `MapPairsReduce` skeleton.

1.11.9 Call

`Call` is special in that it does not provide any pre-defined structure for computation. It is a way to extend SkePU for computations which does not fit into any skeleton, while still utilizing features such as smart containers and tuning. As such, `Call` provides a minimal interface.

The `Call` skeleton is deprecated since SkePU 3.1 and replaced by the `skepu::external` construct.

1.12 Multi-Valued Return from Skeletons and User Functions

SkePU 3 introduces tuple-like return functionality for cases where a single skeleton instance requires multiple (element-wise) output containers. This way, multiple return values can be computed by the same user function, operating on the inputs in one sequence, potentially improving data locality compared to two separate skeleton invocations after each other. Though the values are returned in a tuple-like manner, the output containers are completely separate objects. This distinguishes this new feature from the existing use of custom structs as (inputs or) return values, as those are stored in array-of-records format.

To use this feature, specify the return type in the user function signature as `skepu::multiple<[basic_type, ...]>`, i.e., analogous to `std::tuple`. Then at the site of the `return` statement, construct this compound object by `skepu::ret([expression, ...])`.

Listing 1.19 gives an example of a user function utilizing this:

The skeleton instance declaration and invocation follows the syntax of ordinary `Map`, but instead of supplying one output container as the first argument, specify several of the correct types and order. An example is given in Listing 1.20.

Multi-valued return statements are available in the `Map` skeleton.

Listing 1.18: Example usage of the MapPairsReduce skeleton.

```
int mul(int a, int b)
{
  return a * b;
}

int sum(int a, int b)
{
  return a + b;
}

void mappairsreduce(size_t Vsize, size_t Hsize)
{
  auto mpr = skepu::MapPairsReduce(mul, sum);

  skepu::Vector<int> v1(Vsize), h1(Hsize);
  skepu::Vector<int> res(Hsize);

  mpr.setReduceMode(skepu::ReduceMode::ColWise);
  mpr(res, v1, h1);
}
```

Listing 1.19: Example of a user function using multi-valued return.

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

Listing 1.20: Example for a skeleton instance using multi-valued return.

```
skepu::Vector<int> v1(size), v2(size), r1(size);
skepu::Vector<float> e(1);

auto multi = skepu::Map<2>(multi_f);

multi(r1, r2, v1, v2, e, 10);
```

Listing 1.21: Example for strided access to elementwise accessed data-container operands

```
int f(int a, int b) { /* ... */ };
auto mapper = skepu::Map(f);
mapper.setStride(2, 4, 3);

skepu::Vector<int> out(16), in_a(N_A), in_b(N_B);
mapper(out, in_a, in_b); // out stride = 2, in_a stride = 4, in_b stride =3
```

