Westfälische
Wilhelms-Universität
Münster

> Towards High-Level Programming of Multi-GPU Systems Using the SkelCL Library

Michel Steuwer, Philipp Kegel, and Sergei Gorlatch
University of Muenster, Germany

wissen.leben
WWU Münster

Motivation

- Popular programming approaches for Graphics Processing Units (GPUs):

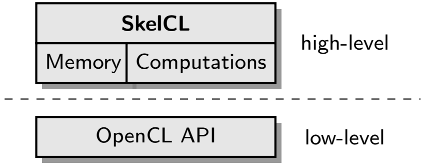
- **Challenges when using OpenCL or CUDA:**
  - explicit coordination of thousands of threads
  - explicit data transfers to and from GPUs
  - explicit handling of complex memory hierarchies
- **Additional challenges for multi-GPU systems:**
  - explicit work balancing to keep all GPUs busy
  - explicit managing of data transfers between GPUs

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

Idea Provide high-level abstractions to simplify programming

SkelCL – Overview

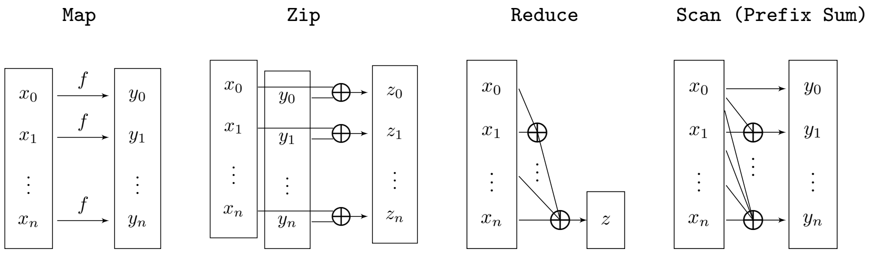
- *SkelCL* is a library introducing high-level abstractions on top of OpenCL



- Built on top of OpenCL:
  - hardware- and vendor-independent, portable
  - access to arbitrary OpenCL *devices*, e. g. GPUs or multi-core CPUs
- Two high-level features:
  - Computations: conveniently expressed using *pre-implemented parallel patterns*
  - Memory: implicitly managed using *abstract vector data type*
- Goals:
  - Simplify programming by providing high-level abstractions
  - Eliminate explicit data transfers
  - Especially address multi-GPU systems

Algorithmic Skeletons

- User expresses computations using pre-implemented parallel patterns, a. k. a. *algorithmic skeletons*
- Skeletons are customized by application-specific functions
- Four basic skeletons currently provided (f and $\oplus$ application-specific)



Abstract Data Type

- Abstract vector data type makes memory accessible by CPU and GPU
- For programmer's convenience:
    - Memory is allocated automatically on the GPU
    - **Implicit data transfers** between the main memory and the GPU memory
- Vectors are used as input and output for skeletons
- SkelCL automatically ensure: input vectors' data are available on GPU
- We use **lazy copying** to **minimizes data transfers**:
  Data is not transfered right away, but only when needed

  Example: Output vector is used as input to another skeleton
    - The output vector's data is not copied to host but resides in device memory

  $\Rightarrow$ no data transfer needed, which leads to **improved performance**

SkelCL – First Example Dot product

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

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

    // declare computation by customizing skeletons:
  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: 7 lines of code**
- **OpenCL: 68 lines of code** (NVIDIA programming example)

Extension: Additional Arguments

- Traditionally, skeletons have fixed number of arguments
- SkelCL extends this:
  - An arbitrary number of arguments can be passed to the skeleton
  - ⇒ Enables more algorithms to be expressed using skeletons

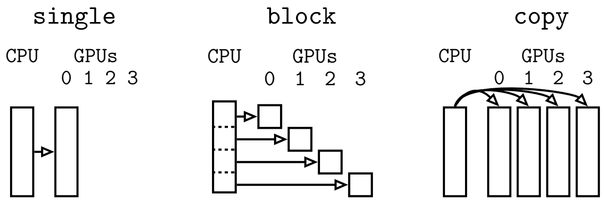
Example: SAXPY calculation in BLAS ($Y = a * X + Y$)

- Can be easily expressed using the zip skeleton
- Scalar a is required in the computation and passed as additional argument:

```
/* create skeleton with one additional argument */
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 );
```

