SkePU-3 User Guide

August Ernstsson, Johan Ahlqvist, Christoph Kessler

May 21, 2020

Contents

1 SkePU User Guide **3**

1.1 Introduction . . . 4
1.2 License . . . 4
1.3 Authors and Maintainers . . . 4
1.3.1 Acknowledgements . . . 5
1.4 Dependencies and Requirements . . . 5
1.5 Example . . . 5
1.6 Installation . . . 7
1.6.1 Clang based skepu-tool . . . 7
1.7 Usage . . . 8
1.7.1 Clang skepu-tool and cmake . . . 9
1.7.2 StarPU MPI . . . 9
1.8 Limitations . . . 10
1.8.1 SkePU general . . . 10
1.8.2 StarPU skeleton backends . . . 10
1.8.3 Clang SkePU tool . . . 10
1.9 Definitions . . . 11
1.10 Skeletons . . . 12
1.10.1 Map . . . 14
1.10.2 Reduce . . . 15
1.10.3 MapReduce . . . 16
1.10.4 Scan . . . 17
1.10.5 MapOverlap . . . 17
1.10.6 MapPairs . . . 20
1.10.7 MapPairsReduce . . . 21
1.10.8 Call . . . 22
1.11 Multi-Valued Return from Skeletons and User Functions . . . 22
1.11.1 Multi-valued Return in User Functions . . . 22
1.12 Multi-Variant User Functions . . . 23

1.13 Manual Backend Selection and Default Settings 23
1.13.1 Backend Specification API Changes 23
1.14 Tuning of Skeleton Instances 24
1.15 Smart Containers 24
1.15.1 Smart Container Set 25
1.15.2 Smart Container Element Access 26
1.15.3 Matrix-row User Function Proxy Containers 27
1.16 Using Custom Types 28
1.17 Calling Library Functions; Whitelisting 29
1.18 SkePU 3 Changelog 29
1.18.1 Namespace Change 29
1.18.2 Other Changes Summary 30
1.19 More Information about SkePU 31

2 Bibliography **33**

Chapter 1

SkePU User Guide

Revision history

- 2019-09-25
  First Exa2Pro release.
- 2019-11-20
  Second Exa2Pro release. Revised installation instructions.
  Updated SkePU 3 changes.
- 2020-01-29
  Third Exa2Pro release. Updated to reflect changes in SkePU API and implementation progress.
- 2020-04-29
  Fourth Exa2Pro release. Updated with new skeleton MapPairsReduce and changes to backend selection process.
- 2020-05-20
  Separate document for public SkePU 3 user guide. Refactored changelog into the user guide proper.

1.1 Introduction

This chapter gives a high-level introduction to programming with SkePU. The version of SkePU documented here is a developement release of SkePU 3, a source-breaking update from SkePU 2. 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 hgh-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 licence. The copyright belongs to the individual contributors.

1.3 Authors and Maintainers

The original SkePU was created by Johan Enmyren and Christoph Kessler [5]. A number of people has contributed to SkePU, including Usman Dastgeer.

The major revision SkePU 2 was designed by August Ernstsson, Lu Li and Christoph Kessler [6].

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. [6].

August Ernstsson[1] is the current maintainer of SkePU.

[1] august.ernstsson@liu.se

1.3.1 Acknowledgements

This work was partly funded by EU H2020 project EXA2PRO, by the EU FP7 projects PEPPHER and EXCESS, by SeRC project OpCoReS, by the Swedish national graduate school in computer science (CUGS), and by Linköping University.

We also acknowledge the National Supercomputer Centre (NSC) and SNIC for providing access to HPC cluster systems used for performance testing (SNIC 2016/5-6).

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.

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 at least StarPU version 1.3.3 or later.

1.5 Example

We will introduce the SkePU syntax with an example.

Listing 1.1: Example SkePU 3 userfunction: 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);

        std::cout << "res:␣" << res << "\n";

