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

SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems

August Ernstsson[1] · **Lu Li**[1] · **Christoph Kessler**[1]

Received: 30 September 2016 / Accepted: 10 January 2017 / Published online: 28 January 2017

Abstract In this article we present *SkePU 2*, the next generation of the SkePU C++ skeleton programming framework for heterogeneous parallel systems. We critically examine the design and limitations of the SkePU 1 programming interface. We present a new, flexible and type-safe, interface for skeleton programming in SkePU 2, and a source-to-source transformation tool which knows about SkePU 2 constructs such as skeletons and user functions. We demonstrate how the source-to-source compiler transforms programs to enable efficient execution on parallel heterogeneous systems. We show how SkePU 2 enables new use-cases and applications by increasing the flexibility from SkePU 1, and how programming errors can be caught earlier and easier thanks to improved type safety. We propose a new skeleton, *Call*, unique in the sense that it does not impose any predefined skeleton structure and can encapsulate arbitrary user-defined multi-backend computations. We also discuss how the source-to-source compiler can enable a new optimization opportunity by selecting among multiple user function specializations when building a parallel program. Finally, we show that the performance of our prototype SkePU 2 implementation closely matches that of SkePU 1.

Keywords Skeleton programming · SkePU · Source-to-source transformation · C++11 · Heterogeneous parallel systems · Portability

✉ August Ernstsson
August.Ernstsson@liu.se

[1] PELAB, Department of Computer and Information Science, Linköping University, Linköping, Sweden

1 Introduction

The continued trend in computer engineering towards heterogeneity and parallelism, whether in the form of increased processor count or the addition of peripheral accelerators, are increasing the demand for accessible, high-level parallel and heterogeneous programming models. The ability to adapt to these systems is increasingly important for all programmers, not just domain experts. As the processor cores, interconnects, memory architectures, and low-level programming interfaces are all very diverse, such a programming model should also adapt the produced executables to the host platform, providing a best-effort utilization of the target architecture in terms of performance and energy efficiency.

To some extent, *algorithmic skeletons* [3] provide a solution to the above problems. Skeletons are generic, high-level programming constructs which hide the platform-specific implementation details and present a clean, parametrizable interface to the user. SkePU [9] is a high-level C++ skeleton programming framework for heterogeneous parallel systems, with a primary focus on multi-core and multi-GPU architectures. Programs targeting SkePU are portable between systems both in terms of source compatibility and performance utilization. It also provides a more programmer-friendly interface than standardized low-level APIs such as OpenCL.

SkePU is six years old, and since its conception there has been significant advancements in this field in general and to C++ in particular. C++11 introduces many new features, among them variadic templates and a standardized attribute syntax.

This article presents the next generation of SkePU, as first proposed by Ernstsson in his master's thesis [11]. Following are the main contributions of our work:

- We introduce a new skeleton *programming interface* for SkePU programs centered on C++11 and variadic templates. A sequential reference implementation is available along with this interface.
- We present the *Call* skeleton, along with other functionality and flexibility improvements.
- We discuss the ramifications the new interface has on *type-safety* compared to previous versions of SkePU.
- We introduce a *source-to-source translation* tool based on Clang libraries which transform code written for the new interface to code targeting OpenMP, OpenCL and CUDA backends.
- We evaluate the readability of SkePU 2 programs with a survey.
- Performance of SkePU 2 is compared with SkePU 1.2, using a collection of example applications. We show that it is possible to preserve or improve on the performance of SkePU 1.2 while having a cleaner and more modern interface.

Section 2 starts by introducing SkePU 1 in more detail, including a critical review of the framework focusing on clarity, flexibility, and type-safety. Section 3 introduces SkePU 2, with the underlying design principles and its interface; Sect. 4 details a prototype implementation. Results from a readability survey are covered in Sect. 5, followed by early performance results in Sect. 6. Section 7 discusses related work. Finally, Sect. 8 concludes the article and presents ideas for future work.

2 SkePU

This section reviews SkePU 1,[1] based on its most recent version, SkePU 1.2, released in 2015. SkePU [9] is a C++ template library for high-level performance-portable parallel programming, based on *algorithmic skeletons* [3]. SkePU has previously been demonstrated to provide an approachable interface [9], an auto-tuning mechanism for backend selection [7] and efficient smart containers [5]. Several industrial-class applications have been ported to SkePU, for example the computational fluid dynamics flow solver *EDGE* [20].

2.1 Skeletons, user Functions, and Smart Containers

SkePU includes the following skeletons:

- **Map**, generic unary, binary, or ternary element-wise data-parallel operation on aggregate data (i.e., vectors and matrices);
- **Reduce**, generic reduction operation with a binary associative operator;
- **MapReduce**, unary, binary, or ternary map followed by reduce;
- **Scan**, generalized prefix sum operation with a binary associative operator;
- **MapArray**, unary Map operation with an extra container argument available in its entirety;
- **MapOverlap**, stencil operation in one or two dimensions with various boundary handling schemes;
- **Generate**, initializes elements of an aggregate data structure based on the element index and a shared initial value.

Most skeletons also take an optional uniform argument, passed unaltered with each user function call. To have multiple such arguments, they must first be wrapped in a `struct`. A user-defined `struct` type may also be used to instantiate the smart container templates, but such types will need explicit redefinition for some backends (e.g., OpenCL).

Listing 1: Vector sum computation in SkePU 1.

```
BINARY_FUNC(add, float, a, b,
  return a + b;
)

skepu::Vector<float> v1(N), v2(N), res(N);

skepu::Map<add> vec_sum(new add);
vec_sum(v1, v2, res);
```

User functions are operators used to instantiate and initialize a skeleton instance. SkePU 1 uses a preprocessor macro language to express and generate multi-platform user functions. For an example, see the vector sum computation in Listing 1.

