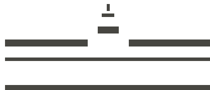


> Towards High-Level Programming for Systems with Many Cores

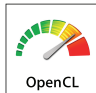
Sergei Gorlatch and Michel Steuwer

wissen.leben
WWU Münster

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

Many-Cores and Programming

- **Many-cores:** Multi-core CPUs + GPUs $\Rightarrow$ $10^2 - 10^4 - 10^6$ **cores**
- **State-of-the-art programming** for many-cores:



- **Challenges on a system with just one GPU:**
  - coordination of hundreds/thousands of work-items ($\approx$ threads)
  - data transfers to and from GPU
  - handling of the complex memory hierarchy
- **Additional challenges for multi-GPU systems:**
  - work balancing to keep all GPUs busy
  - managing of data transfers between GPUs

$\Rightarrow$ **Two major drawbacks of the state-of-the-art approaches:**
  - explicit, low-level coding produces lengthy, error-prone programs
  - missing formal base hinders optimizations through code transformations, performance prediction, reasoning and verification, etc.

Our approach: SkelCL (Skeleton Computing Language) –
a high-level programming model on top of OpenCL

Advantages of building on top of OpenCL:

- hardware- and vendor-independent, portable
- access to arbitrary OpenCL *device*, multi-core CPUs, GPUs, and other accelerators (Cell, FPGA, . . .)

Advantages of high-level constructs:

- shorter and better structured codes
- formal semantics => transformations and performance modeling

Three high-level mechanisms in SkelCL:

- *Parallel container data types* for automatic memory management
- *Data (re)distributions* for automatic data exchange between multiple GPUs
- *Parallel patterns (skeletons)* for expressing parallel computations

- **Container data types (Vector and Matrix)** make memory management implicit for both CPU and GPUs in the system

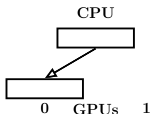
- For programmer's convenience:
  - **Memory is allocated automatically** on the GPU
  - **Automatic data transfers** between the host and the GPU memory

- We use **lazy copying** to **minimize data transfers**:
  - Data is not transfered right away, but rather only when needed
  - *Example:* Output vector is used as input to another computation
  - The output vector's data is not copied to host but resides in device memory

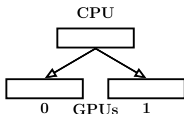
  $\Rightarrow$ no data transfer needed, which leads to **improved performance**

(Re)Distribution Mechanisms

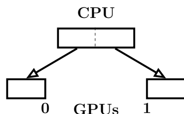
For partitioning data across multiple GPUs, there are four *data distributions*:



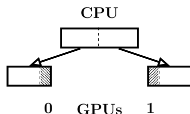
(a) *single*



(b) *copy*



(c) *block*



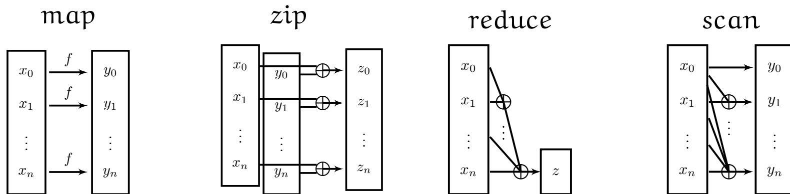
(d) *overlap*

- The distributions are either chosen by the programmer, or SkelCL automatically chooses default distributions
- Distributions for vector shown here, same distributions exist for matrix
- Changing distribution at runtime triggers automatic data exchange, e. g.:

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

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

- The programmer expresses computations using pre-implemented parallel patterns, a. k. a. *algorithmic skeletons* (higher-order functions)
- Skeletons are customized by application-specific functions
- Four basic (Map, Zip, Reduce, Scan) and three specialized (MapOverlap, Stencil, Allpairs) skeletons are currently provided



