Introduction & Tutorial

August Ernstsson       Christoph Kessler
{firstname.lastname}@liu.se

Linköping University, Sweden
Overview

- Introduction and history
- SkePU interface
- Installation
- Skeletons in depth
  - Map
  - Reduce
  - MapReduce
  - Scan
  - MapOverlap
- Preview: Future SkePU
- Demonstration, Outlook, Discussion…
Skeleton Programming
Programming parallel systems is hard!

- Resource utilization
- Synchronization, Communication
- Memory consistency
- Different hardware architectures, heterogeneity

Skeleton programming (algorithmic skeletons)

- A high-level parallel programming concept
- Inspired by functional programming
- Generic computational patterns
- Abstracts architecture-specific issues
Skeletons
Parametrizable higher-order constructs

• Map
• Reduce
• MapReduce
• Scan
• and others
User functions
User-defined operators

- Mult
- Add
- Abs
- Pow
- …
Skeleton Programming :: Example

Skeleton parametrisation example
Dot product operation
### Sequential algorithm

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>Mult</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Mult</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>Mult</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>Mult</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>Mult</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>Mult</td>
<td>12</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>Mult</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>Mult</td>
<td>16</td>
</tr>
</tbody>
</table>

![Map](image)
### Parallel algorithm

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>Mult</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>Mult</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>Mult</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>Mult</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>Mult</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>Mult</td>
<td>12</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>Mult</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>Mult</td>
<td>16</td>
</tr>
</tbody>
</table>

Map

Mult
Sequential algorithm

1 2 3 4 5 6 7 8

Add

3 6 10 15 21 28 36

Add

Reduce

Add
Parallel algorithm?

1  2  3  4  5  6  7  8

3  6  10  15  21  28  36

Dependencies!

Add

Reduce  Add

Skeleton Programming :: Reduce

August Ernstsson – 2019-11-27
Parallel algorithm (assuming associativity)
SkePU
• SkePU uses ”modern” C++

```cpp
// "auto" type specifier
auto addOneMap = skepu2::Map<1>(addOneFunc);

skepu2::Vector<float> input(size), res(size);
input.randomize(0, 9);

// Lambda expression
capture by reference
auto dur = skepu2::benchmark::measureExecTime([&] {
    addOneMap(res, input);
});
```

• Implementation reliant on variadic templates, template metaprogramming, and other C++11 features
✓ Efficient parallel algorithms

✓ Memory management and data movement

✓ Automatic backend selection and tuning
History

• **SkePU 1**, released **2010**
  - C++ template-based interface (limited arity)
  - Multi-backend using macro-based code generation

• **SkePU 2**, released **2016**
  - C++11 *variadic* template interface (flexible arity)
  - Multi-backend using source-to-source precompiler

• **SkePU 3**, in development for **2020**
  - Expanding skeleton set, container set, and expressivity
Features

• Skeleton programming framework
  
  • C++11 **library** with skeleton and data container classes
  
  • A **Clang**-based source-to-source **pre-compiler**

• Smart containers: `Vector<T>`, `Matrix<T>`
  
  • In development: `SparseMatrix<T>`

• For **heterogeneous multicore** systems
  
  • Multiple backends

• Active research tool with a number of publications 2010-2019 (see website)
• **Skeletons provided by SkePU**
  
  • Map
  
  • Reduce
  
  • MapReduce
  
  • Scan
  
  • MapOverlap
  
  • *(In development)* MapPairs
C++ interface
(skeletons, smart containers, …)

<table>
<thead>
<tr>
<th>C++</th>
<th>OpenMP</th>
<th>OpenCL</th>
<th>CUDA</th>
<th>MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Multi-core CPU</td>
<td>Accelerator</td>
<td>GPU</td>
<td>Cluster</td>
</tr>
</tbody>
</table>

Internal prototype
Smart Containers

Vector

Matrix

Sparse matrix
• C++ template class instance

• Contains:
  • CPU memory buffer **pointer** (alt. StarPU handles)
  • Size information (size, width/height)
  • OpenCL/CUDA/MPI **handles**
  • Consistency states

• Template type can be **custom struct**, but be careful!