SkePU includes aggregate data structures in the form of *smart containers* [5]. Available as vectors and matrices with generic element types, smart containers are run-time

[1] http://www.ida.liu.se/labs/pelab/skepu/

data structures that reside in main memory, but can temporarily store subsets of their elements in GPU device memories for access by skeleton backends executing on these devices. Smart containers additionally perform transparent software caching of the operand elements that they wrap, with a MSI-based coherence protocol. Hence, smart containers automatically keep track of valid copies of their element data and their locations, and can, at run-time, automatically optimize communication and device memory allocation. Smart containers can lead to a significant performance gain over ”normal”? containers, especially for iterative computations (such as N-body simulation) on sufficiently large data, where data can stay on the GPU device.

2.2 Limitations

SkePU was conceived and designed six years ago, with the goal of portability across very diverse programming models and toolchains such as CUDA and OpenMP; since then, we have seen significant advancements in this field in general and to C++ in particular. C++11 provides fundamental performance improvements, e.g., by the addition of move semantics, `constexpr` values, and standard library improvements. It introduces new high-level constructs: range-for loops, lambda expressions, and type inference among others. C++11 also expands its metaprogramming capabilities by introducing variadic template parameters and the aforementioned `constexpr` feature. Finally (for the purpose of this paper), the new language offers a standardized notation for attributes used for language extension. The proposal for this feature explicitly discusses parallelism as a possible use case [17], and it has been successfully used in, for example, the REPARA project [4]. Even though C++11 was standardized in 2011, it is only in recent years that compiler support is getting widespread, see, e.g., Nvidia’s CUDA compiler.

For this project, we specifically targeted the following limitations of SkePU.

Type safety Macros are not type-safe and SkePU 1 does not try to work around this fact. In some cases, errors which semantically belong in the type system will not be detected until run-time. For example, SkePU 1 does not match user function type signatures to skeletons statically, see Listing 6. This lack of type safety is unexpected by C++ programmers.

Flexibility A SkePU user can only define user functions whose signature matches one of the available macros. This has resulted in a steady increase of user function macros in the library: new ones have been added ad-hoc as increasingly complex applications has been implemented on top of SkePU. Some additions have also required more fundamental modifications of the runtime system. For example, when a larger number of auxiliary containers was needed in the context of MapArray, an entirely new *MultiVector* container type [20] had to be defined, with limited smart container features. Real-world applications need more of this kind of flexibility.

An inherent limitation of all skeleton systems is the restriction of the programmer to express a computation with the given set of predefined skeletons. Where these do not fit naturally, performance will suffer. It should rather be possible for programmers to add their own multi-backend components [8] that could be used together with SkePU skeletons and containers in the same program and reuse SkePU’s auto-tuning mechanism for backend selection.

Optimization opportunity Using the C preprocessor for code transformation drastically limits the possible specializations and optimizations which can be performed on user functions, compared to, e.g., template metaprogramming or a separate source-to-source compiler. A more sophisticated tool could, for example, choose between separate *specializations* of user functions, each one optimized for a different target architecture. A simple example is a user function specialization of a vector sum operation for a system with support for SIMD vector instructions.

Implementation verbosity SkePU skeletons are available in multiple different modes and configurations. To a large extent, the variants are implemented separately from each other with only small code differences. Using the increased template and metaprogramming functionality in C++11, a number of these can be combined into a single implementation without loss of (run-time) performance.

3 SkePU 2 Design Principles

We have created a new version of SkePU to overcome the limitations of the original design. SkePU 2 builds on the mature runtime system of SkePU 1: highly optimized skeleton algorithms for each supported backend target, smart containers, multi-GPU support, etc. These are preserved and have been updated for the C++11 standard. This is of particular value for the Map and MapReduce skeletons, which in SkePU 1 are implemented thrice for unary, binary and ternary variants; in SkePU 2 a single variadic template variant covers all N-ary type combinations. There are similar improvements to the implementation wherever code clarity can be improved and verbosity reduced with no run-time performance cost.

The main changes in SkePU 2 are related to the programming interface and code transformation. SkePU 1 uses preprocessor macros to transform user functions for parallel backends; SkePU 2 instead utilizes a source-to-source translator (precompiler), a separate program based on libraries from the open source Clang project.[2] Source code is passed through this tool before normal compilation. However, a SkePU 2 program is valid C++11 as-is; a sequential binary (with identical semantics to the parallel one) will be built if the code is compiled directly by a standard C++ compiler.

Listing 2: Vector sum computation in SkePU 2.

```
template<typename T>
T add(T a, T b) {
  return a + b;
}

skepu2::Vector<float> v1(N), v2(N), res(N);

auto vec_sum = skepu2::Map<2>(add<float>);
vec_sum(res, v1, v2);
```

This section introduces the new programming interface, syntax and other features of SkePU 2, first by means of an example. Listing 2 contains a vector sum computation in SkePU 2 syntax, mirroring Listing 1.

[2] http://clang.llvm.org.

Table 1 SkePU 2 skeletons and their features and attributes

	Map	Reduce	Scan	MapReduce	MapOverlap	Call
Elwise vector	•	•	•	•	With overlap	
Elwise matrix	•	•	•	•	With overlap	
Elwise arity	$N \geq 0$	1	1	$N > 0$	1	0
Extra containers	•			•	•	•
Uniform arguments	•			•	•	•
Indexing	•			•		

3.1 Skeletons

A skeleton is invoked with the overloaded `operator()`, with arguments matching those of the user function. Additionally, the output container is (where applicable) passed as the first argument. Smart containers may be passed either by reference or by iterator, the latter allowing operations on partial vectors or matrices. A particular *argument grouping* is required by SkePU 2: all element-wise containers must be grouped first, followed by all random-access containers, and uniform arguments last.