        return 0;
}
```

1.6 Installation

In Section 1.6.1 we will look at building and installing the clang-based version of SkePU tool.

The installation and use process for SkePU is still in a prototype state.

1.6.1 Clang based 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 source code is available at the Exa2Pro repository at `https://gitlab.seis.iti.gr/exa2pro/skepu-clang`.git. Enter the cloned repository and fetch the submodules.

Building skepu-tool

New for skepu-tool is the move from a Makefile to a cmake based build procedure. The CMake scripts requires CMake version 3.10 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[2].

Please look at the included example programs and Makefiles 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`;
- include path to Clang's compiler-specific C++ headers, `-I <path_to_clang>/lib/Headers`, where the path is the root of the Clang sources (typically in the `tools` directory in the LLVM tree);

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

- 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 Clang skepu-tool and cmake

The clang skepu-tool offer a utility 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.

1.7.2 StarPU MPI

To use the StarPU MPI backend, precompile the source with only the OpenMP backend enabled. When compiling the precompiled source, use the compile flag `-DSKEPU_MPI_STARPU` to enable the StarPU MPI 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. Writing to file is one example where multiple nodes cannot execute the same region at the same time.

1.8 Limitations

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

1.8.1 SkePU general

Known issues with the SkePU headers:

- MapOverlap (all) code generation is disabled for OpenCL (Section 1.10.5).
- MapOverlap (3D, 4D) code generation is disabled for CUDA (Section 1.10.5).
- MapPairsReduce code generation is disabled for CUDA and OpenCL (Section 1.10.7).
- Multi-valued return (Section 1.11.1) is not implemted 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 skeleton backends

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

- The Tensor4 container is missing.
- The containers might not be fully API compatible with the normal version of SkePU.
- The skeletons Call, MapOverlap, MapPair, MapPairReduce, MapReduce, and Scan are missing.

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 computation structure on *container*s, e.g. map or reduce. The skeletons in SkePU 2 are all data-parallel, i.e., the computation graph is directed by the structure of container parameters and not dependent on the value of individual elements in a container.

Container An object of some SkePU container class, i.e., vector or matrix. Homogenous; contains objects of a single *scalar* type. In this document, the term container refers exclusively to SkePU containers (as opposed to, e.g., raw data pointers or STL vectors).

Scalar The type of elements in a *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 refered 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 exeption of writing to *random access arguments*.

Skeleton instance An object of some skeleton type instantiated with one or more *user functions*. May include state such as

- *backend specification*,
- *execution plan*, and
- *skeleton*-specific parameters such as the starting value for a reduction.

Skeleton invocation The process of applying a *skeleton instace* 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 instace*, 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, with

Random access parameter/argument A *container* argument to a *skeleton instace*, which, during *skeleton invocation*, is passed to the corresponding *user function* parameter and av.

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 Skeletons

SkePU (3) encompasses currently eight different skeletons:

- `Map`,

- `Reduce`,
- `MapReduce`,
- `Scan`,
- `MapOverlap`,
- `MapPairs`,
- `MapPairsReduce`, and
- `Call`.

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 behaviours for one- and two-dimensional computations.

Skeletons in SkePU are instantated 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[3].

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.

`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`.

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

Listing 1.2: Example usage of the Map skeleton.

```
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);
}
```

`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 dereferenced by the caller anymore.

1.10.1 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.

Example

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.

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

// OpenMP
spec.setBackend(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.setBackend(skepu::Backend::Type::CUDA);
spec.setBackend(skepu::Backend::Type::OpenCL);
spec.setDevices(<int>); // number of GPUs
spec.setGPUThreads(/*int*/);
spec.setGPUBlocks(/*int*/);

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

1.10.2 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

Listing 1.3: Example usage of the Reduce skeleton.

```
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);
}
```

types.

Example

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 behavoir up to a set container size, `-DSKEPU_REDUCE2DCOL_TRANSPOSE_SIZE_MAX [n]`.

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

1.10.3 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.

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 start value for reduction. Defaults to a default-constructed object, which is `0` for built-in numeric types.

Listing 1.4: 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);
}
```

Example

1.10.4 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.

Example

1.10.5 MapOverlap

`MapOverlap` is a stencil operation. It is similar to Map, but instead of a single element, a region of elements is available in the user function. The region is passed as a pointer, so manual pointer arithmetic is required to access the data. The pointer points to the center element.

A MapOverlap can either be one-dimensional, working on vectors or matrices (separable computations only) or two-dimensional for matrices. The type is set per-instance and deduced from the user function.

Listing 1.5: Example usage of the Scan skeleton.

```
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);
}
```

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(`*`x_radius, y_radius`*`)` For 2D `MapOverlap` only. Sets the overlap for x and y directions.

`instance.getOverlap()` Returns the overlap radius: a single value for 1D `MapOverlap`, a `std::pair` (x, y) for 2D `MapOverlap`.

`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.

Major Revision to MapOverlap Skeleton

To improve upon the usability of `MapOverlap`, and also to better generalize the syntax to higher-dimensionality smart containiners (see the addition of tensors in SkePU 3), the interface of MapOverlap user functions has changed.

Before, the signature of a MapOverlap user function was very deliberate with explicit parameters for overlap size, and the smart container data was passed by a raw pointer. This also meant that the adressing into the overlap region was left to the user, and this could be quite tricky to get right.