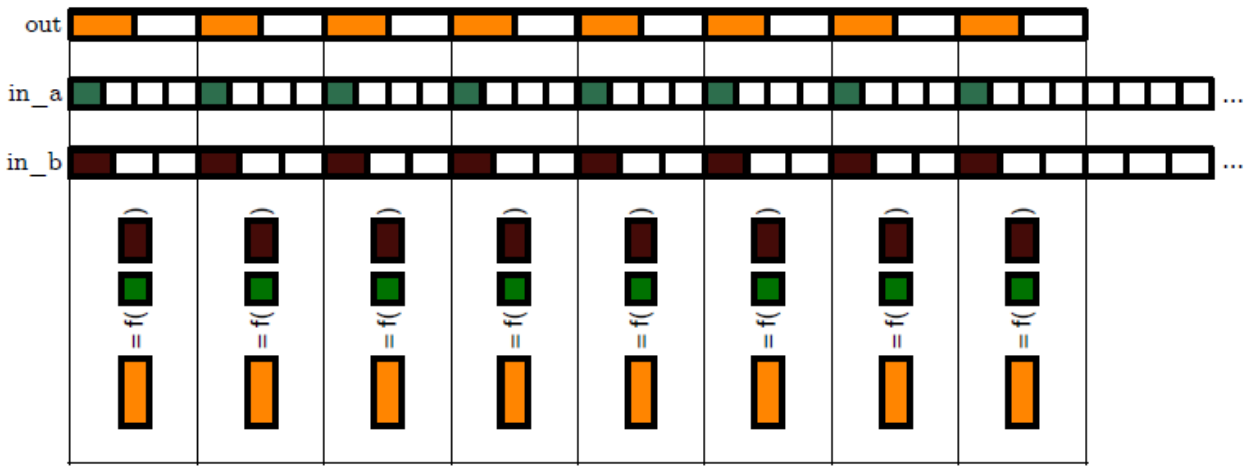

Figure 1.9: Strided elementwise access example. Image source: A. Ernstsson [9]

1.13 Strided Access to Data-Containers

Strided access to elementwise accessed container operands with non-unit strided is supported for all Map-based skeletons since 2025. An example can be found in Listing 1.21, resulting in the access pattern of Figure 1.9.

1.14 Manual Backend Selection and Default Settings

In SkePU 3 it is possible to set a global backend specification, either externally or at runtime by calling the function `setGlobalBackendSpec`. This specification will be used by default for all skeleton instances. Overriding can be done on an instance basis using the member function `setBackend`, see the example in Listing 1.22.

Listing 1.22: API Mechanisms for manual backend selection in SkePU

```
skepu::BackendSpec spec{/*string or SkePU::Backend::Type enum*/};

skepu::setGlobalBackendSpec(spec);

skepu::restoreDefaultGlobalBackendSpec();

skel.setBackend(other_spec); // now overrides global specification
```

1.15 Tuning of Skeleton Instances

A skeleton instance can be tuned for backend selection by going though a process of training on different input sizes of the *element-wise* arguments. This process is automated, but since there is significant overhead (during the tuning process, not afterwards) it has to be started manually. An instance is tuned by calling `instance.tune()`. Note that this is an experimental feature with limitations. Only the size of element-wise arguments can be used as the tuner's problem size, which is not applicable to all types of computations possible with SkePU.

Tuning creates an internal *execution plan* for each skeleton instance, which is used as a look-up table during skeleton instance invocation. It is also possible to construct such a plan manually, and assign it to the skeleton instance.

1.16 Smart Data-Containers

The smart data-containers available in SkePU are `Vector`, `Matrix`, `Tensor3` and `Tensor4`. They are called "smart" because they internally perform run-time optimizations such as coherent software caching of accessed elements in GPU device memory [4] and some other optimizations [13]. Using these is mostly transparent, as they will optimize memory management and data movement dynamically between CPU and GPUs.

There is also a manual low-level interface to the data containers' coherence mechanism in order to explicitly control coherence and data movement:

- `container.updateHost()` forces download of up-to-date data from the GPUs (i.e., a flush operation), and
- `container.invalidateDeviceData()` forces a re-upload at the next skeleton invocation on a GPU (i.e., an invalidation operation).

Data-container element access in user functions should normally use the `operator(`*`index`*`)` access. Coherence for container elements is maintained at skeleton calls only. Element access on the CPU (e.g., in I/O code) can be done either with the—now deprecated—`operator[`*`index`*`]`, which includes overhead for automatically checking for the coherence state of remote copies, or `operator(`*`index`*`)` which provides direct, no-overhead access, if the coherence state of the container is clear from the context (e.g., after a manual flush operation).

When smart data-containers are *not* used as *element-wise accessed parameters* to user functions (which is the default access type in `Map`) but require random access to multiple or all element, it is important to note that separate data-container proxy object types are to be used in user-function code, such as `Vec` and `Mat`. These proxy types do not provide the full smart data-container functionality and are rather used with a C-style interface for better portability to accelerator system platforms not supporting full C++, such as OpenCL. Elements are retrieved using *`proxycontainer`*`.data[`*`index`*`]` member, and `size`, `rows`, `cols` etc. are members and not member functions. By default, the arguments are read/writeable and will encur copy operations both up and down from GPUs; by adding the `const` qualifier, the copy-down is eliminated. Similarily, a `[[skepu::out]]` attribute will turn them into output parameters.