There are six skeletons available in SkePU 2: *Map*, *Reduce*, *MapReduce*, *Scan*, *MapOverlap*, and *Call*; this is fewer than in SkePU 1, as the generalized Map now covers the use-cases of MapArray and Generate. See Table 1 for a list of the skeletons.

Skeletons *instances* are declared with an inferred type (using the `auto` specifier) and defined by assignment from a factory function, as exemplified in Listing 2. The actual type of a skeleton instance should be regarded as unknown to the programmer.

Map is greatly expanded compared to SkePU 1. A Map skeleton accepts N containers for any integer N including 0. These containers must be of equal size, as do the return container. As one element from each of these containers will be passed as arguments to a call to a user function, we refer to these containers as *element-wise* arguments. Map additionally takes any number of SkePU containers which are accessible in their *entirety* inside a user function called *random access* arguments thus rendering MapArray from SkePU 1 redundant. These parameters are declared to be either *in* (by `const` qualification), *out* (with a C++11 attribute), or *inout* (default) arguments and only copied (e.g., between main memory and device memory) when necessary. Finally, scalar arguments can also be included, passed unaltered to the user function. The Map skeleton is thus three-way variadic, as each group of arguments is handled differently and is of arbitrary size.

Another feature of Map 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 two `structs`: `Index1D` for vectors and `Index2D` for matrices. This 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.

Reduce is a generic reduction operation with an associative operator available in multiple variants. A vector is reduced in only one way, but for matrices five options

exist. A reduction on a matrix may be performed in either one or two dimensions (for two-dimensional reduction the user supplies two user functions), both either row-wise or column-wise. The fifth mode treats the matrix as a vector (in row-major order) and is the only mode available if an iterator into a matrix is supplied.

MapReduce is a combination of Map and Reduce and offers the features of both, with the limitation that the element-wise arity must be at least 1.

Scan implements two variants of the prefix sum operation generalized to any associative binary operator. The variants are inclusive or exclusive scan, where the latter supports a user-defined starting value.

MapOverlap is a one or two-dimensional stencil operation. Parameters for specializing the boundary handling are available, and there is specific support for separable 2D stencils.

Listing 3: Example usage of the Call skeleton.

```
void swap_f(int *a, int *b)
{
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

void sort_f(skepu2::Vec<int> arr, size_t nn)
{
#if SKEPU_USING_BACKEND_CL

  // Even-odd sort
  // Multiple invocations in parallel
  size_t idx = get_global_id(0);
  size_t l = nn / 2 + ((nn % 2 != 0) ? 1 : 0);

  for (size_t i = 0; i < l; ++i)
  {
    if (idx % 2 == 0 && idx < nn - 1 && arr.data[idx] > arr.data[idx + 1])
      swap_f(&arr.data[idx], &arr.data[idx + 1]);
    barrier(CLK_GLOBAL_MEM_FENCE);

    if (idx % 2 == 1 && idx < nn - 1 && arr.data[idx] > arr.data[idx + 1])
      swap_f(&arr.data[idx], &arr.data[idx + 1]);
    barrier(CLK_LOCAL_MEM_FENCE);
  }

#else // SKEPU_USING_BACKEND_CPU

  // Insertion sort
  // A single, sequential invocation
  for (size_t c = 1; c <= nn - 1; c++)
    for (size_t d = c; d > 0 && arr.data[d] < arr.data[d-1]; --d)
      swap_f(&arr.data[d], &arr.data[d - 1]);

#endif
}


void sort(skepu2::Vector<int> &v, skepu2::BackendSpec spec)
{
  auto sort = skepu2::Call(sort_f);

  spec.setGPUBlocks(1);
  spec.setGPUThreads(v.size());
  sort.setBackend(spec);

  sort(v, v.size());
}
```

Call is a completely new skeleton for SkePU 2. It is not a skeleton in a strict sense, as it does not enforce a specific structure for computations. Call simply invokes its user function. The programmer can provide arbitrary computations as explicit user function backend specializations, which must include at least a sequential general-purpose CPU backend as a default variant. The direction (in, out, inout) of parameter data flow follows the same principles as for the Map skeleton described above. Call provides seamless integration with SkePU features such as smart containers and auto-tuning of back-end selection. Basically, Call extends the traditional skeleton programming model in SkePU with the functionality of user-defined *multi-variant components* (i.e., "PEPPHER" components [8]) with auto-tunable automated variant selection.

Listing 3 contains an example application of the Call skeleton, integer sorting, which can otherwise be difficult to implement in data-parallel skeleton programming. One of two distinctly different algorithms are selected depending on whether the Call instance is executed on CPU or GPU. (Note that the example is just an illustration; the CPU insertion sort algorithm is inefficient, and the even-odd sorting in the GPU variant works only inside a single work group. Also, the syntax for specializing user functions will be refined in the future.)

3.2 User Functions

Listing 4: Skeleton instance with lambda syntax for the user function.

```
1 auto vsum = Map<2>([](float a, float b) { return a + b; });
```

In the example in Listing 2, the user function is defined as a free function template. This is one of two ways to define user functions in SkePU 2; the other is with lambda expression syntax as in Listing 4, where the function is written inline with the skeleton instance. Free functions are reminiscent of the macros used in SkePU 1, and still suitable for cases where a user function can be shared across skeleton instances. In most cases, however, the lambda syntax is superior; it increases code locality while eliminating namespace pollution. There are no run-time differences between the two, as identical code is generated by the precompiler.

Naturally, the source-to-source translator is limited in scope when transforming user functions for parallel execution. Operations with side effects, for example memory allocation or I/O operations, have undefined behavior inside user functions unless explicitly allowed by SkePU 2. Also, not all syntactical constructs of C++ are supported, e.g., range-for loops. In general, the body of a user function should be written in C-compatible syntax. SkePU 2 does not enforce these rules with error messages at this time.

User functions can be nested, i.e., called from inside other user functions. This is demonstrated in Listing 5.

