SkePU User Guide$^1$

Oskar Sjöström
Linköping University

Christoph Kessler
Linköping University

May 28th, 2015
Revised July 20, 2015

$^1$Copyright ©2015 held by the authors.
Contents

1 Introduction .................................................. 4
  1.1 What is SkePU? ............................................. 4
  1.2 Advantages of using SkePU ............................... 4
  1.3 The structure of a SkePU program ....................... 4

2 Possible combinations of inputs for each type of skeleton ........... 5
  2.1 Map ..................................................... 5
  2.2 MapReduce ............................................... 5
  2.3 MapArray ............................................... 5
  2.4 MapOverlap ............................................. 6
  2.5 MapOverlap2d .......................................... 6
  2.6 Generate ............................................... 6
  2.7 Reduce ................................................ 6
  2.8 Scan .................................................. 6

3 User Function Specification .................................. 7
  3.1 Which types of user functions are compatible with each type of skeleton .... 7
  3.2 Current limitations of user function specifications .................. 8

4 How is it determined which backend will be used? .................... 8

5 Using the Tuning Framework ................................ 8

6 Using SkePU Container Data Structures .......................... 9
  6.1 Wrapping existing array data structures ................... 9
  6.2 Using partial data ....................................... 9
  6.3 Coherence Control ..................................... 9
  6.4 Reuse ................................................ 10

7 Preprocessing variables that affect the behavior of SkePU .......... 10

8 Compilation Issues ........................................... 10

9 Example Programs ............................................. 11
  9.1 Map ..................................................... 11
  9.2 MapReduce ............................................. 12
  9.3 MapArray ............................................. 12
  9.4 MapOverlap ........................................... 13
  9.5 MapOverlap2d ......................................... 14
  9.6 Generate ............................................... 15
  9.7 Reduce ............................................... 15
  9.8 Scan ................................................ 16

References ...................................................... 18
1 Introduction

This SkePU user guide aims to make it easier for current and future users of SkePU to understand the applications and limitations of the library. It assumes that the reader has already got a high-level overview and understanding of SkePU from the published theses, articles and on-line material available on the SkePU web page\(^1\). It includes some information that goes beyond the presentation in these published works, without repeating them.

This introductory section gives thus only a brief introduction to the SkePU library. It will outline what the advantages of using SkePU are and in which situation SkePU is a suitable choice. This section also contains a description of the structure of a SkePU program and its different parts. For more technical details the reader is advised to consult the publications section of the SkePU web page.

1.1 What is SkePU?

SkePU \(^5\) is an open-source skeleton programming framework for multicore CPUs and multi-GPU systems. It is a C++ template library with six data-parallel and one task-parallel skeletons, two generic container types, and support for execution on multi-GPU systems both with CUDA and OpenCL.

1.2 Advantages of using SkePU

There are several advantages to choosing the SkePU library rather than handwritten parallel solutions. It allows the programmer to get the functionality of a parallel program without having to learn all the details of parallel programming such as synchronization. SkePU code is highly portable, as all the implementation details reside in the lower layers of each backend the same code can be run on a wide variety of platforms. SkePU code has been shown to in some cases yield a greater speedup than handwritten CUDA or OpenCL solutions.

1.3 The structure of a SkePU program

This section describes the different parts of a SkePU program. The following example code shows a program performing a dot product implemented with SkePU.

The skeletons and datatypes to be used are included, in this case the SkePU vector and the mapreduce skeleton:

```cpp
#include <iostream>
#include "skepu/vector.h"
#include "skepu/mapreduce.h"

BINARY_FUNC(plus_f, double, a, b,
    return a+b;
)

BINARY_FUNC(mult_f, double, a, b,
    return a*b;
)
```

\(^1\)http://www.ida.liu.se/labs/pelab/skepu
Inside the main function, the SkePU containers and skeleton objects are initialized and the skeleton call is made:

```c
int main()
{
    skepu::MapReduce <mult_f, plus_f> dotProduct(new mult_f, new plus_f);
    skepu::Vector <double> v1(500,4);
    skepu::Vector <double> v2(500,2);
    double r = dotProduct(v1,v2);
    std::cout <<"Result: " << r << 

    return 0;
}
```

The following sections will outline some of the limitations of the SkePU library that the programmer should be aware of. We will also provide some instructions on how to make use of various SkePU features such as offline tuning and backend selection.