In SkePU 3, this is now changed by the addition of a new set of smart container proxy classes. Similar to existing `Vec`, `Mat`, and the new `MatRow` (see Section 1.15.3), the `RegionND` (where $N = 1, 2, 3, 4$) classes are proxy classes representing an overlap region from a smart container of dimensionality N.

The region object contains members for accessing the overlap size, `region.on` where $n = i, j, k, l$.

A region object allows element access by using paranthesis notation, with one argument for each dimension. **Note:** the element at position $(0, ..., 0)$ is the center element, and the notation allows access in the range $[-overlap, +overlap]$ (inclusive) for each dimension.

Below are some examples of how to use the new user function syntax. Note that the parameter list after the Section parameter is flexible like before, with any number of random-access and uniform arguments allowed.

```
float over_1d(skepu::Region1D<float> r, int scale)
{
        return (r(-2)*4 + r(-1)*2 + r(0) + r(1)*2 + r(2)*4) / scale;
}
```

```
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;
}
```

```
float over_3d(skepu::Region3D<float> r)
{
```

```
        float res = 0;
        for (int i = -r.oi; i <= r.oi; ++i)
                for (int j = -r.oj; j <= r.oj; ++j)
                        for (int k = -r.ok; k <= r.ok; ++k)
                                res += r(i, j, k);
        return res;
}
```

```
float over_4d(skepu::Region4D<float> r, skepu::Ten4<float> stencil)
{
        float res = 0;
        for (int i = -r.oi; i <= r.oi; ++i)
                for (int j = -r.oj; j <= r.oj; ++j)
                        for (int k = -r.ok; k <= r.ok; ++k)
                                for (int l = -r.ol; l <= r.ol; ++l)
                                        res += r(i, j, k, l) * stencil
                                            (i + r.oi, j + r.oj, k + r
                                            .ok, l + r.ol);
        return res;
}
```

1.10.6 MapPairs

SkePU 3 adds an additional top-level skeleton, **MapPairs**. This skeleton applies a cartesian product-style pattern from *two* `Vector<T>` sets (note that the templated type may differ across these vectors).

Each cartesian combination of vector set indicies generates one userfunction 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.

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.

`MapPairs` is only defined for Vector-Matrix application, there is no analog for Matrix-Tensor4 at this point.

Below is a simple example of `MapPairs` in use.

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

void test1(size_t Vsize, size_t Hsize, skepu::BackendSpec spec)
{
        auto pairs = skepu::MapPairs(uf);
        pairs.setBackend(spec);

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

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 (4-way variadic compared to Map being 3-way):

- Elementwise-V args
- Elementwise-H args
- Random-access args
- Uniform args

A more involved skeleton instance might look like this: `auto pairs = skepu::MapPairs<3, 2>(uf2);` Note that the skeleton is templated in number of containers for each set ("vertical" and "horizontal" dimensions). A compatible user function can be seen below:

```
int uf2(
        skepu::Index2D i,
        int ve1, int ve2, int ve3,
        int he1, int he2,
        skepu::Vec<int> test,
        int u1, int u2
)
{
        // some computation involving the above...
        return i.row + i.col + u1 + ve1 + ve2 + ve3
         + he1 + he2 + test.data[0] + u1 + u2;
}
```

1.10.7 MapPairsReduce

SkePU 3 also adds `MapPairsReduce`, analogous to `MapReduce` for `MapPairs`.

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.

`MapPairsReduce` supports two reduce modes, both reduce-to-vector: row-wise and col-wise. See the code example below.

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

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


{
        auto mpr = skepu::MapPairsReduce(uf, sum);

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

        mpr.setReduceMode(skepu::ReduceMode::ColWise);
        mpr(resH, v1, h1);

        mpr.setReduceMode(skepu::ReduceMode::RowWise);
        pairs(resV, v1, h1);
}
```

1.10.8 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.

1.11 Multi-Valued Return from Skeletons and User Functions

1.11.1 Multi-valued Return in 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, ...])`.

Below is an example of a user function utilizing this:

```
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);
}
```

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

```
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 `Map` skeleton.

1.12 Multi-Variant User Functions

TBD ...

See our ParCo'19 paper [8] or EXA2PRO Deliverable D2.2 (Oct. 2019) for details.

1.13 Manual Backend Selection and Default Settings

1.13.1 Backend Specification API Changes

SkePU 3 changes the backend selection mechanism, in both API and implementation. Especially the OpenMP parameters are expanded with new scheduling mechanisms.

Global Backend Specification

In SkePU 3 it is now possible to set a global backend specification. This specification will be used by default for all skeleton instances. Overriding can be done on an instance basis using a member function, see example.