Listing 5: Mandelbrot fractal generation in SkePU 2.

```
[[skepu::userconstant]] constexpr float
  CENTER_X = -.5f,
  CENTER_Y = 0.f,
  SCALE = 2.5f;

[[skepu::userconstant]] constexpr size_t
  MAX_ITERS = 1000;

struct cplx
{
  float a, b;
};

cplx mult_c(cplx lhs, cplx rhs)
{
  cplx r;
  r.a = lhs.a * rhs.a - lhs.b * rhs.b;
  r.b = lhs.b * rhs.a + lhs.a * rhs.b;
  return r;
}

cplx add_c(cplx lhs, cplx rhs)
{
  cplx r;
  r.a = lhs.a + rhs.a;
  r.b = lhs.b + rhs.b;
  return r;
}

size_t mandelbrot_f(skepu2::Index2D index, size_t height, size_t width)
{
  cplx a;
  a.a = SCALE / height * (index.col - width/2.f) + CENTER_X;
  a.b = SCALE / height * (index.row - width/2.f) + CENTER_Y;
  cplx c = a;

  for (size_t i = 0; i < MAX_ITERS; ++i)
  {
    a = add_c(mult_c(a, a), c);
    if ((a.a * a.a + a.b * a.b) > 4)
      return i;
  }
  return MAX_ITERS;
}

auto mandelbrot = skepu2::Map<0>(mandelbrot_f);
```

3.3 User Types and Constants

For many applications, basic types such as `int` and `float` may not be sufficient in a high-level programming interface. SkePU 2 therefore includes the possibility of using a custom `struct` as the element type in smart containers or used as extra argument to a skeleton instance. Even then, there are major restrictions on such types depending on the backends used; the type should not have any features outside those of a C-style `struct` and the memory layout needs to match across backends.

Listing 5 demonstrates user types in SkePU 2 with the use of a complex number type `cplx` for Mandelbrot fractal generation. Functions operating on objects of type `cplx` are defined as free functions and are treated as user functions by the precompiler. The example also uses the related feature *user constants*, e.g., `MAX_ITERS`, which are compile-time constant values that can be read in user functions. These objects are annotated with the `[[skepu2::userconstant]]` attribute.

3.4 Improved Type Safety

One of the goals with the SkePU 2 design was to increase the level of type safety from SkePU 1. In the following example, a programmer has made the mistake of supplying a unary user function to Reduce. Listing 6 shows the error in SkePU 1 code, and Listing 7 illustrates the same in SkePU 2 syntax.

Listing 6: Faulty SkePU 1 code.

```
UNARY_FUNC(plus_f, float, a,
  return a;
)

skepu::Vector<float> v(N);
skepu::Reduce<plus_f> globalSum(new plus_f);
globalSum(v);
```

Listing 7: Faulty SkePU 2 code.

```
[[skepu::userfunction]]
float plus_f(float a) {
  return a;
}

skepu2::Vector<float> v(N);
auto [[skepu::instance]] globalSum = skepu2::Reduce(plus_f);
globalSum(v);
```

Listing 8: Error messages from SkePU 1 and 2.

```
// In SkePU 1, at run-time:
[SKEPU_ERROR] Wrong operator type!
              Reduce operation require binary user function.

// In SkePU 2, at compile-time:
error: no matching function for call to 'Reduce'
   auto [[skepu::instance]] globalSum = skepu2::Reduce(plus_f);
                                        ^~~~~~~~~~~~~~
note: candidate template ignored: failed template argument deduction
   Reduce(T(*red)(T, T))
```

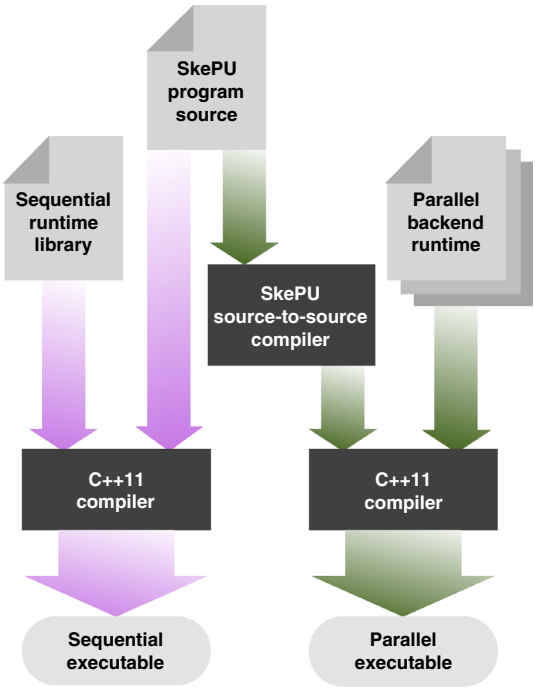
The SkePU 1 example compiles without problem, and only at run-time terminates with the error message in Listing 8. The message itself is shared between all reduce instances, limiting the information obtained by the user. SkePU 2, on the other hand, halts compilation and prints an error message even before the precompiler has transformed the code. It directs the user to the affected skeleton instance. (The message does not directly describe the issue, an aspect which can be further improved with C++11's `static_assert`.)

4 SkePU 2 Implementation

SkePU 2 is implemented in three parts. There is a sequential runtime system, a source-to-source compiler tool, and the parallel runtime system with multiple backends supported. The integration of these parts is illustrated in Fig. 1.

A SkePU 2 program can be compiled with any standard C++11 compiler, producing a sequential executable. This means that the sequential skeletons can act as a reference implementation, both to users who can test their applications sequentially at first,

Fig. 1 SkePU 2 compiler chain



with the advantages of simpler debugging and faster builds and to SkePU backend maintainers.