- *Example:* Calculation of the vector dot product expressed with skeletons:

$$\mathrm{dotProd}(a, b) = \sum_{k=1}^{d} a_k \cdot b_k = \mathrm{reduce}(+)\Big(\mathrm{zip}(\cdot)(a, b)\Big)$$

Calculation of the dot product: $\sum_{k=1}^{d} a_k \cdot b_k = \text{reduce}(+)\Big(\text{zip}(\cdot)(a, b)\Big)$

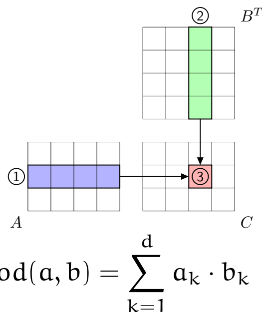
```
using namespace skelcl;
float dot_product(const std::vector<float>& a,
                  const std::vector<float>& b) {
 skelcl::init(); // initialize SkelCL

   // declare computation by customizing skeletons:
 Zip<float> mult("float f(float x, float y){ return x*y; }");
 Reduce<float> sum_up("float f(float x,float y){return x+y;}","0");

   // create data vectors:
 Vector<float> A(a.begin(), a.end());
 Vector<float> B(b.begin(), b.end());
   // perform calculation:
 Vector<float> C = sum_up( mult(A, B) );
 return C.front(); // access result
}
```

SkelCL: 7 lines of code vs. **OpenCL: 68 lines of code** (NVIDIA example)

- *Allpairs computations:*
  The same computation is performed for all possible pairs of vectors from two matrices
- Possible applications: N-body simulation, matrix multiplication, etc.
- Matrix multiplication expressed using allpairs:



$$A \times B = \mathrm{allpairs}(\mathrm{dotProd})(A, B^T), \text{ where } \mathrm{dotProd}(a, b) = \sum_{k=1}^{d} a_k \cdot b_k$$

```
Allpairs<float> mm(
 "float func(float_matrix_t a, float_matrix_t b) {\
  float c = 0.0f;\
  for (int i = 0; i < width(a); ++i) {\
    c += getElementFromRow(a, i) * getElementFromCol(b, i); }\
  return c; }");
Matrix<float> result = mm(A, B);
```

Specialization rules enable optimizations of skeleton implementations

Proposition. If the customizing function of the $allpairs$ skeleton can be expressed as a seq. composition of zip and $reduce$, then an optimized GPU implementation can be automatically derived

- Example matrix multiplication: $A \times B = allpairs(dotProd)(A, B^T)$, where

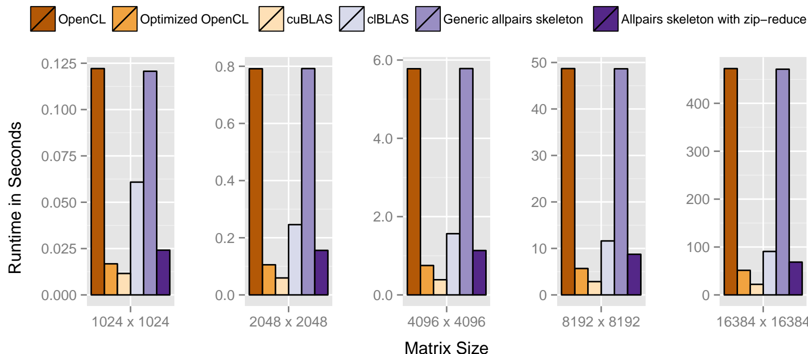
$$dotProd(a, b) = \sum_{k=1}^{d} a_k \cdot b_k = reduce(+)\Big(zip(\cdot)(a, b)\Big)$$

```
Zip<float> mult("float f(float x,float y){return x*y;}");
Reduce<float> sum_up("float f(float x,float y){return x+y;}", "0");
Allpairs<float> mm(sum_up, mult);
Matrix<float> result = mm(A, B);
```

- Optimized implementation uses additional semantical information of zip and $reduce$ to make use of the fast local GPU memory