2 Possible combinations of inputs for each type of skeleton

For each skeleton there is a set of backend implementations to choose from. There is not necessarily an implementation available for every combination of SkePU containers. Because of this, the possible combinations available are listed in this section. Any other input parameters used by the skeletons are also listed.

Some skeletons also allow for a constant input. This should be set before executing the skeleton by calling the setConstant member function.

2.1 Map

Map takes one to three SkePU containers as inputs. These containers should all be of the same type (i.e., all matrices or all vectors).

If only one container is used then the output may be given in the same container. In all other cases the skeleton requires an additional result parameter.

Map may also take a constant input.

2.2 MapReduce

MapReduce takes the same types of input as the map skeleton.

2.3 MapArray

The first input operand to MapArray is always a SkePU vector. This vector is accessed as one element inside the user function, similarly to in the Map skeleton.

The second input operand can be any one of Vector, Matrix, SparseMatrix and MultiVector. This second input operand can be accessed in its entirety inside the user function.

The output operand is given in a separate parameter. When the second parameter is a matrix, the output parameter may be either a matrix or a vector. In the case that the second parameter is a vector, MultiVector or sparse matrix the output parameter is a vector.

MapArray may also take a constant input operand.
2.4 MapOverlap

MapOverlap takes either a SkePU matrix or vector as its main input operand. The output can be given either in the same parameter or in a separate output parameter.

The MapOverlap skeleton also has several other inputs.

When using a vector as input, one should also specify the `EdgePolicy` parameter. `EdgePolicy` is an enum with three possible values, CONSTANT, CYCLIC and DUPLICATE. This parameter is used to decide how to handle the overlap at the beginning and end of the vector.

When using a matrix as input, one should instead specify the `OverlapPolicy`. OverlapPolicy is an enum with three possible values, OVERLAP_ROW_WISE, OVERLAP_COL_WISE and OVERLAP_ROW_COL_WISE.

There is also an optional parameter `pad` which may be used in conjunction with either of the previous inputs. `Pad` is used to decide the size of the overlap.

2.5 MapOverlap2d

MapOverlap2d will take one SkePU matrix as an input. The result can be given either in the same parameter or in a separate one.

MapOverlap2d also has an optional matrix input which can be used to set a filter.

Another alternative is to set the number of rows and columns used as neighbouring elements during the calculation.

Finally there is another optional input which can be used in combination with any of the previous options, a boolean input which determines if tiling will be used or not.

2.6 Generate

Generate can be used to construct either a vector or a matrix.

When generating a vector, the first parameter is the number of elements of the vector and the second parameter is the vector to be filled with generated data.

When generating a matrix, the skeleton takes three inputs. The first two parameters are the numbers of rows and columns of the matrix, and the last parameter is the matrix to be filled with generated data.

Generate may also take a constant input.

2.7 Reduce

The reduce skeleton takes only one SkePU container as an input. This input can be either a vector, matrix or sparse matrix. The output from the reduction is given as a return value.

Additionally, when performing the reduction on a matrix or sparse matrix, the reduce skeleton has an optional parameter `ReducePolicy`. ReducePolicy is an enum with two possible values, REDUCE_ROW_WISE_ONLY or REDUCE_COL_WISE_ONLY. This parameter can be used to decide how the reduction will be performed.

2.8 Scan

The scan skeleton takes one input parameter, which can be either a vector or a matrix. The output can be given either in the same container or in a separate parameter.

After the SkePU container parameters, the `ScanType` should be given. `ScanType` is an enum with two possible values, INCLUSIVE or EXCLUSIVE. This determines the type of scan that is performed.

The scan skeleton may also take an optional parameter providing the starting point of the scan.
3 User Function Specification

3.1 Which types of user functions are compatible with each type of skeleton

For each skeleton there are requirements regarding which type of user function (and thus, user function definition macro) can be used with it. This section lists the types of user functions that are compatible with each skeleton. The macros for the definitions of each type of user function can be found in the `operator_macros` sourcecode files.

- Map

  UNARY_FUNC
  BINARY_FUNC
  TERNARY_FUNC

- Mapreduce

  See map and reduce respectively.