The parallel runtime of SkePU 2 is based on the original SkePU backends. By a combination of using new and powerful C++11 language features, offloading boilerplate work to the precompiler, and general improvement of the implementation structure, the verbosity and code size of the implementation has been greatly reduced. In some areas, e.g., combining the source code for unary, binary, and ternary Map skeletons into a single variadic template, the amount of lines of code is reduced by over 70%.

4.1 Source-to-Source Compiler

The role of SkePU 2's source-to-source precompiler is to transform programs written for the sequential interface for parallel execution.

Listing 9: Before precompiler transformation.

```
template<typename T>
T arr(skepu2::Index1D row, T *m, const T *v, T *v2 [[skepu::out]])
{
  // body
}
```

Listing 10: After precompiler transformation.

```
struct skepu2_userfunction_arr_float {
  using T = float;
  constexpr static size_t totalArity = 4;
  constexpr static size_t anyContainerArity = 3;
  constexpr static bool indexed = 1;
  static skepu2::AccessMode anyAccessMode[anyContainerArity];
  using Ret = float;

#define SKEPU_USING_BACKEND_CUDA 1
  static inline SKEPU_ATTRIBUTE_FORCE_INLINE __device__ float
  CU(skepu2::Index1D row, T *m, const T *v, T *v2) {
    // body
  }
#undef SKEPU_USING_BACKEND_CUDA

#define SKEPU_USING_BACKEND_OMP 1
  static inline SKEPU_ATTRIBUTE_FORCE_INLINE float
  OMP(skepu2::Index1D row, T *m, const T *v, T *v2) {
    // body
  }
#undef SKEPU_USING_BACKEND_OMP
};

skepu2::AccessMode
skepu2_userfunction_arr_float::anyAccessMode[anyContainerArity] {
  skepu2::AccessMode::ReadWrite,
  skepu2::AccessMode::Read,
  skepu2::AccessMode::Write,
};

"#define SKEPU_USING_BACKEND_CL 1
static float arr_float(index1_t row, __global float * m,
  __global const float * v, __global float * v2) {
  typedef float T;
  // body
}"
```

The precompiler is limited by design. Its main purpose is to transform user functions, for example by adding `__device__` keywords for CUDA variants and stringifying the OpenCL variant. A user function is represented as a `struct` with static member functions in the transformed program. The precompiler also transforms skeleton instances, redirecting to a completely different implementation accepting the `structs` as template arguments. It also redefines user types for backends where necessary. For some backends such as OpenCL and CUDA, all kernel code is generated by the precompiler.

An example of a transformation[3] of the template user function in Listing 9 can be seen in Listing 10.

In the future, the precompiler role will be expanded to include selection of system-specific user function specializations, guided by a platform description language [14]. The precompiler can either select the most appropriate specialization directly, or include multiple variants and generate logic to select the best one at run-time based on dynamic conditions. The extensibility was an important motivation when deciding to construct SkePU 2 as a precompiler-based framework using the Clang libraries.

[3] Slightly altered and reformatted for presentation purposes. The intermediate code format is undocumented and subject to change.

4.2 Dependencies

SkePU 2 requires the target platform to provide a C++11-conforming compiler. C++11 support in compilers is quite mature today, and support is available in all recent versions of GCC, Clang, and the Intel, Microsoft, and Nvidia toolchains. Access to the precompiler tool is also necessary for parallel builds, so by extension a development system needs to be able to build LLVM and Clang. However, the SkePU 2 tool chain has been designed to allow for cross-precompilation. In other words, all decisions based on the architecture and available accelerators, etc., are made after the precompilation step.

5 Readability Evaluation

The interface of SkePU 2 aims to improve on that of SkePU 1 with increased clarity and a syntax that looks and feels more native to C++. To evaluate this, a survey was issued to 16 participants, all master-level students in computer science. The participants were presented with two short example programs: one very simple and one somewhat complex, each both in SkePU 1 and SkePU 2 syntax. To avoid biasing either of the SkePU versions, the order of introductions was reversed in half of the questionnaires. See the thesis [11] for more discussion on the survey, including the code examples presented to the respondents.

Note, however, that the survey was issued when the syntax of SkePU 2 was not yet finalized. At the time C++11 attributes were required to guide the precompiler: `skepu::userfunction` on user functions, `skepu::instance` on skeleton instances, `skepu::usertype` on user-defined `struct` types appearing in user functions, and `skepu::userconstant` for global constants on `constexpr` global variables. The attributes allowed for a straightforward implementation of the precompiler, and the reasoning was that clearer expression of intent from the programmer could improve any error messages emitted.

Figure 2 presents the responses comparing the two SkePU versions in terms of code clarity (to the question *How would you rate the clarity of this code in relation to the previous example?*). The usability evaluation indicates that the SkePU 1 interface

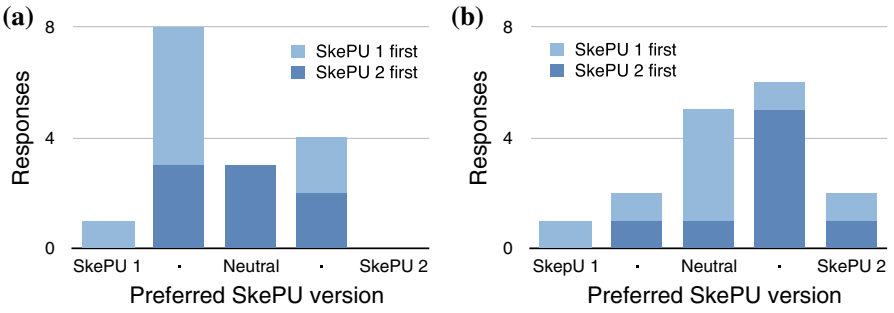


Fig. 2 Comparison of code clarity, SkePU 1 versus SkePU 2. **a** Simple example and **b** complex example

is sometimes preferred to the SkePU 2 variant, at least when the user is not used to C++11 attributes as indicated by the free-form comments in the survey. We realized that the decision to use attributes as a fundamental part of the syntax needed to be revisited.