Runtime Results Matrix Multiplication

- NVIDIA System using one Tesla GPU with 240 streaming processors

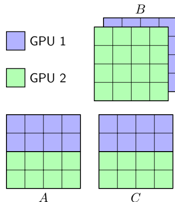


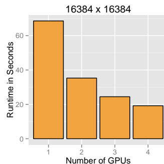
- Specialized allpairs implementation is ≈ **7 times faster** than first implementation, and close to the performance of BLAS implementations
- cuBLAS implementation is the fastest as it is highly tuned by the vendor, but restricted to matrix multiplication and to NVIDIA hardware

The Allpairs Skeleton using Multiple GPUs

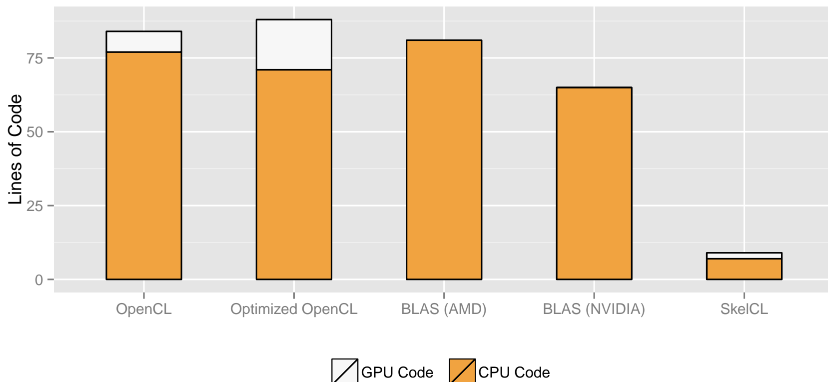
The allpairs skeleton works for multi-GPU systems as well

- SkelCL automatically divides the computation among GPUs using its *distribution* feature
- By using the semantics of the allpairs skeleton:
  - Matrix A and C are *block* distributed, i. e. row-divided across GPUs
  - Matrix B is *copy* distributed, i. e. copied entirely to all GPUs
- The distributions are selected automatically
  ⇒ **No additional lines of code necessary**
- Good scalability:
  Four GPUs are **3.57** faster than one GPU





Programming effort
Matrix Multiplication



- SkelCL: 9 lines vs. BLAS: 65 and 81 lines vs. OpenCL 84 and 88 lines

Skeletons for Stencil Computations

Two new skeletons for supporting stencil computations: *MapOverlap* and *Stencil*

- Both skeleton are similar to Map: executes given function for every element
- But customizing function can take neighboring elements in into account
- Application developer provides:
  - The customizing function
  - A description of the stencil shape
  - Out-of-bound handling: accesses returns either neutral or nearest value
- *MapOverlap* skeleton for simple stencil applications
- *Stencil* skeleton for more complex iterative stencil applications

SkelCL provides fast multi-GPU ready implementations of these skeletons.

Sobel edge detection with *MapOverlap*

- Produces an output image marking all edges in the input image white
- Basic idea: Search for differences in color as compared to neighboring pixels

Original "'Lena"' image

Output of the sobel edge detection

SkelCL implementation:

```
MapOverlap<char(char)> m(
"char func(const char* img) {
  short h = -1*get(img,-1,-1)
            +1*get(img,+1,-1)
            -2*get(img,-1, 0)
            +2*get(img,+1, 0)
            -1*get(img,-1,+1)
            +1*get(img,+1,+1);
  short v = // ...
  return sqrt(h*h + v*v);
}", 1, Padding::NEUTRAL, 0);

// execution of the skeleton
Matrix<char> out_img = m(img);
```

Sobel edge detection: SkelCL implementation

Application is a perfect fit for the MapOverelap skeleton

Sequential implementation:

(boundary checks omitted)

```
for (i = 0; i < width; ++i)
 for (j = 0; j < height; ++j)
  h = -1*img[i-1][j-1]
      +1*img[i+1][j-1]
      -2*img[i-1][j  ]
      +2*img[i+1][j  ]
      -1*img[i-1][j+1]
      +1*img[i+1][j+1];
  v = // ...
  out_img[i][j]=sqrt(h*h+v*v);
```

SkelCL implementation:

```
MapOverlap<char(char)> m(
"char func(const char* img) {
   short h = -1*get(img,-1,-1)
             +1*get(img,+1,-1)
             -2*get(img,-1, 0)
             +2*get(img,+1, 0)
             -1*get(img,-1,+1)
             +1*get(img,+1,+1);
   short v = // ...
   return sqrt(h*h + v*v);
}", 1, Padding::NEUTRAL, 0);

// execution of the skeleton
Matrix<char> out_img = m(img);
```

- SkelCL implementation is very similar to the sequential version (**good!**)

Sobel edge detection: OpenCL implementation

- OpenCL implementation requires additional low-level code, like:
  - Knowledge and use of OpenCL keywords and functions
  - Boundary checks
  - Pointer arithmetic

```
__kernel void sobel_kernel(__global const char* img,
                           __global       char* out_img,
                           int w, int h ) {
 size_t i = get_global_id(0);   size_t j = get_global_id(1);

 if(i < w && j < h) { // perform boundary checks manually
  char ul = (j-1 > 0 && i-1 > 0) ? img[((j-1)*w)+(i-1)] : 0;
  char um = (j-1 > 0           ) ? img[((j-1)*w)+(i+0)] : 0;
  // ... 7 more of these lines

  out_img[j * w + i] = computeSobel( ul, um, ..., );  } }
```

OpenCL requires almost five times more lines of code than SkelCL (19 vs. 4)

More complex stencils with *Stencil*

- Iterative applications (e.g. simulations) can be performed using the Stencil skeleton
- Non-square stencil shapes can be expressed
- *Example*: Simulation of heat transfer

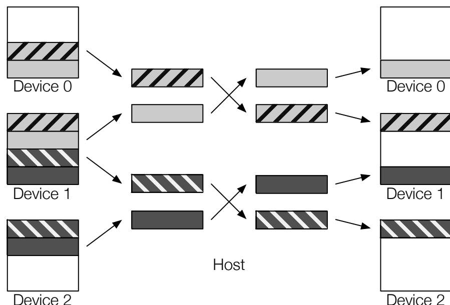
```
Stencil <char(char)> heatSim(
  "char func(const char* in) {
   char lt = get(in, -1, -1);
   char lm = get(in, -1,  0);
   char lb = get(in, -1, +1);
   return computeHeat(lt,lm,lb); }",
  StencilShape(1, 0, 1, 1),
  Padding::NEUTRAL, 255);
output = heatSim(100, input);
```

- Often applications consist of multiple stencils
- *Example*: Canny algorithm for more advanced edge detection

```
Stencil<Pixel(Pixel)> gauss(...);
Stencil<Pixel(Pixel)> sobel(...);
Stencil<Pixel(Pixel)> nms(...);
Stencil<Pixel(Pixel)> threshold(..);

StencilSequence<Pixel(Pixel)> canny(
  gauss, sobel, nms, threshold);
output = canny(1, input);
```

- Automatic support for multi-GPU Systems
- Using the *overlap distribution* elements on the "border" are stored on 2 GPUs
- Data exchange is necessary between iterations
- Currently implemented by copying data to the CPU and back to the GPUs

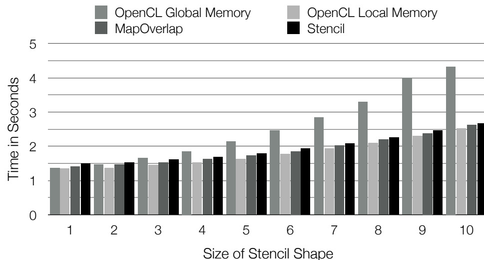


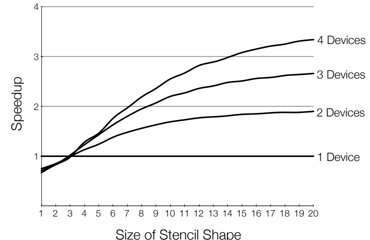
- **Everything is done automatically. No change of source code necessary!**