- Maparray

  ARRAY_FUNC
  VAR_FUNC

- Mapoverlap

  OVERLAP_DEF_FUNC
  OVERLAP_FUNC
  OVERLAP_FUNC_STR

- Mapoverlap2d

  OVERLAP_FUNC_2D_STR

- Generate

  GENERATE_FUNC
  GENERATE_FUNC_MATRIX

- Reduce

  BINARY_FUNC

- Scan

  BINARY_FUNC
3.2 Current limitations of user function specifications

The SkePU user functions are, in principle, limited by the backends that they will possibly be executed through. For example, if the CUDA backend is used then any functions called from within the user function needs to also be available to the GPU.

In general, it is advised that user functions consist of standalone computations without loops or conditional statements. With the exception of some basic functions, which are available on all platforms (such as common math functions), function calls should also not be used within a user function.

As a consequence of this, only basic data types should be used. This means primitive data types such as integers, doubles and also structs can be used, but class types such as strings should not be used.

4 How is it determined which backend will be used?

SkePU provides three ways of controlling the backend selection:

1. Through code
   The programmer can explicitly call any of the available backends to manually choose which one will be used.

2. Through compilation
   The compilation options will decide which backends are made available to SkePU. For example, if the program is compiled with for example the SKEPU_CUDA flag, then the CUDA backend will be used. See Sections 7 and 8 for further details.

3. By SkePU itself
   When multiple backends are available and no decision has been made by the programmer, SkePU will attempt to make an intelligent decision regarding which backend to use.

5 Using the Tuning Framework

The tuning framework in SkePU [2, 4] is based on execution plans. An execution plan in SkePU is a data structure containing information regarding which backend should be used for each problem size, as well as some backend-specific data.

Below is a code example showing how to manually create and apply an execution plan to a skeleton.

```cpp
skepu::Reduce<plus_f> globalSum(new plus_f);
skepu::Vector<double> input(100,10);
skepu::ExecPlan plan;
plan.add(1,5000, CPU_BACKEND);
plan.add(5001,1000000, OMP_BACKEND,8);
plan.add(1000001,INFINITY,CL_BACKEND,65535,512);
globalSum.setExecPlan(plan);
std::cout<<"Result: "<globalSum(input);
```

What this code will do is ensure that for problem sizes (input vector lengths) between 1 and 5000 the sequential CPU backend will be used in all future calls of `globalSum`. For problem sizes
between 5001 and 1000000 the OpenMP backend with 8 threads will be called, and for problem sizes larger than that the OpenCL backend is called with 65535 blocks and 512 threads.

In order to improve upon a manually selected execution plan such as the one in the example above, the SkePU tuner class can be used. In the folder examples/tuning there are several code examples showing how to perform training runs of the code. At the end of execution, the tuner class will create a .dat output file. This output file contains information regarding at which point it is optimal to switch between the different backends. This file can then be used to load an execution plan.

6 Using SkePU Container Data Structures

6.1 Wrapping existing array data structures

SkePU Vectors can wrap existing data without performing any copy operations. In order to allow this, SkePU provides a custom constructor.

skepu::Vector(T * const ptr, size_type size, bool deallocEnabled = true);

The first parameter is a pointer to the start of the data. The second parameter is the number of elements in the vector.

6.2 Using partial data

In the same way as for an STL vector, it is possible to select parts of a SkePU vector or matrix as an input to a skeleton call without creating a new data structure. This is accomplished by the use of iterators.

The following piece of example code shows this, taken from Dastgeer’s doctoral thesis [1]:

skepu::Vector<int> v0(10); // create vector (v0) of 10 elements
... skel( v0 ); // readwrite all 10 elements of v0
skel( v0.begin()+5, v0.end() ); // readwrite last 5 elements of v0
skel( v0.begin()+3, v0.end()-3 ); // readwrite elements 3,...,6 of v0

This can lead to the creation of partial device copies when executing on a GPU. Details of how this is handled by SkePU can be found in [3] and in Chapter 4 of Dastgeer’s doctoral thesis [1].

6.3 Coherence Control

The SkePU smart containers [3] aim to limit the amount of communication between GPU device memory and main memory by transparent software caching with lazy copying, applying a MSI-based coherence protocol for device copies. This means that the data inside SkePU data structures is not always consistent between device memory and main memory. The user should in particular take note of the behavior when using the () and [] operators.