• Data layout not verified across backends/languages
Using SkePU
Source-to-Source Translation

High-level program (C++) → Compiler → Low-level program (Assembly)

High-level program (C++) → Source-to-source translator → Same-level program (C++)
Compilation Architecture

Program sources

Source-to-source compiler

Backend sources (C++, OpenCL, etc.)

SkePU runtime library

Handled by Makefiles

Backend compiler (e.g., GCC)

Executable
Two main installation options:

- Use provided **pre-built** pre-compiler tool and library (For 64-bit Linux systems)

- Use installation script to **download and build** pre-compiler
  (Requires time and disk space for LLVM/Clang source trees)

See website: https://www.ida.liu.se/labs/pelab/skepu/
```cpp
int add(int a, int b, int m) {
    return (a + b) % m;
}

auto vec_sum = Map<2>(add);
vec_sum(result, v1, v2, 5);
```
User Functions

```c
int add(int a, int b, int m)
{
    return (a + b) % m;
}
```

- User functions are C++ (rather, C) functions
- The signature is analyzed by the pre-compiler to extract the skeleton signature
- Each skeleton has their own expected patterns for UF parameters (but the general structure is shared)
- The UF body is **side-effect free** C (compatible with CUDA/OpenCL)
  - No communication/synchronization
  - No memory allocation
  - No disk IO
Flexibility

- **Variable arity** on Map and MapReduce skeletons
- **Index** argument (of current Map’d container element)
- **Uniform** arguments
- Smart container arguments accessible **freely** inside user function
  - **Read-only / write-only / read-write** copy modes
- User function **templates**
```cpp
template<typename T>
T abs(T input)
{
    return input < 0 ? -input : input;
}

template<typename T>
T mvmult(Index1D row, const Mat<T> m, const Vec<T> v)
{
    T res = 0;
    for (size_t i = 0; i < v.size; ++i)
        res += m[row.i * m.cols + i] * v[i];

    return abs(res);
}
```
Advanced/Experim. Features

- Multi-variant user function **specialization**
  - Targeting backend
- Custom **types**
- **Chained** user functions
- In-line **lambda** syntax for user functions
- "**Intrinsic**" functions
  Some functions exist in the standard library of most SkePU backends. SkePU will allow certain such functions to be called from a user function.
  **Examples**: sin(x), pow(x, e)
```cpp
auto vec_sum = Map<2>([](int a, int b)
{
    return a + b;
});

// ...

vec_sum(result, v1, v2);
```
Compilation options
Compilation

- Precompiler options
  - `skepu-tool -name map_precompiled map.cpp -dir bin -opencl -- [Clang flags]`
- Handled by Makefiles
  - Makefile.in
    - Set up your system configuration, e.g. backend compiler
  - Makefile.include
    - Set up backends
  - Makefile
    - Set up programs
auto spec = skepu2::BackendSpec{skepu2::Backend::typeFromString(argv[2])};

- Sets the backend from a string, must match any of:
  - "CPU"
  - "OpenMP"
  - "OpenCL"
  - "CUDA"
Smart Containers
• `container.updateHost();`
  
  • Flushes and ensures that the CPU buffer contains up-to-date values.

• `container[i] = value;`
  
  • Managed element access.Flushes if needed.

• `container(i) = value;`
  
  • Direct element access into the raw CPU buffer.
Map
• Three groups of user function parameters:
  
  • **Element-wise**
    Only one element per user function call
  
  • **Random-access containers**
    Replicated for each memory space (e.g. GPUs)
    Proxy types Vec<T> and Mat<T> in user function
  
  • **Uniform scalars**
    Same values everywhere

• Argument groups are variadic (flexible count, including 0)

• Above order must be obeyed (element-wise first etc.)

• The parallelism/number of user function invocations is always determined by the return container (first argument), also in case of element-wise arity of 0.