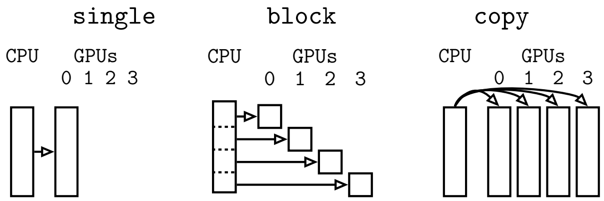
Programming Multi-GPU Systems

- Programming multi-GPU systems is especially complicated:
  - explicit distribution of data among GPUs
  - explicit data exchange between GPUs
- To address this, SkelCL supports three *data distributions*:



- Distribution of input vector implies automatic parallelization:
  - `single` $\Rightarrow$ skeleton is executed on a single GPU
  - `block` $\Rightarrow$ all GPUs cooperate in skeleton execution
  - `copy` $\Rightarrow$ skeleton is executed on all GPUs separately

Programming Multi-GPU Systems



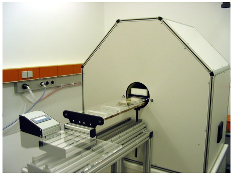
- Distribution is either set by programmer or by default
- Changing distribution at runtime ⇒ automatic data exchange. e.g.:

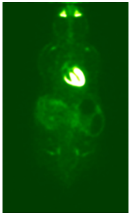
```
// set single as intitial distribution
vector.setDistribution(Distribution::single);
...
// changing from single to block distribution
vector.setDistribution(Distribution::block);
```

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

Application Study: Tomography

- Application study: *List-Mode Ordered Subset Expectation Maximization* (list-mode OSEM)
- List-mode OSEM[1]is a time-intensive iterative image reconstruction algorithm for computer tomography
- 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)
- Up to several hours on a common PC $\Rightarrow$ not practical

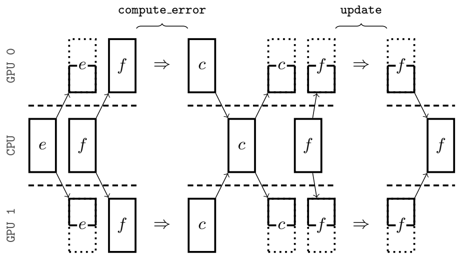




[1] T. Kösters et al. EMrecon: An expectation maximization based image reconstruction framework for emission tomography data. NSS/MIC Conference Record, IEEE, 2011

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)
- Data partitioning and data transfers between CPU and two GPUs:



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

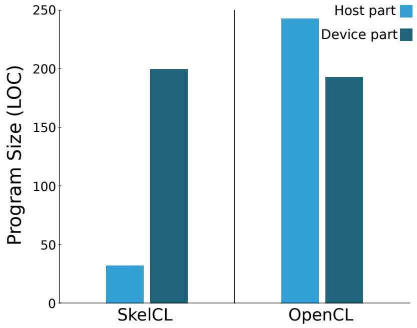
List-mode OSEM in SkelCL

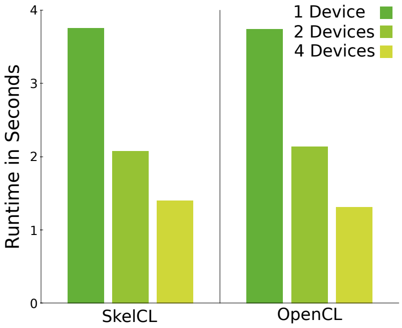
- We can easily express the identified distribution of data in SkelCL:

```
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);
}
```

- **All data movements are performed automatically by SkelCL**

Experimental Results





- LOC for the host part was drastically reduced: **from 249 to only 32**
- Runtime overhead of SkelCL is **less than 5%**

Conclusion

- *SkelCL*: a high-level programming library for single- and multi-GPU systems
- Skeletons implicitly express parallel calculations on GPUs
  $\Rightarrow$ No explicit coordination of thousands of threads
  $\Rightarrow$ No explicit handling of the complex memory hierarchies
- Skeletons are flexible due to the ability to pass *additional arguments*
- *Abstract vector data type* implicitly transfers data to and from the device
  $\Rightarrow$ No explicit data transfers to and from GPUs
- *Distributions* simplify parallelization across multiple GPUs
  $\Rightarrow$ No explicit managing of data transfers between GPUs
- Experiments show minor overhead and significantly shorter codes

SkelCL is open-source and available at:
`http://skelcl.uni-muenster.de`

What is next?

- Fully support heterogeneous systems

  Advantage We built on top of OpenCL
  $\Rightarrow$ SkelCL already can use *every* OpenCL device

  Challenges
  - Find fair work balancing between different compute devices
  - Optimize skeleton implementations for different devices

- Add two-dimensional data type

- Integrate more skeletons