The () operator will not enforce any synchronization, it will simply return the (possibly, invalidated) data that is available in main memory and thus accessible from the CPU.

The [] operator will make sure that the data that it returns has been synchronized. This does however not mean that the entire data structure has been synchronized upon a [] access; only the accessed range of elements is updated.

If only part of the data in a container is used, then the [] operator will likely yield the best performance.
However if the user wishes to traverse the entirety of the data structure, better performance is often achieved by first synchronizing the entire container and then accessing data with the () operator. In order to synchronize a container the member function updateHost() can be called.

6.4 Reuse

The SkePU smart containers are designed in order to minimize data transfers. Users will get the most benefit from this if the same data structures are reused for multiple computations as SkePU can then execute these skeletons on data that is already present on the GPU.

7 Preprocessing variables that affect the behavior of SkePU

In SkePU there are a number of preprocessing variables that can be set by the programmer to affect parts of SkePU's behavior.

- **SKEPU_NUMGPU**
  
  As a default SkePU will only be using one GPU. If a system has multiple GPUs available, this variable can be set to determine the number of GPUs that will be used by SkePU. Currently the maximum number of GPU’s that SkePU can make use of is four.

- **USE_PINNED_MEMORY**
  
  If this variable has been set and the CUDA backend is being used SkePU containers will be using pagelocked memory. This allows for faster datatransfers between the CPU and the GPU aswell as the possibility for SkePU to use multiple CUDA streams to overlap datatransfers with kernel execution.

- **SKEPU_MAX_GPU_THREADS**
  
  As a default, SkePU will attempt to use as many GPU threads as possible. In some cases it may be desirable to limit the number of threads that will be used. For instance, if the user function is very large, the GPU may not contain enough registers to be able to execute the function on the maximum number of threads supported by the GPU. The **SKEPU_MAX_GPU_THREADS** variable can then be set to limit the maximum number of threads that SkePU is allowed to use.

- **SKEPU_OPENMP_THREADS**
  
  The default behavior of SkePU is to use as many threads as possible. This variable can be used to limit the number of threads used by the OpenMP backend.

8 Compilation Issues

The compilation is straightforward in most cases. Following are some possible compilation commands to enable certain kind of skeleton implementations:

- Sequential C++ (Default backend, always available):
  
  -> g++ <filename>.cpp -I<path-to-skepu-include-folder>
  
  -> nvcc <filename>[.cu | .cpp] -I<path-to-skepu-include-folder>

- OpenMP:
-> g++ <filename>.cpp -I<path-to-skepu-include-folder> -DSKEPU_OPENMP -fopenmp
-> nvcc <filename>[.cu | .cpp] -I<path-to-skepu-include-folder> -DSKEPU_OPENMP
   -Xcompiler -fopenmp

• OpenCL:

-> g++ <filename>.cpp -I<path-to-skepu-include-folder> -DSKEPU_OPENCL
   -I<path-to-opencl-include-folder> -lOpenCL

• Both OpenCL and OpenMP:

-> g++ <filename>.cpp -I<path-to-skepu-include-folder> -DSKEPU_OPENMP -fopenmp
   -DSKEPU_OPENCL -I<path-to-opencl-include-folder> -lOpenCL

• CUDA:

-> nvcc <filename>.cu -I<path-to-skepu-include-folder> -DSKEPU_CUDA -Xcompiler
   -fopenmp

• Both CUDA and OpenMP:

-> nvcc <filename>.cu -I<path-to-skepu-include-folder> -DSKEPU_CUDA -DSKEPU_OPENMP
   -Xcompiler -fopenmp

Note that the compilation command on your machine may look different depending on the
compiler versions and/or installation paths.

9 Example Programs

This section will contain a brief description of each skeleton as well as example source code. All
examples in this section are also available in the examples folder of the SkePU release3.

9.1 Map

The map skeleton performs a one to one mapping between its inputs.

The following example program performs a squaring operation on a five by five matrix:

```cpp
#include <iostream>
#include "skepu/matrix.h"
#include "skepu/map.h"

UNARY_FUNC(square_f, int, a,
    return a*a;
)

int main()
{
    skepu::Map<square_f> square(new square_f);
    skepu::Matrix<int> m(5, 5, 3);
    skepu::Matrix<int> r(5, 5);
    square(m, r);
    std::cout<<"r: " << r <<"\n";
    return 0;
}
```
9.2 MapReduce

The MapReduce skeleton first performs a Map and then a reduction on the result of the Map.

