Westfälische
Wilhelms-Universität
Münster

> SkelCL – A Portable Skeleton Library for High-Level GPU Programming

Michel Steuwer, Philipp Kegel, and Sergei Gorlatch

wissen.leben
WWU Münster

Group Parallel and Distributed Systems
Department of Computer Science
University of Münster, Germany

- Programming approaches for Graphics Processing Units (GPU)

CUDA | OpenCL

Programming challenges:

- coordinate thousand of threads
- explicit data transfers to and from GPU
- exploit complex GPU memory hierarchy manually

Additional challenges for multi-GPU systems:

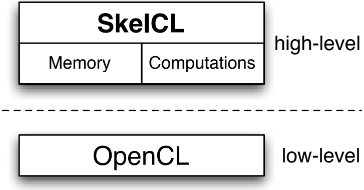
- keep all GPUs busy
- perform data transfers between GPUs

⇒ low-level coding makes GPU programming complex and error-prone

Idea Provide high-level abstractions to simplify GPU programming

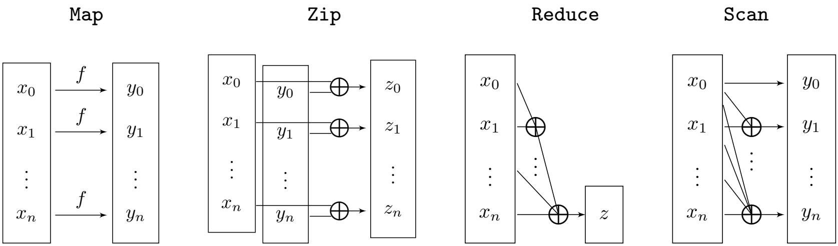
SkelCL – Overview

- *SkelCL* is a high-level library for single- and multi-GPU computing



- Built on top of OpenCL ⇒ hardware-independent and portable
- Two high-level features:
  - Memory: implicit management using *abstract vector data type*
  - Computations: conveniently expressed using *pre-implemented parallel patterns*
- Goals:
  - Ease GPU programming by providing high-level abstractions
  - Eliminate explicit data transfers
  - Especially address multi-GPU systems

- User expresses computations using pre-implemented parallel patterns, a. k. a. *algorithmic skeletons*
- Skeleton implementations are optimized for GPU
- User customizes skeletons by providing application-specific function
- Four common basic skeletons provided:



Abstract Data Type

- Abstract vector data type makes memory accessible by CPU and GPU
- For convenience:
  - Memory is allocated automatically on the GPU
  - **Implicit data transfers** between the main memory and the GPU memory

- Skeletons accept vectors as input and output
- Skeletons automatically ensures: input vectors' data available on GPU

- The output vector's data is not copied to CPU but resides in GPU memory
- This **lazy copying minimizes data transfers** ⇒ Improved performance

  Example:
  - Output vector is used as input to another skeleton ⇒ no data transfer needed

SkelCL – First Example
Dot product

- Calculation of the dot product of two vector *a* and *b*: $\sum_{i=0}^{size-1} a_i \cdot b_i$

```
float dot_product(const std::vector<float>& a,
                  const std::vector<float>& b) {
  SkelCL::init(); // initialize SkelCL

    // declare computation:
  SkelCL::Zip<float>    mult(
      "float func(float x, float y){ return x*y; }");
  SkelCL::Reduce<float> sum_up(
      "float func(float x, float y){ return x+y; }");

    // create data vectors:
  SkelCL::Vector<float> A(a.begin(), a.end()),
                        B(b.begin(), b.end());

    // perform calculation:
  SkelCL::Vector<float> C = sum_up( mult(A, B) );
  return C.front(); // access result
}
```

- SkelCL: 6 lines of code
- OpenCL: 68 lines of code (NVIDIA programming example)

Extension: Additional Arguments

- Usual skeletons have fixed number of arguments
- SkelCL extends this:
  - An arbitrary number of arguments can be passed to the customizing function
  - ⇒ Enable more algorithms to be expressed using skeletons
- SAXPY calculation ($Y = a * X + Y$) with zip skeleton as example:

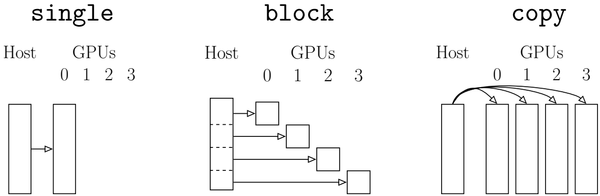
```
/* create skeleton with three arguments */
Zip<float> saxpy (
 "float func(float x,float y,float a) { return a*x+y; }" );

 /* create input vectors */
 Vector<float> X(SIZE); fillVector(X);
 Vector<float> Y(SIZE); fillVector(Y);
 float a = fillScalar();

 /* execute skeleton, pass additional argument (a) */
 Y = saxpy( X, Y, a );
```

Multi-GPU

- Programming multi-GPU systems is especially complex
- Main challenges:
  - Data distribution among GPUs
  - Data exchange between GPUs
- To address this, SkelCL supports three different *distributions*:

