SkePU 2 User Guide

For the preview release

August Ernstsson

October 20, 2016

Contents

Revision history

- **0.1:** 2016-10-20
  Initial version for the preview release.

1 Introduction

This document gives a high-level introduction to programming with SkePU 2[1]. 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 2 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 2 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 `skepu2` namespace. Nested namespaces are not part of the API and should be considered implementation-specific. The `skepu2::` qualifier is implicit for all symbols in this document.

2 License

SkePU 2 is distributed as open source and licensed under GPL version 3. The copyright belongs to the individual contributors.

3 Authors and Maintainers

The original SkePU was created by Johan Enmyren and Christoph Kessler [1]. 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 [2].

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

3.1 Acknowledgements

This work was partly funded by the EU FP7 projects PEPPHER and EXCESS, and by SeRC.

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 7.5.

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

[1]In this document, all mentions of SkePU implicitly refers only to SkePU 2
[2]august.ernstsson@liu.se

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").

5 Example

We will introduce the SkePU 2 syntax with an example.

Listing 1: Example SkePU 2 userfunction: A linear congruential generator.

```
#include <iostream>
#include <cmath>
#include <skepu2.hpp>

// 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(skepu2::Vector<float> &x, skepu2::Vector<float> &y)
{
        // Instance of Reduce skeleton
        auto sum = skepu2::Reduce(plus<float>);

        // Instance of MapReduce skeleton
        auto sumSquare = skepu2::MapReduce<1>(square, plus<float>);

        // Instance with lambda syntax
        auto dotProduct = skepu2::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
        skepu2::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;
}
```

6 Installation and Usage

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

6.1 Installation and Set-up

Follwing are the steps required to set up the SkePU source-to-source compiler.

- Clone LLVM and Clang Git repositories and confirm that Clang builds.
- Patch Clang (see below) to add SkePU attributes and diagnostics to Clang. It also instructs the Clang build system to build the SkePU tool.
- Create a directory symlink in '¡clang source¿/tools' named 'skepu' pointing to 'clang_precompiler'. Clang will now find the source files for the SkePU tool.
- Run 'ninja skepu' in your Clang build directory.
- If successful, the SkePU precompiler binary should now be in '¡clang build¿/bin/skepu'.
- Run 'skepu -help' to confirm that everything worked and to see available options.

6.1.1 Cloning LLVM and Clang

```
mkdir ~/clang-llvm && cd ~/clang-llvm

git clone http://llvm.org/git/llvm.git

cd llvm/tools

git clone http://llvm.org/git/clang.git
```

It is recommended to check out specific commits (from May 2016) from the repositories since the patch below is not tested on other states. Run the commands at the root of the respective repository.

LLVM:

```
git checkout d3d1bf00 .
```

Clang:

```
git checkout 37b415dd .
```

6.1.2 Patch Clang

```
cd <clang source>

git apply <skepu source>/clang_patch.patch
```

6.1.3 Symlink SkePU sources in Clang

```
ln -s <skepu source>/clang_precompiler <clang source>/tools/skepu-tool
```

6.1.4 Building SkePU precompiler

Create and/or move to build directory.

```
cd <clang build>
```

6.1.5 Set up CMake

```
cmake -G "Unix Makefiles" <llvm source> -DCMAKE_BUILD_TYPE=Release
```

(Release mode is faster and much more space efficient, remove option to build in Debug mode.)

6.1.6 Build the SkePU tool

```
make skepu-tool
```

6.2 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`.

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 `skepu2.hpp`, which contains all of the SkePU library[3].

Please look at the included example programs and Makefiles to get an idea of how everything works in practice.

6.2.1 Include directories

`skepu-tool` is based on 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);
- 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.

6.2.2 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.)

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

7 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.

8 Skeletons

SkePU encompasses six different skeletons:

- `Map`,
- `Reduce`,
- `MapReduce`,
- `Scan`,
- `MapOverlap`, 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[4].

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`.

`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.

8.1 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.

Listing 2: Example usage of the Map skeleton.

```cpp
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);
}
```

Listing 3: Example usage of the Reduce skeleton.

```cpp
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);
}
```

Example

8.2 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.

Listing 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

8.3 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.

Example

8.4 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.

Listing 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);
}
```

Example

8.5 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.

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.

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

`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.

8.6 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.

More on Call coming soon...

9 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

10 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 `[[skepu2::out]]` attribute will turn them into output parameters.

11 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.

References

[1] 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.

[2] August Ernstsson, Lu Li, and Christoph Kessler. SkePU 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. Accepted for HLPP-2016, Münster, Germany, 4–5 July 2016.