• Also applies to **MapReduce, MapOverlap**!
• Optionally, use iterators with Map (and most other skeletons)

  • mapper(r.begin(), r.end(), v1.begin(), v2.begin());
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);
}
Reduce
Reduce Modes

- **1D Reduce**
  - Regular Vector
  - Matrix RowWise (returns Vector)
  - Matrix ColWise (returns Vector)

- **"2D" Reduce**
  - Regular Matrix (treated as a vector)

```
instance.setReduceMode(ReduceMode::RowWise) // default
```

```
instance.setReduceMode(ReduceMode::ColWise)
```

- `instance.setStartValue(value)`
  - Set Reduction start value. Defaults to 0-initialized.
`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);
}
float plus_f(float a, float b)
{
    return a + b;
}

float max_f(float a, float b)
{
    return (a > b) ? a : b;
}

auto max_sum = skepu2::Reduce(plus_f, max_f);

max_sum.setReduceMode(skepu2::ReduceMode::RowWise);
r = max_sum(m);
MapReduce
• `instance.setDefaultSize(size_t)`

  • When the element-wise arity is 0, this controls the number of user function invocations (That is, the size of the "virtual" temporary container in between the Map and Reduce steps)

• `instance.setStartValue(value)`

  • Set Reduction start value. Defaults to 0-initialized.
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);
}
Scan
• `instance.setScanMode(mode)`

  • Set the scan mode:
    `ScanMode::Inclusive` (default)
    `ScanMode::Exclusive`

• `instance.setStartValue(value)`

  • Set start value of scans. Defaults to 0-initialized.
```cpp
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);
}
```
MapOverlap
MapOverlap Modes

- **1D MapOverlap**
  - With 1D user function

- **Regular Vector**

- **Matrix RowWise**
  - `instance.setOverlapMode(Overlap::RowWise)` // default

- **Matrix ColWise**
  - `instance.setOverlapMode(Overlap::ColWise)`

- **2D MapOverlap**

- **Regular Matrix**

- **Separable MapOverlap (2D-with-1D)**

- **Matrix RolColWise**
  - `instance.setOverlapMode(Overlap::RowColWise)`

- **Matrix ColRowWise**
  - `instance.setOverlapMode(Overlap::ColRowWise)`
• `instance.setOverlap(x, [y])`
  • Set overlap radius

• `instance.setEdgeMode(mode)`
  • `Edge::Pad`
    • `instance.setPad(pad)` – set value
  • `Edge::Duplicate` (default)
  • `Edge::Cyclic`
MapOverlap 1D Example

```cpp
float conv(
    int overlap, size_t stride,
    const float *v, const Mat<float> stencil, float scale
) {
    float res = 0;
    for (int i = -overlap; i <= overlap; ++i)
        res += stencil[i + overlap] * v[i*stride];
    return res / scale;
}

Vector<float> convolution(Vector<float> &v) {
    auto convol = MapOverlap(conv);
    Vector<float> stencil {1, 2, 4, 2, 1};
    Vector<float> result(v.size());
    convol.setOverlap(2);
    return convol(result, v, stencil, 10);
}
```
```cpp
float over_2d(
    int ox, int oy, size_t stride, const float *m,
    const skepu2::Mat<float> filter
)
{
    float res = 0;
    for (int y = -oy; y <= oy; ++y)
        for (int x = -ox; x <= ox; ++x)
            res += m[y*(int)stride+x] * filter.data[(y+oy)*ox + (x+ox)];
    return res;
}
```
SkePU 3 preview
• New skeleton
  • MapPairs
• Higher-dimensionality containers
  • Tensor3<T>, Tensor4<T>
• Multiple return values from user functions
  • Can already return by struct: array-of-records format
  • New feature allows result in multiple separate arrays
• Improved MapOverlap user-function syntax
• And more to come!
```cpp
int uf(int a, int b) { return a; } // ...

auto pairs = skepu::MapPairs<1, 1>(uf);
skepu::Vector<int> v1(Vsize, 3), h1(Hsize, 7);
skepu::Matrix<int> res(Vsize, Hsize);
pairs(res, v1, h1);
```
float over_1d(skepu::Region1D<float> r, int scale)
{
    return (r(-2)*4 + r(-1)*2 + r(0) + r(1)*2 - r(2)*4) / scale;
}

float over_2d(skepu::Region2D<float> r, const skepu::Mat<float> stencil)
{
    float res = 0;
    for (int i = -r.oi; i <= r.oi; ++i)
    {
        for (int j = -r.oj; j <= r.oj; ++j)
        {
            res += r(i, j) * stencil(i + r.oi, j + r.oj);
        }
    }
    return res;
}
SkePU in Current Research
SkePU in Teaching