In the more complex example, respondents generally considered the SkePU 2 variant to be clearer. We believe that the reason for this is the fact that it has fewer user functions and skeleton instances than the SkePU 1 version (thanks to the increased flexibility offered in SkePU 2). The user functions are also fairly complex, so the macros in SkePU 1 may be more difficult to understand.

6 Performance Evaluation

The system used for testing consists of two eight-core Intel Xeon E5-2660 "Sandy Bridge" processors at 2.2 GHz with 64 GB DDR3 1600 MHz memory, and a Nvidia Tesla k20x GPU. The test programs were compiled with GCC g++ 4.9.2 or, when CUDA was used, Nvidia CUDA compiler 7.5 using said g++ as host compiler. Separate tests were conducted on consumer-grade development systems, showing similar results after accounting for the performance gap. The framework has also been tested on multi-GPU systems using CUDA and OpenCL, and a Xeon Phi accelerator using Intel's OpenCL interface. Results are shown in Figs. 3 and 4. All tests include data movement to and from accelerators, where applicable.

The following test programs were evaluated:

- **Pearson product-movement correlation coefficient**
  A sequence of three independent skeletons: one Reduce, one unary MapReduce and one binary MapReduce. The user functions are all trivial, containing a single floating point operation. The problem size is the vector length.
- **Mandelbrot fractal**
  A Map skeleton with a non-trivial user function. There is no need for copy-up of data to a GPU device in this example, but the fractal image is copied down from device afterwards. In fact, there are no non-uniform inputs to the user function, as the index into the output container is all that is needed to calculate the return value. The problem size is one side of the output image.
- **Coulombic potential**
  Calculates electrical potential in a grid, from a set of charged particles. An iterative computation invoking one Map skeleton per iteration. The user function takes one argument, a random-access vector containing the particles. It also receives a unique two-dimensional index from the runtime, from which it calculates the coordinates of its assigned point in the grid.
- **N-body simulation**
  Performs an N-body simulation on randomized input data. The program is similar to Coulombic potential, both in its iterative nature and the types of skeletons used.

The preview release of SkePU 2 has not been optimized for performance. Even so, it has already shown to match or surpass the performance of SkePU 1.2 in some tests. However, the results vary with the programs tested and seems particularly dependent on

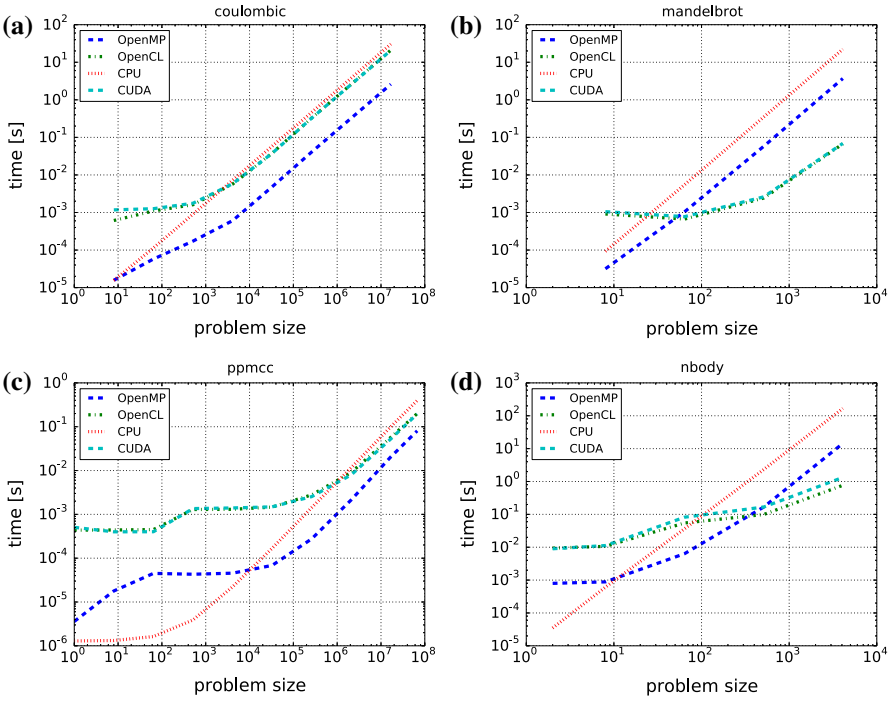


Fig. 3 Test program evaluation results. Log–log scale. **a** Coulombic potential, **b** Mandelbrot fractal, **c** Pearson product-movement correlation co-efficient and **d** N-body simulation

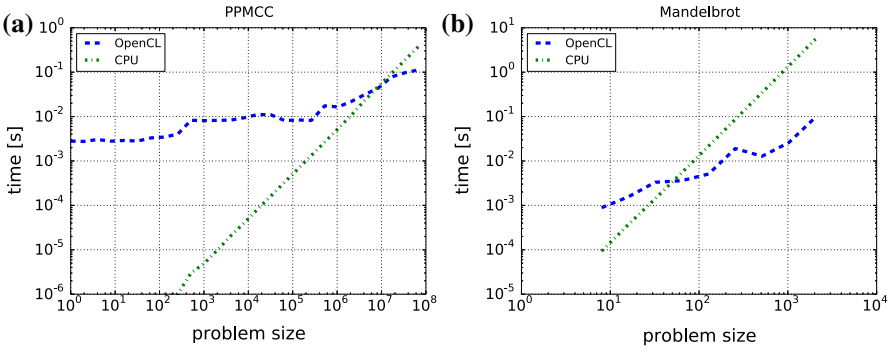


Fig. 4 Evaluation results on Xeon Phi using OpenCL. **a** Pearson product-movement correlation co-efficient and **b** Mandelbrot fractal

the choice of compiler. A mature optimizing C++11 compiler is required for SkePU 2 to be competitive performance-wise.