1.16.1 Smart Data-Container Set

The SkePU smart data-container set includes array abstractions in one to four dimensions: 1D "vector", 2D "matrix", and tensors of three and four dimensions. Smart data-container dimensionality in SkePU 3 is therefore static, though their extents (sizes in each dimension) are dynamic.

Instances of these tensor types are created with one constructor argument for each dimension. Optionally an additional argument of type `T` specifices the default value of all elements in the container.

```
skepu::Vector<float> v(dim1);
skepu::Matrix<float> m(dim1, dim2);
skepu::Tensor3<float> t3(dim1, dim2, dim3);
skepu::Tensor4<float> t4(dim1, dim2, dim3, dim4);
```

The `Index`*XD* object set in SkePU, useable in e.g. user function signatures, comprises these structs for 1D to 4D container access, respectively:

```
struct Index1D { size_t i; };
struct Index2D { size_t row, col; }; // note!
struct Index3D { size_t i, j, k; };
struct Index4D { size_t i, j, k, l; };
```

Note that the naming convention is different for matrix indices for compatibility reasons.

Tensor Usage

`Tensor3<T>` and `Tensor4<T>` are useable in a way analogous to `Matrix<T>` in most cases.

- `Map`: Over the full domain without regard to dimensionality Optional argument `Index3D` for `Tensor3<T>` and `Index4D` for `Tensor4<T>` ufs
- `Reduce`: Over full domain or over innermost dimension.
- `MapReduce` See `Map`. For the reduce step, over full domain or the innermost dimension.
- `Scan`: Over full domain.
- `MapOverlap`: Overlap radius limited to 1 in each dimension (default). Larger overlap is possible, dependent on backend support.
- `Call`: See `Map`
- `MapPool`: ... TBD

`MapPairs` and `MapPairsReduce` are not supported for 3D and 4D tensors.

1.16.2 Smart Container Element Access

SkePU 3 **deprecates** the angle bracket `[ ]`-notation for smart container element read/write access outside user functions. This is part of a simplification of the coherency systems for manual element access from the host (CPU) side.

The user should flush the whole container instead before doing single-element accesses of user function data, see Section 1.16.2.

Instead of angle brackets, the parentheses `( )`-notation is extended to higher dimensionality. This syntax accepts one index argument for each dimension of the underlying container. The indices count must equal container dimensionality, otherwise there is a compile-time error.

Formally, the syntax is `container(i[, j[, k[, l]]]) [= value];`

This change means that there is no interface for 1D indexing of higher-dimensionality containers.

There is no longer a coherency-satisfying single-element access mechanism in SkePU smart containers except inside user function proxy objects (`Vec<T>`, `Mat<T>`, etc). However, for correctness debugging purposes, there is a macro to enable explicit flush of a container upon access,
`-DSKEPU_ALWAYS_UPDATE_HOST_ON_CPU_ELEMENT_ACCESS [0,1]`,
but note that this has serious performance implications.

Memory Coherency

SkePU smart data-containers software-cache elements in device memory when used with a GPU backend. Due to lazy data transfers in an attempt of avoiding unnecessary data moves with subsequent skeleton calls, cached and modified elements are only copied back to main memory at their next use on CPU. This coherence mechanism is fully managed by SkePU—but only for data-container element accesses that SkePU knows about, i.e., within skeleton instance calls. In some cases, e.g., for C/C++ I/O operations with global side-effects beyond the scope of SkePU, cached elements need to be flushed manually.

SkePU provides a flush operation with options:

```
enum class FlushMode  Default, Dealloc ;
```

where `FlushMode::Default` is implicit if no other value is given.