Implementation & Performance

- Both skeletons automatically use fast local memory
- Implementation of the MapOverlap skeleton avoids some out-of-bound accesses by extending the input data
- For Stencil skeleton all out-of-bound handling is done on GPU to support sequences of Stencils (with different handling modes)

Performance for Gaussian Blur:

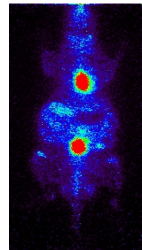




Application: Computer Tomography

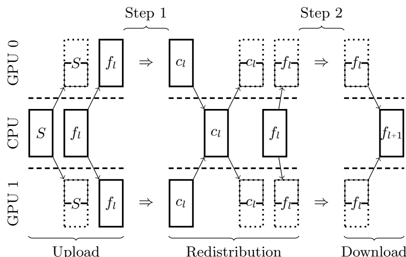
- Application study: *List-Mode Ordered Subset Expectation Maximization* (LM OSEM)
- LM OSEM is a time-intensive image reconstruction algorithm, takes hours on a PC $\Rightarrow$ not practical

- 3D-images are reconstructed from sets of *events* recorded by a scanner; events are split into *subsets*, processed iteratively
- In every iteration a subset is processed in two steps:
  - Subset's events (S) are used to process an *error image* (c)
  - The error image is used to update a *reconstruction image* (f)



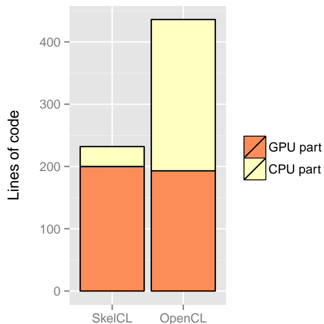
Parallelizing LM OSEM

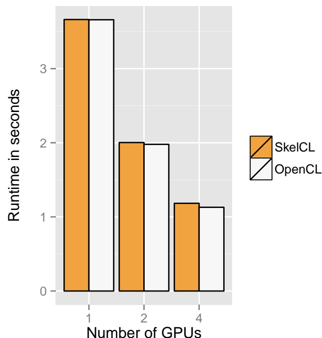
- The two computational steps require different parallelization approaches:
  - *Step 1*: divide subset's events (S) across processing units, every processing unit requires copy of reconstruction image (f) to compute an error image (c)
  - *Step 2*: divide error image (c) and reconstruction image (f) to refine the reconstruction image



- In SkelCL:
  - S, f, and c are expressed as SkelCL vectors
  - Step 1 and Step 2 are expressed using algorithmic skeletons
  - Distribution and redistribution of data is easily expressed in SkelCL

LM OSEM Results





- Lines of code for the CPU part was drastically reduced: **from 243 to only 32**
- SkelCL only introduces a moderate overhead of **less than 5%**

- *SkelCL* is a high-level programming model for (multi-)GPU programming
- Three high-level features: *Container data types*; *Distributions*; *Skeletons*
- *Container data types* implicitly transfer data to and from the devices
  $\Rightarrow$ No explicit data transfers to and from GPUs
- *Distributions* simplify parallelization across multiple GPUs
  $\Rightarrow$ No explicit managing of data transfers between GPUs
- Skeletons implicitly express parallel calculations on GPUs
  $\Rightarrow$ No explicit coordination of thousands of threads
  $\Rightarrow$ No explicit handling of the complex memory hierarchies
- Experiments show significant shorter code with competitive performance
- Our implementation of SkelCL is an open source C++ library available at `http://skelcl.uni-muenster.de`