The following example implements a dot product computation.

```cpp
#include <iostream>
#include "skepu/vector.h"
#include "skepu/mapreduce.h"

// User-function used for mapping
BINARY_FUNC(mult_f, float, a, b,
    return a*b;
)

// User-function used for reduction
BINARY_FUNC(plus_f, float, a, b,
    return a+b;
)

int main()
{
    skepu::MapReduce<mult_f, plus_f> dotProduct(new mult_f, new plus_f);
    skepu::Vector<float> v0(20, (float)2);
    skepu::Vector<float> v1(20, (float)5);
    std::cout<<"v0: " <<v0 <<"n"
    std::cout<<"v1: " <<v1 <<"n"
    float r = dotProduct(v0, v1);
    std::cout<<"r: " <<r <<"n"
    return 0;
}
```

9.3 MapArray

In MapArray, each element of the result vector is a function of the corresponding element of one of the input vectors and any number of elements from the other input.

Below follow two examples, the first one using an ordinary Vector container as the secondary input and the other one using the MultiVector container.

```cpp
#include <iostream>
#include "skepu/vector.h"
#include "skepu/maparray.h"

ARRAY_FUNC(arr_f, float, a, b_i,
    int index = (int)b_i;
    return a[index];
)

int main()
{
    skepu::MapArray<arr_f> reverse(new arr_f);
    skepu::Vector<float> v0(10);
    skepu::Vector<float> v1(10);
    skepu::Vector<float> r;
    //Sets v0 = 1 2 3 4 5...
    // v1 = 9 8 7 6 5...
    for(int i = 0; i < 10; ++i)
    {
        v0[i] = (float)(i+1);
        v1[i] = (float)(10-i-1);
    }
```
The following example uses the MultiVector container (SkePU 1.2) with MapArray:

```cpp
#include <iostream>
#include "skepu/maparray.h"
#include "skepu/vector.h"
using namespace skepu;

VAR_FUNC(multivec_f, int, char, MultiVector, index, cont,
char r; // result variable
// extract inputs from multivector:
int *in1 = subvector(cont,0,int*);
char *in2 = subvector(cont,1,char*);
// perform function:
    r = in2[in1[index]]; // will be accessed as single element inside user function
return r;
)

int main()
{
    // input arrays
    int arr1[5] = {0,4,14,20,25};
    char arr2[33] = "Hi, Everybody Let’s Like Oranges";
    int arr3[5] = {0,1,2,3,4}; // input vector, used to divide work between threads,
    // instantiate a SkePU function from MapArray skeleton:
    skepu::MapArray<multivec_f> mvfunc(new multivec_f);
    // create input and output vectors:
    skepu::Vector<int> skeleton_index;
    skepu::Vector<char> result(5,0);
    skeleton_index.assign(arr3,arr3+5);
    // build multivector structure:
    MultiVector cont;
    cont.allocData(3); // def. number of vectors that should be held inside the MultiVector
    cont.addData(0,5*sizeof(int),arr1);
    cont.addData(1,33*sizeof(char),arr2);
    // run the skeleton
    mvfunc(skeleton_index,cont,result);
    // print result
    result.updateHost();
    for (int i=0;i<5;i++)
        printf("result[%d]: %c \n",i,result(i));
    return 0;
}

9.4 MapOverlap

The Mapoverlap skeleton can be used to perform a stencil operation. This example program shows how to use it to compute a convolution.