There is a data-container member function `flush` as well as a variadic free template function `flush`. The member function accepts a dynamic flush mode as an optional argument, which can be selected at runtime. The free function takes the flush mode as a static constant which is known to the compiler (and precompiler).

```
skepu::Vector<int> v1(n), v2(n);
skepu::Matrix<int> m1(n, n), m2(n, n);

v1.flush(); // FlushMode::Default
m1.flush(); // FlushMode::Default

skepu::flush(v2, m2); // FlushMode::Default

v1.flush(skepu::FlushMode::Dealloc);
m1.flush(skepu::FlushMode::Dealloc);
skepu::flush<skepu::FlushMode::Dealloc>(v2, m2);
```

There is no `#pragma` for flush declarations in SkePU, but the flush (member) functions are compiler-known symbols to the precompiler, as are smart container classes, so the presence or absence of flush operations in SkePU source code is subject to static analysis and optimization.

1.16.3 Matrix-row User Function Proxy Containers

SkePU has since version 2 allowed for flexible parameter lists for user functions, including so-called *random-access* containers, in addition to the, for skeleton programming standard, element-wise mapped containers. While this allows for powerful expressivity, very little about the access patterns of these random-access containers is known to SkePU, and performance may thus not always be ideal.

One common pattern when using `Matrix` as a random-access container argument is that each user function invocation is only interested in one row of the matrix. This pattern is seen in matrix-vector multiplication and similar multi-reduction-style computations. To improve SkePU performance in these cases, SkePU 3 introduces a new proxy container object, `MatRow<T>`. Bridging the gap between element-wise mapped and random-access container arguments, this proxy type when used in a `Map` skeleton instance that maps over vectors (i.e., the result container(s) of the skeleton are `Vector`), makes available one single row of the argument matrix container to the user function.

Note: it is required that the matrix container has at least as many rows as the result vector has elements.

Example Matrix-vector multiplication using `MatRow<T>` may be implemented as folows:

```
template<typename T>
T arr(const skepu::MatRow<T> mr, const skepu::Vec<T> v)
{
        T res = 0;
        for (size_t i = 0; i < v.size; ++i)
                res += mr.data[i] * v.data[i];
        return res;
}
```

Compared to the closest corresponding SkePU 2 implementation below (still valid in SkePU 3), the code is more succinct and there is more information about the access pattern avaiable to SkePU.

```
template<typename T>
T arr(skepu::Index1D row, const skepu::Mat<T> m, const skepu::Vec<T> v)
{
        T res = 0;
        for (size_t i = 0; i < v.size; ++i)
                res += m.data[row.i * m.cols + i] * v.data[i];
        return res;
}
```

There is no change in syntax of skeleton instantiation or skeleton invocation needed for this feature to apply.

Matrix-row user function proxy containers are available in user functions for `Map`, `MapReduce`, and `MapOverlap` skeleton instances that satisfy the above requirements.

1.17 Using Custom Types

It is possible to use custom types in SkePU containers or inside user functions. These types should be C-style structs for compatibility with OpenCL. **Note**: It is not guaranteed that a struct has the same data layout in OpenCL as on the CPU. SkePU does not perform any translation between layouts, so it is the responsibility of the user to ensure that the layout matches.

1.18 Calling Library Functions; Whitelisting

Sometimes, it can be beneficial to call built-in/library functions from inside user functions. By default, SkePU assumes that all called functions are to be processed by the precompiler, which will prevent using library functions, because SkePU does not know in general whether these are available on every accelerator type supported and functionally equivalent to their CPU counterparts.

To avoid this issue, use the `-fnames` argument of the SkePU precompiler. This flag tells SkePU to ignore any function with this symbol name (all possible overloads), and it is up to the user to ensure that the compiler and linker for each backend can find suitable functions to call.