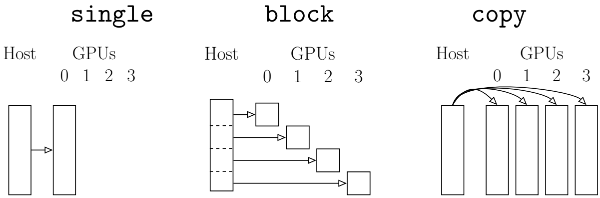


- Changing distribution at runtime triggers data exchange. Example:

```
vector.setDistribution(Distribution::block);
```

- **All required data transfers are performed automatically by SkelCL!**

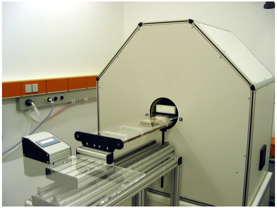
Skeletons and Multi-GPU

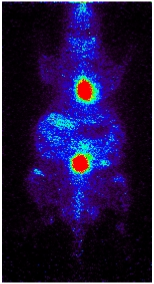


- Distribution of input vector implies the parallelization:
  - `single` ⇒ skeleton is executed on a single GPU
  - `block` ⇒ all GPUs cooperate in skeleton execution
  - `copy` ⇒ skeleton is executed on all GPUs separately
- User does not have to set distribution explicitly
- For convenience SkelCL automatically sets a default distribution

Application Study: Tomography

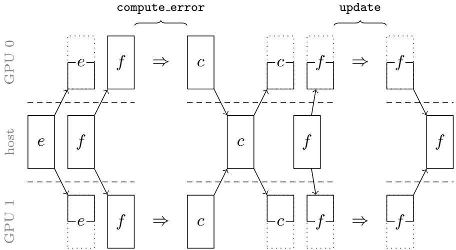
- Application study: *List-Mode Ordered Subset Expectation Maximization* (list-mode OSEM)
- List-mode OSEM is a time-intensive image reconstruction algorithm
- Up to several hours on common PCs ⇒ not practical
- 3D-images are reconstructed from sets of *events* recorded by a scanner; events are split into *subsets* which are processed iteratively
- For every subset, two steps are performed:
  - All events are used to process an *error image* (c)
  - The error image is then used to update a *reconstruction image* (f)





List-mode OSEM

- The two steps require different parallelization approaches:
  - `compute_error`: divide `events` (e) across processing units, every processing unit requires copy of error image (c) and reconstruction image (f)
  - `update`: divide error image (c) and reconstruction image (f)



- In a multi-GPU system multiple data exchanges are required every iteration

List-mode OSEM in SkelCL

- In SkelCL we can easily express the distribution of the different vectors
- Data movement is performed automatically by SkelCL

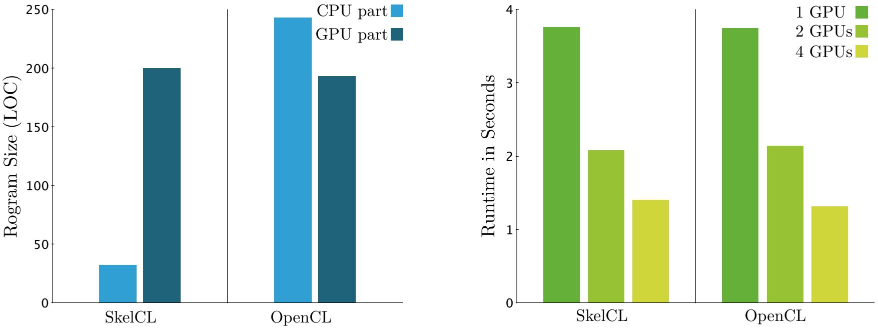
```
for (l = 0; l < num_subsets; l++) {
  SkelCL::Vector<Event> events = read_events(l);
  events.setDistribution(Distribution::block); // divide events
  f.setDistribution(Distribution::copy); // copy recon. image
  c.setDistribution(Distribution::copy); // copy error image

  // map skeleton
  compute_error_image(index, events, events.sizes(), f, out(c));

  f.setDistribution(Distribution::block);    // change distribution
  c.setDistribution(Distribution::block, add);

  // zip skeleton
  update_reconstruction_image(f, c, f);
}
```

List-mode OSEM Results



- Lines of code for the CPU part was drastically reduced: **from 249 to only 32**
- SkelCL only introduces a moderate overhead of **less than 5%**

Summarizing SkelCL

- *SkelCL* is a library for high-level (multi-)GPU programming
- Skeletons implicitly express parallelism calculations on the GPU
- Skeletons are flexible due to the ability to pass *additional arguments*
- *Abstract vector data type* implicitly transfers data to and from GPU
- *Distributions* simplify parallelization across multiple GPUs
- Experiments show that SkelCL implements real-world application with:
  - minor overhead as compared to OpenCL (5% in performance)
  - significantly higher level of programming (over 85% reduction in LOCs)

Related projects using skeletons:

- SkePU (J. Emmyren and C. Kessler, University Linköping, Sweden)
    - Generates CPU, OpenCL or CUDA code
    - No additional arguments
    - No data distribution
- Thrust (J. Hoberock and N. Bell, NVIDIA Research)
    - Only works with CUDA
    - No unified memory management
    - No multi-GPU
    - No data distribution