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

skepu::setGlobalBackendSpec(spec);

skepu::restoreDefaultGlobalBackendSpec();

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

1.14 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 which is used as a look-up table during *skeleton invocation*. It is also possible to construct such a plan manually, and assign it to

1.15 Smart Containers

The smart containers available in SkePU are `Vector` and `Matrix`. Using these is mostly transparent, as they will optimize memory management and data movement dynamically between CPU and GPUs.

There is also manual interface for data movement: `container.updateHost()` will force download of up-to-date data from the GPUs, and `container.invalidateDeviceData()` forces a re-upload for the next skeleton invocation on a GPU.

Element access on the CPU can be done either with `operator[`*`index`*`]`, which includes overhead for checking remove copies, or `operator(`*`index`*`)` which provides direct, no-overhead access.

When smart containers are used as *element-wise parameters* to user functions, it is important to note that separate types are used, `Vec` and

`Mat`. These proxy types do not provide the full smart container functionality and are used with a C-style interface. Elements are retrieved using `container.data[index]` member, and `size`, `rows`, and `cols` 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 `const` qualifier, the copy-down is emliminated. Similarily, a `[[skepu::out]]` attribute will turn them into output parameters.

1.15.1 Smart Container Set

The SkePU container set is extended with tensors, which are higher-dimensionality containers. In SkePU 3 there are tensors of three and four dimensions, complementing the existing 1D "vector" and 2D "matrix". Smart container dimensionality in SkePU 3 is therefore static, though their sizes in each dimension is dynamic.

The interface for these containers are virtually indentical to those of the other containers, differing in the obvious ways of naming and element access detailed below. They are defined as follows:

```
template<typename T> Tensor3 { ... };
template<typename T> Tensor4 { ... };
```

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

Compare with existing containers:

```
skepu::Vector<float> v(dim1);
skepu::Matrix<float> m(dim1, dim2);
```

The Index object set in SkePU, useable in e.g. user function signatures, is extended with these structs:

```
struct Index3D { size_t i, j, k; };
struct Index4D { size_t i, j, k, l; };
```

This complements the existing structs. Note that the naming convention is different for matrix indices for compatibility reasons.

```
struct Index1D { size_t i; };
struct Index2D { size_t row, col; }; // note!
```

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 inmost dimension.

MapReduce See Map. For the reduce step, over full domain or the innmost dimension.

Scan Over full domain.

MapOverlap Overlap radius limited to 1 in each dim (default) Larger overlap dependent on backend support.

Call See Map

1.15.2 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 1.15.2.

Instead of angle brackets, the parantheses `( )`-notation is extended to higher dimensionality. This syntax accepts one index argument for each dimension of the underlying container. The indicies 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's no interface for 1D indexing of higher-dimensionality containers.

There is no longer a coherency-satisying 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

flush interface revised from feedback from partners flush with options

`enum class FlushMode  Default, Dealloc ;`

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

Container member function flush as well as variadic free template function flush

Member function: dynamic flush mode, can be selected at runtime

Free function: flush mode is static constant, 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.

1.15.3 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 are 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.

1.16 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.

1.17 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"
```

1.18 SkePU 3 Changelog

This section goes through all the syntactical and behavioral changes from SkePU 2 to SkePU 3.

In particular, 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.

The smart container set has also seen an extension in SkePU 3, the framework now has higher-dimensionality *tensors* in `Tensor3` and `Tensor4`. The coherency model of smart containers has also seen an update.

1.18.1 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.)

1.18.2 Other Changes Summary

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 and Reduce;
- New container types Tensor3, Tensor4 and new container proxy types MatRow, Ten3, Ten4;
- Deprecation of any STL-inherited dynamic features of the SkePU containers as a clarification that they are to be used as static objects;
- 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 MPI backend (available for some skeletons at this time), along with a new construct (interface still pending) to frame external I/O operations.

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

Moreover, the public distribution of SkePU has moved to github.com/skepu/skepu and the SkePU web page has moved from Linköping University to 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.

1.19 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 [5].

The integration of SkePU and StarPU to provide data-driven dynamic scheduling of asynchronous skeleton calls was presented by Dastgeer et al. [3].

The second generation of a back-end selection tuning framework for SkePU was developed by Dastgeer et al. [4]. Dastgeer also further developed the smart data-containers (Vector, Matrix) in [2]. Selection tuning and smart 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 [11] and Movidius Myriad 2 [15], 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 [14], 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. [9].

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 [7].

A new hybrid CPU-GPU back-end for SkePU 2 was proposed by Öhberg et al. [12] 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 [8].

Panagiotou et al. [13] 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 [10].

A paper presenting SkePU 3 in its entirety is currently in preparation.

Chapter 2

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

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

[13] 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, to appear*. ACM, May 2020.

[14] 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.

[15] 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.