This feature is useful for whitelisting mathematical functions or `printf` debugging, but is best used very carefully, especially if accelerator backends are enabled. See the example usage below, as part of the skepu-tool invocation. Multiple function names are separated by whitespace.

```
skepu-tool  -fnames "sin␣cos"
```

1.19 Example programs

The SkePU distribution comes with a selection of example programs:

- **Cellular automaton** (`cellular_automaton`)
- **Cumulative moving average** (`cma`)
- **Conjugate gradient solver** (`conjugate_gradient`)
- **Coulumbic potential** (`coulombic`)
- **Dot product** (`dotproduct`)
- **Heat diffusion** (`heat_diffusion`)
- **Horner** (`horner`)
- **Image processing** (`image_manip`)
- **Mandelbrot fractal generator** (`mandelbrot`)
- **Maximum-minimum** (`maxmin`)
- **Miller-Rabin primality test** (`miller_rabin`)
- **Cellular automaton** (`mmmult`)
- **Monte-Carlo pi calculation** (`montecarlo`)
- **Matrix-vector multiplication, variant A** (`mvmult`)
- **Matrix-vector multiplication, variant B** (`mvmult_row`)
- **Matrix-vector multiplication, variant C** (`mmmult_row_col`)
- **N-body simulation, variant A** (`nbody`)
- **N-body simulation, variant B** (`nbody_mappairs`)
- **Pearson product-movement coefficient** (`ppmcc`)
- **Peak signal-to-noise ratio** (`psnr`)
- **Riemann sum** (`riemann_sum`)
- **Taylor series** (`taylor`)

1.20 More Information about SkePU

For further information about SkePU we refer to our publications.

SkePU 1 was introduced in 2010 and is presented in Enmyren and Kessler [7].

The integration of SkePU and StarPU to provide data-driven dynamic scheduling of asynchronous skeleton calls was presented by Dastgeer et al. [5].

The second generation of a back-end selection tuning framework for SkePU was developed by Dastgeer et al. [6]. Dastgeer also further developed the smart data-containers (Vector, Matrix) in [4]. Selection tuning and smart data-containers are still contained in today's SkePU implementation.

SkePU 1 supports sequential and multithreaded CPU execution as well as single- and multi-GPU execution in OpenMP and CUDA backends. These backends are, in modified form, still part of today's SkePU implementation. Experimental back-ends for SkePU 1 had also been developed for

plain MPI [18] and Movidius Myriad 2 [23], but were not mature enough to be included in the public distribution and were finally abandoned at the transition to SkePU-2.

Case studies on SkePU 1 include the porting of the EDGE flow simulation code [22], which revealed a number of weaknesses in the SkePU 1 API and finally led to the design of SkePU 2.

SkePU 2 was introduced in 2016. It involved the complete redesign of the SkePU programming interface based on C++11, and is described in Ernstsson et al. [15].

The generalization of smart containers for lazy execution of skeletons to provide global run-time optimizations such as tiling and kernel fusion across data-flow graphs of containers and skeletons was presented by Ernstsson and Kessler [13].

A new hybrid CPU-GPU back-end for SkePU 2 was proposed by Öhberg et al. [19] and is included in today's implementation.

The concept of multi-variant user functions and their implementation (included in SkePU-3) is presented in Ernstsson and Kessler [14].

Panagiotou et al. [20] present a case study of using SkePU in a brain modeling application, including first scaling results for the new SkePU 3 cluster backend based on StarPU-MPI [1], which is included in the distribution.

A SkePU tutorial, including most of the new SkePU 3 features, can be found on the SkePU web page [17].

The article [10] presents SkePU 3 (released 2020/2021) in its entirety. More details can be found in Ernstsson's PhD thesis [9].

The portable parallel pseudorandom number generator (SkePU-PRNG) with new syntax and standard library functionality added in SkePU 3.1 is described in [16].

A study of the performance portability of SkePU 3 has been presented by Ernstsson et al. [12].

An experimental FPGA backend for a SkePU subset (realized as a custom variant of the OpenCL backend) is described by Birath et al. [2].

Strided skeleton computations and the SkePU-DNN library for CNN/DNN training and inference atop SkePU has been presented by Qummar et al. [21].

The *SkeVU* interactive trace visualization tool for SkePU is described by Ernstsson et al. [11].

Chapter 2

SkeVU Visualization Tool

SkeVU [11] is an interactive visualization tool for execution traces of SkePU programs. It comes pre-installed along with SkePU and can be found in the visualizer folder.