In cases where the increased flexibility of SkePU 2 allows a program to be implemented more efficiently for example by reducing the amount of auxiliary data or number of skeleton invocations SkePU 2 may outperform SkePU 1 significantly. Figure 5 shows such a case: approximation of the natural logarithm using Taylor series.

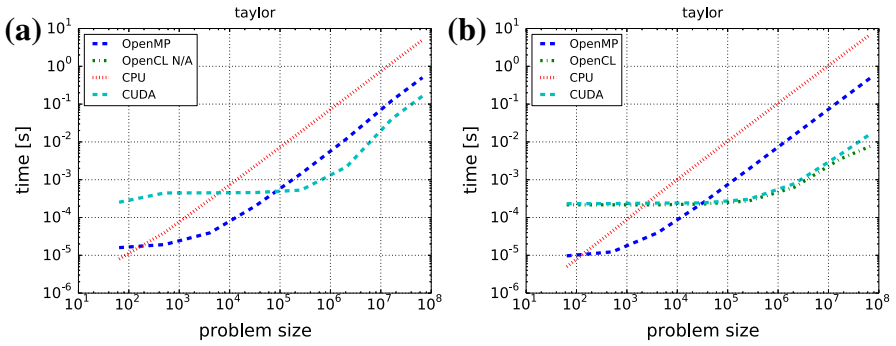


Fig. 5 Comparison of Taylor series approximation. **a** SkePU 1.2 and **b** SkePU 2

For SkePU 1, this is implemented by a call to Generate followed by a call to MapReduce; in SkePU 2 a single MapReduce is enough, reducing the number of GPU kernel launches and eliminating the need for $\mathcal{O}(n)$ auxiliary memory.

7 Related Work

SkelCL [22] is an actively developed OpenCL-based skeleton library. It is more limited than SkePU 2, both in terms of programmer flexibility and available backends. Implemented as a library, it does not require the usage of a precompiler like SkePU 2, with the downside that user functions are defined as string literals. SkelCL includes the AllPairs skeleton [21], an efficient implementation of certain complex access modes involving multiple matrices. In SkePU 2 matrices are accessed either element-wise or randomly.

Nvidia *Thrust* [1] is a C++ template library with parallel CUDA implementations of common algorithms. It uses common C++ STL idioms, and defines operators (comparable to SkePU 2 user functions) as native functors. The implementation is in effect similar to that of SkePU 2, as the CUDA compiler takes an equivalent role to the source-to-source compiler presented in this article.

The *Muesli* skeleton library [2] for MPI and OpenMP execution has been ported for GPU execution [10]. It currently has a limited set of data-parallel skeletons which makes it difficult to port more complex applications.

Marrow [15] is a skeleton programming framework for single-GPU OpenCL systems. It provides both data and task parallel skeletons with the ability to compose skeletons for complex computations.

Bones is a source-to-source compiler based on algorithmic skeletons [18]. It transforms `#pragma`-annotated C code to parallel CUDA or OpenCL using a translator written in Ruby. The skeleton set is based on a well-defined grammar and vocabulary. Bones places strict limitations on the coding style of input programs.

SkePU 2 is not alone in the aim to apply modern C++ features in a parallel and heterogeneous programming context. *PACXX* is a unified programming model for systems with GPU accelerators [12], utilizing the new C++14 language. PACXX shares

many fundamental choices with SkePU 2, for example using attributes and basing the implementation on Clang. However, PACXX is not an algorithmic skeleton framework.

SYCL [19] is an emerging modern C++ interface to OpenCL. It simplifies source code sharing across CPU and GPU and introduces an intermediate bitcode format for kernels. Like SkePU 2 it provides a lambda expression syntax for declaring kernels, and a larger subset of C++ can be used inside them.

A future version of the C++ standard will include extensions to the standard library for parallelism [13]. Prototype implementations exist today, e.g., based on SYCL. Parts of the proposal resemble algorithmic skeletons, but overall the algorithms are more specialized.

A different kind of GPU programming research project, *CU2CL* [16] was a pioneer in applying Clang to perform source-to-source transformation; the library support in Clang for such operations has been greatly improved and expanded since then. A very recent Clang- and LLVM-based project is *gpucc* [23], the first open-source alternative to Nvidia's CUDA compiler. It focuses on improving both compile-time and run-time performance, outperforming `nvcc` in some tests.

We refer to earlier SkePU publications [5,7,9] for other work relating to specific features, such as smart containers.

8 Conclusions and Future Work

We have presented SkePU 2, a next generation skeleton programming framework for heterogeneous parallel systems. SkePU 2 has a modern, native C++ syntax and new opportunities for backend-specific tuning and optimization, while extending the functionality and flexibility to reduce the burden of porting implementations of complex applications to the framework. This has been done by introducing C++11 language features and a source-to-source precompiler based on Clang, all while preserving the performance characteristics of previous versions of SkePU.

A survey on the readability of an in-progress version of the SkePU 2 syntax convinced us to remove most usage points of C++11 attributes in the syntax, as the respondents clearly indicated that the attributes had an adverse effect on their understanding of the programs. Some attributes remain and are required for advanced features, however.

A preview version of SkePU 2 is available as an open-source distribution at http://www.ida.liu.se/labs/pelab/skepu/. SkePU 2 is an active research project, and as such both the interface and implementation are subject to change.

Future work includes integrating SkePU 2 with other tools and libraries. For example, a platform description language such as XPDL [14] can be used to guide the automated selection of specializations of user functions and user-defined multi-backend components [6].

Acknowledgements This work has been partly funded by EU FP7 project EXCESS (http://excess-project.eu), by SeRC (http://www.e-science.se), and by the Swedish National Graduate School in Computer Science (CUGS). We thank the National Supercomputing Centre (NSC) and SNIC for access to their GPU-based computing resources (Project 2016/5-6).