```
#include <iostream>
#include "skepu/vector.h"
#include "skepu/mapoverlap.h"
OVERLAP_FUNC(over_f, float, 2, a,
```
int main()
{
    skepu::MapOverlap<over_f> conv(new over_f);
    skepu::Vector<float> v0(10,10);
    skepu::Vector<float> r(10);
    std::cout << "v0: " << v0 << "\n";
    conv( v0, r, skepu::CONSTANT, (float)0 );
    std::cout << "r: " << r << "\n";
    return 0;
}

9.5 MapOverlap2d

Mapoverlap2d is used to compute stencil operations in two dimensions.

The following example program shows the computation of a two dimensional convolution.

/* ...
* user-function: For 2D-convolution we use "OVERLAP_FUNC_2D_STR"
* Parameters are:
*   name,  
*   datatype,  
*   overlap length on horizontal axis, 
*   overlap length on vertical axis, 
*   name of parameter,  
*   the stride which is used to access items column-wise, 
*   actual function body.
*/
OVERLAP_FUNC_2D_STR(over_f, int, 1, 2, a, stride,
)

// some size typedefs....
#define OUT_ROWS 16
#define OUT_COLS 10
#define OVERLAP_ROWS 2 // vertical axis
#define OVERLAP_COLS 1 // horizontal axis
#define IN_ROWS (OUT_ROWS + OVERLAP_ROWS*2)
#define IN_COLS (OUT_COLS + OVERLAP_COLS*2)
#define OUT_SIZE (OUT_ROWS * OUT_COLS)
#define IN_SIZE (IN_ROWS * IN_COLS)

int main()
{
    skepu::MapOverlap2D<over_f> mat_conv(new over_f);
    skepu::Matrix<int> m0(IN_ROWS,IN_COLS, -1);
    skepu::Matrix<int> m1(OUT_ROWS,OUT_COLS, 0);
    // initializing non-edge elements of input matrix
    for (int i=OVERLAP_ROWS; i<OUT_ROWS+OVERLAP_ROWS; i++)
        for (int j=OVERLAP_COLS; j<OUT_COLS+OVERLAP_COLS; j++)
            m0(i,j)=i+j;
    std::cout << "Input " << m0 << "\n";
// Applying 2D convolution for neighbouring elements
mat_conv(m0, m1);

std::cout << "Output " << m1 << "\n";
return 0;
}

9.6 Generate

The generate skeleton can be used to generate element values for a SkePU vector or matrix in parallel.

The following example program shows a matrix being generated with random integer values.

```cpp
#include <iostream>
#include <ctime>
#include "skepu/vector.h"
#include "skepu/generate.h"

GENERATE_FUNC(lcg_f, int, int, index, seed,
    unsigned int next = seed;
    unsigned int i;
    for(i = 0; i < index; ++i) {
        next = 1103515245*next + 12345;
    }
    return (next % 10) + 1;
)

int main()
{
    skepu::Generate<lcg_f> rand(new lcg_f);
    skepu::Vector<int> v0;
    rand.setConstant(time(0));
    rand(50, v0);
    std::cout << "v0: " << v0 << "\n";
    return 0;
}
```

9.7 Reduce

The reduce skeleton will apply a binary operator between each element of its input to create an output consisting of a single element.

The following example shows how to calculate the sum of a vector.

```cpp
#include <iostream>
#include "skepu/matrix.h"
#include "skepu/reduce.h"

BINARY_FUNC(plus_f, float, a, b,
    return a+b;
)

int main()
{
    skepu::Reduce<plus_f> globalSum(new plus_f);
    skepu::Vector<float> v0(50, (float)2);
    std::cout << "v0: " << v0 << "\n";
    float r = globalSum(v0);
    std::cout << "Result: " << r << "\n";
    return 0;
}
```
9.8 Scan

This example of the scan skeleton shows how to compute the prefix-sum of a vector.

```cpp
#include <iostream>
#include "skepu/vector.h"
#include "skepu/scan.h"

BINARY_FUNC(plus_f, float, a, b,
    return a+b;
)

int main()
{
    skepu::Scan<plus_f> prefix_sum(new plus_f);
    skepu::Vector<float> v0(50, (float)1);
    skepu::Vector<float> r;
    std::cout << "v0: " << v0 << "\n";
    prefix_sum( v0, r, skepu::INCLUSIVE, (float)10 );
    std::cout << "r: " << r << "\n";
    return 0;
}
```
Acknowledgments

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

Previous major contributors to SkePU include Johan Enmyren and Usman Dastgeer. Streaming support for CUDA in SkePU 1.2 has been contributed by Claudio Parisi from University of Pisa, Italy, in a recent cooperation with the FastFlow project. SkePU example programs in the public distribution have been contributed by Johan Enmyren, Usman Dastgeer, Mudassar Majeed and Lukas Gillsjö.
References