SkeVU uses a web server model where the web server component is implemented in Python using Flask.

2.1 Setup

To setup the visualizer, it can be good to make a virtual environment due to Python dependencies.

1. (First time only) Create a virtual environment inside the visualizer directory.
   `cd visualizer/`
   `python -m venv venv/`
2. We can now activate the virtual environment:
   `source venv/bin/activate`
3. (First time only) Next, we need to install Flask to run the visualization server.
   `pip install -r requirements.txt`
4. Finally, to start the server, we simply run the following:
   `python visualizer.py`
5. Now we have our server up and running, and can access it on the web on the following localhost URL: `http://127.0.0.1:5001`. It is recommended to keep the server running in a dedicated terminal window.

2.2 Basic usage

To use the visualizer, we first need to enable tracing in the SkePU runtime. We can do so by adding the `-DSKEPU_TRACING` flag when compiling. This causes the program to generate a JSON file when it executes. Upload this file along with the source code for the program to the visualizer.

2.3 Setting and view options

TODO: Document the settings

2.4 Dependence graph view

TODO: Document the graph view, mainly the node types and edge types. Use text from the paper (CC license) once published.

Appendix A

Changes from SkePU 2 to SkePU 3

This appendix chapter summarizes the syntactical and behavioral changes from SkePU 2 to SkePU 3.

A.1 Changes in skeleton set and data-container set

The skeleton set has changed in SkePU 3, with the addition of the all-new `MapPairs` and `MapPairsReduce` skeletons, important extensions to the capabilities of the standard `Map` skeleton, and an interface change to improve usability of the `MapOverlap` skeleton. `MapPool` has been added for SkePU 3.3 in 2025.

The smart data-container set has also seen an extension in SkePU 3, by adding higher-dimensionality *tensors* in `Tensor3` and `Tensor4`. The coherency model of smart data-containers has been revised.

A.2 Namespace Change

The namespace for SkePU is changed in SkePU 3. Historically, the `skepu::` namespace was used by the initial SkePU release ("SkePU 1"), and since SkePU 2 was a major source-breaking change from SkePU 1, the decision was made to switch over the namespace as a way to communicate the source incompatibilities.

Today, there is to our knowledge little or no application code in active use which depends on SkePU 1. The decision was therefore made to switch back to the version-agnostic `skepu::` namespace for new releases of SkePU, starting with SkePU 3. This has the additional benefit of communicating that SkePU 3 also is a source-breaking transition from SkePU 2, although this time the scope of the changes is much smaller and transitioning between SkePU 2 to 3 is expected to be much simpler.

The intention is to keep this namespace for future versions of SkePU, and future source-breaking changes will be communicated in other ways.

The above namespace applies to skeletons and smart containers alike, as well as suporting constructs such as enums, backend specifications, container proxy objects, and index structs. In short, every C++ symbol in the library is affected. The exeption is C++11 attributes, whose names are forced to be namespaced but these namespaces are distinct from the standard C++ namespace definitions. This was introducted to SkePU in version 2 and have always been `skepu::`, (e.g. `[[skepu::out]]` and will remain as such.

In addition to the namespace change the default include header (the main entry point to the SkePU header library) has been changed. `#include <skepu2.hpp>` is replaced by `#include <skepu>` which aside from dropping the version now also mirrors the standard library with the absence of a file extension. (Note that the include directive may be different depending on the directory setup.)

A.3 Other Changes

The following major changes, described in more detail above, have been applied in the transition from SkePU-2 to SkePU-3 (as of May 2020):