References

1. Bell, N., Hoberock, J.: Thrust: a productivity-oriented Library for CUDA. In: GPU Computing Gems, Jade Edition, vol. 2, pp. 359–371. Morgan Kaufmann, San Francisco, CA (2011)
2. Ciechanowicz, P., Poldner, M., Kuchen, H.: The Münster skeleton library Muesli—a comprehensive overview. ERCIS Working Paper No. 7 (2009)
3. Cole, M.I.: Algorithmic Skeletons: Structured Management of Parallel Computation. Pitman and MIT Press, Cambridge, MA (1989)
4. Danelutto, M., Matteis, T.D., Mencagli, G., Torquati, M.: Parallelizing high-frequency trading applications by using C++ 11 attributes. In: Proceedings of the 2015 IEEE Trustcom/BigDataSE/ISPA, vol. 3, pp. 140–147 (2015). doi:10.1109/Trustcom.2015.623
5. Dastgeer, U., Kessler, C.: Smart containers and skeleton programming for GPU-based systems. Int. J. Parallel Program. (2016). doi:10.1007/s10766-015-0357-6
6. Dastgeer, U., Kessler, C.W.: Conditional component composition for GPU-based systems. In: Proceedings of MULTIPROG-2014 Workshop at HiPEAC-2014 Conference, Vienna, Austria (2014)
7. Dastgeer, U., Li, L., Kessler, C.: Adaptive implementation selection in the SkePU skeleton programming library. In: Advanced Parallel Processing Technologies: 10th International Symposium, APPT 2013, Stockholm, Sweden, August 27–28, 2013, vol. 8299, pp. 170–183. Springer, Berlin, Heidelberg (2013). doi:10.1007/978-3-642-45293-2_13
8. Dastgeer, U., Li, L., Kessler, C.: The PEPPHER composition tool: performance-aware composition for GPU-based systems. Computing **96**(12), 1195–1211 (2013)
9. Enmyren, J., Kessler, C.W.: 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, pp. 5–14. ACM (2010)
10. Ernsting, S., Kuchen, H.: Algorithmic skeletons for multi-core, multi-GPU systems and clusters. Int. J. High Perform. Comput. Netw. **7**, 129–138 (2012)
11. Ernstsson, A.: SkePU 2: language embedding and compiler support for flexible and type-safe skeleton programming. Master's thesis, Linköping University, Linköping, LIU-IDA/LITH-EX-A–16/026–SE (2016)
12. Haidl, M., Gorlatch, S.: PACXX: towards a unified programming model for programming accelerators using C++14. In: Proceedings of the 2014 LLVM Compiler Infrastructure in HPC. LLVM-HPC '14, pp. 1–11. IEEE Press, Piscataway (2014)
13. Hoberock, J.: Working draft, technical specification for C++ extensions for parallelism. Technical Report N4505, ISO/IEC JTC1/SC22/WG21 (2015)
14. Kessler, C., Li, L., Atalar, A., Dobre, A.: XPDL: extensible platform description language to support energy modeling and optimization. In: Proceedings of 44th International Conference on Parallel Processing Workshops, ICPP-EMS Embedded Multicore Systems, in Conjunction with ICPP-2015 (2015). doi:10.1109/ICPPW.2015.17
15. Marques, R., Paulino, H., Alexandre, F., Medeiros, P.D.: Algorithmic skeleton framework for the orchestration of GPU computations. In: European Conference on Parallel Processing, vol. 8097, pp. 874–885. Springer, Berlin, Heidelberg (2013). doi:10.1007/978-3-642-40047-6_86
16. Martinez, G., Gardner, M., Feng, W.C.: CU2CL: a CUDA-to-OpenCL translator for multi-and many-core architectures. In: IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS), IEEE, pp. 300–307. (2011)
17. Maurer, J., Wong, M.: Towards support for attributes in C++ (revision 6). Technical Report N2761, ISO/IEC JTC1/SC22/WG21 (2008)
18. Nugteren, C., Corporaal, H.: Introducing 'Bones': a parallelizing source-to-source compiler based on algorithmic skeletons. In: Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units, GPGPU-5, pp. 1–10. ACM, New York (2012). doi:10.1145/2159430.2159431

19. Potter, R., Keir, P., Bradford, R.J., Murray, A.: Kernel composition in SYCL. In: Proceedings of the 3rd International Workshop on OpenCL, IWOCL '15, pp. 11:1–11:7. ACM, New York (2015). doi:10.1145/2791321.2791332
20. Sjöström, O., Ko, S.H., Dastgeer, U., Li, L., Kessler, C.: Portable parallelization of the EDGE CFD application for GPU-based systems using the SkePU skeleton programming library. In: Joubert, G.R., Leather, H., Parsons, M. Peters, F., Sawyer, M. (eds.) Advances in Parallel Computing, Volume 27: Parallel Computing: On the Road to Exascale. Proceedings of ParCo-2015 Conference, Edinburgh, Sep. 2015, pp. 135–144. IOS Press (2016). doi:10.3233/978-1-61499-621-7-135
21. Steuwer, M., Friese, M., Albers, S., Gorlatch, S.: Introducing and implementing the AllPairs skeleton for programming multi-GPU systems. Int. J. Parallel Program. **42**(4), 601–618 (2013)
22. Steuwer, M., Kegel, P., Gorlatch, S.: SkelCL a portable skeleton library for high-level GPU programming. In: 16th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'11) (2011)
23. Wu, J., Belevich, A., Bendersky, E., Heffernan, M., Leary, C., Pienaar, J., Roune, B., Springer, R., Weng, X., Hundt, R.: gpucc: an open-source GPGPU compiler. In: Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO, pp. 105–116. ACM, New York (2016). doi:10.1145/2854038.2854041