- New skeletons `MapPairs`, `MapPairsReduce`;
- Revised interface for `MapOverlap` (e.g., `Region`X`D` proxy data-container object) and Reduce;
- New container types Tensor3, Tensor4 and new container proxy types MatRow, Ten3, Ten4;
- New backend selection mechanism, in both API and implementation, with settable global defaults and more options. Especially the OpenMP backend configuration options are expanded with new scheduling mechanisms.
- Deprecation of any STL-inherited dynamic features of the SkePU data-containers as a clarification that they are to be used as static objects;
- The SkePU 2 flush interface of smart data-containers was revised after feedback from EXA2PRO project partners: flush with options.
- Multi-valued return from skeletons and user functions;
- Multi-variant user functions;
- Dynamic scheduling option for all skeletons except Scan and Call;
- New memory consistency model for smart containers: weak consistency;
- New `skepu::external` construct to frame external I/O operations to/from data-containers or similar operations with side-effects that are not under the control of SkePU, providing sequential consistency for given data-container objects at the construct's boundaries;
- New StarPU-MPI backend (available for some skeletons at this time).

Several of these design changes are the result of feedback from application partners in the running H2020 FETHPC project EXA2PRO.

After the initial release of SkePU 3 in 2020/2021, new features have been added:

- `MapPool` skeleton and `Pool`X`D` data-container proxy object;
- Strided access in map-based skeletons;
- Extensions of the standard library, e.g., SkePU-DNN for deep learning training and inference with a PyTorch-like API implemented on top of SkePU skeletons;
- The SkeVU interactive visualization tool for SkePU program execution traces.

A.4 Repository and License Changes

With the release of SkePU 3 in 2020/2021, the public distribution of SkePU has moved to `https://github.com/skepu/skepu` and the SkePU web page has moved from Linköping University to `https://skepu.github.io`.

The SkePU license has been changed from GPLv3 for SkePU-2 to a less restrictive *modified 4-clause BSD license* for SkePU-3.

Appendix B

Acknowledgements

Work on SkePU was partly funded by EU H2020 project EXA2PRO, by the EU FP7 projects PEPPHER and EXCESS, by SeRC project OpCoReS, by ELLIIT project C05 GPAI, by SSF project FUS21-0033 ASTECC, by the Swedish national graduate school in computer science (CUGS), and by Linköping University.

We also thank the National Supercomputer Centre (NSC) and SNIC/NAISS for providing access to HPC resources used for performance testing (SNIC 2016/5-6, NAISS 2025/22-60, LiU-gpu-2023-05).

Appendix C

Bibliography

[1] Cédric Augonnet, Olivier Aumage, Nathalie Furmento, Raymond Namyst, and Samuel Thibault. StarPU-MPI: Task programming over clusters of machines enhanced with accelerators. In Jesper Larsson Träff, Siegfried Benkner, and Jack J. Dongarra, editors, *Recent Advances in the Message Passing Interface*, pages 298–299. Springer, 2012.

[2] Björn Birath, August Ernstsson, John Tinnerholm, and Christoph Kessler. High-level programming of fpga-accelerated systems with parallel patterns. *International Journal of Parallel Programming*, 52:253–273, 2024.

[3] Usman Dastgeer. *Performance-aware component composition for GPU-based systems*. PhD thesis, Linköping University, Sweden, 2014.

[4] Usman Dastgeer and Christoph Kessler. Smart containers and skeleton programming for GPU-based systems. *International Journal of Parallel Programming*, 44(3):506–530, 2016.

[5] Usman Dastgeer, Christoph Kessler, and Samuel Thibault. Flexible runtime support for efficient skeleton programming on hybrid systems. In *Proc. ParCo-2011 Int. Conference on Parallel Computing, Ghent, Belgium. In: Advances in Parallel Computing vol. 22*, pages 159–166. IOS press, 2012.

[6] Usman Dastgeer, Lu Li, and Christoph Kessler. Adaptive implementation selection in the SkePU skeleton programming library. In *Advanced Parallel Processing Technologies*, pages 170–183. Springer, 2013.

[7] Johan Enmyren and Christoph W Kessler. SkePU: A multi-backend skeleton programming library for multi-GPU systems. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*, pages 5–14. ACM, 2010.

[8] August Ernstsson. SkePU 2: Language embedding and compiler support for flexible and type-safe skeleton programming. Master's thesis, Linköping University, Linköping, Sweden, 2016. LIU-IDA/LITH-EX-A--16/026--SE.

[9] August Ernstsson. *Pattern-based Programming Abstractions for Heterogeneous Parallel Computing*. PhD thesis, Linköping University, Sweden, 2022.

[10] August Ernstsson, Johan Ahlqvist, Stavroula Zouzoula, and Christoph Kessler. SkePU 3: Portable high-level programming of heterogeneous systems and HPC clusters. *International Journal of Parallel Programming*, 49:846–866, May 2021. Open access: `https://link.springer.com/article/10.1007/s10766-021-00704-3`.

[11] August Ernstsson, Elin Frankell, and Christoph Kessler. Interactive performance visualization and analysis of execution traces for pattern-based parallel programming. *International Journal of Parallel Programming*, to appear, 2025.

[12] August Ernstsson, Dalvan Griebler, and Christoph Kessler. Assessing application efficiency and performance portability in single-source programming for heterogeneous parallel systems. *International Journal of Parallel Programming*, 51, January 2023.

[13] August Ernstsson and Christoph Kessler. Extending smart containers for data locality-aware skeleton programming. *Concurrency and Computation: Practice and Experience*, 31(5):e5003, 2019. e5003 cpe.5003.

[14] August Ernstsson and Christoph Kessler. Multi-variant user functions for platform-aware skeleton programming. In *Proc. of ParCo-2019 conference, Prague, Sep. 2019, in: I. Foster et al. (Eds.), Parallel Computing: Technology Trends, series: Advances in Parallel Computing, vol. 36, IOS press*, pages 475–484, March 2020.

[15] August Ernstsson, Lu Li, and Christoph Kessler. SkePU 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. *International Journal of Parallel Programming*, pages 1–19, 2017.

[16] August Ernstsson, Nicolas Vandenbergen, Jörg Keller, and Christoph Kessler. A deterministic portable parallel pseudo-random number generator for pattern-based programming of heterogeneous parallel systems. *Int. J. of Parallel Programming*, 50:319–340, August 2022.

[17] Christoph Kessler et al. SkePU: Autotunable multi-backend skeleton programming framework for multicore CPU and multi-GPU systems. `http://skepu.gitlab.io`.

[18] Mudassar Majeed, Usman Dastgeer, and Christoph Kessler. Cluster-SkePU: A multi-backend skeleton programming library for GPU clusters. In *Proc. Int. Conf. on Parallel and Distr. Processing Techniques and Applications (PDPTA-2013), Las Vegas, USA*, July 2013.

[19] Tomas Öhberg, August Ernstsson, and Christoph Kessler. Hybrid CPU-GPU execution support in the skeleton programming framework SkePU. *J. Supercomput.*, March 2019.

[20] Sotirios Panagiotou, August Ernstsson, Johan Ahlqvist, Lazaros Papadopoulos, Christoph Kessler, and Dimitrios Soudris. Portable exploitation of parallel and heterogeneous HPC architectures in neural simulation using SkePU. In *Proc. SCOPES'20*. ACM, May 2020.

[21] Sehrish Qummar, August Ernstsson, Christoph Kessler, and Oleg Sysoev. SkePU-DNN: Algorithmic skeleton programming for deep learning on heterogeneous systems. In *Proc. IEEE Int. Parallel and Distributed Processing Symposium Workshops (IPDPSW), Milano, Italy, June 2025*, pages 423–432. IEEE, 2025.

[22] Oskar Sjöström. Parallelizing the Edge application for GPU-based systems using the SkePU skeleton programming library. Master's thesis, Linköping University, Linköping, Sweden, 2015. LIU-IDA/LITH-EX-A--15/001--SE.

[23] Sebastian Thorarensen, Rosandra Cuello, Christoph Kessler, Lu Li, and Brendan Barry. Efficient execution of SkePU skeleton programs on the low-power multicore processor Myriad2. In *Proc. Euromicro PDP-2016 Int. Conf. on Parallel, Distributed, and Network-Based Processing, Heraklion, Greece*, pages 398–402. IEEE, February 2016.