High-Level Programming of Multi-GPU Systems

Michel Steuwer, jointly with Philipp Kegel and Sergei Gorlatch
Goal: Easily exploit all GPUs in a multi-GPU System

- State of the art GPU programming models:
  - OpenCL
  - CUDA
  - NVIDIA

- Challenges when using OpenCL or CUDA:
  - explicit coordination of thousands of work-items (∝ threads)
  - explicit data transfers to and from GPUs
  - explicit handling of complex memory hierarchy

- 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
• State of the art programming models for **distributed systems** with GPUs:

  ![OpenCL, CUDA, NVIDIA, MPI]

• **Challenges when mixing programming models:**
  - explicit implementation of a single logical data transfer as multiple steps in different programming models
  - explicit special handling of global operations (e.g., broadcast) in different programming models
  - explicit synchronization of execution status of different programming models to avoid race conditions

**Our approach:** high-level, uniform, on top of OpenCL

**SkelCL:** library of high-level abstractions for simplify GPU computing

**dOpenCL:** middleware for transparent distributed GPU programming

**SkelCL + dOpenCL:** High-level programming of distributed systems with GPUs
SkelCL is a high-level programming model based on 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, . . .)

SkelCL offers three high-level mechanisms:
- Parallel container data types for automatic memory management
- Data (re)distributions for automatic data exchange between multiple GPUs
- Parallel skeletons for simplified parallel programming

Our implementation of the SkelCL programming model is an open source C++ library available at http://skelcl.uni-muenster.de
Container data types (Vector and Matrix) make memory accessible by both CPU and GPUs in the system.

For programmer’s convenience:
- Memory is allocated automatically on the GPU
- Implicit data transfers between the CPU memory and the GPU memory

We use lazy copying to minimize data transfers: Data is not transferred right away, but 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
  ⇒ no data transfer needed, which leads to improved performance
For partitioning data across multiple GPUs, there are four *data distributions*:

- **single**
- **copy**
- **block**
- **overlap**

- User sets distributions, 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!
Parallel Skeletons

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

map

\[
x_0 \xrightarrow{f} y_0 \\
x_1 \xrightarrow{f} y_1 \\
\vdots \\
x_n \xrightarrow{f} y_n
\]

zip

\[
x_0 \quad y_0 \quad z_0 \\
x_1 \quad y_1 \quad z_1 \\
\vdots \\
x_n \quad y_n \quad z_n
\]

reduce

\[
x_0 \rightarrow y_0 \\
x_1 \rightarrow y_1 \\
\vdots \\
x_n \rightarrow y_n
\]

scan

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

\[
\text{dotProd}(a, b) = \sum_{k=1}^{d} a_k \cdot b_k = \text{reduce}(+)\left(\text{zip}(\cdot)(a, b)\right)
\]
Calculation of the dot product: \[ \sum_{k=1}^{d} a_k \cdot b_k = \text{reduce}(+)\left(\text{zip}()\left(a, b\right)\right) \]

```cpp
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(float, float)> mult("float func(float x, float y) { return x*y; }");
    Reduce<float(float, float)> sum_up("float func(float x, float y) { return x+y; }", "0.0f");

    // create data vectors:
    Vector<float> A(a.begin(), a.end()), 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)
Sobel edge detection with the MapOverlap Skeleton

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

**MapOverlap** skeleton performs a computation for every pixel

In addition access to neighbouring pixels is allowed

**SkelCL implementation:**

```cpp
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);
Matrix<char> out_img = m(img);
```
Sobel edge detection: SkelCL implementation

Application is a perfect fit for the MapOverlap skeleton

Sequential implementation:
(boundary checks omitted)

```c
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:

```c
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);

Matrix<char> out_img = m(img);
```

SkelCL implementation is very similar to the sequential version (good!)
OpenCL kernel requires five times more lines of code than SkelCL (19 vs. 4)
• **Allpairs computations:**
  The same computation is performed for all possible pairs of vectors from two matrices

• Possible applications: N-body simulations, matrix multiplication, etc.

• **Example:** Matrix multiplication expressed using `allpairs`:

\[
A \times B = \text{allpairs} (\text{dotProd})(A, B^T), \text{ where}
\]

\[
dotProd(a, b) = \sum_{k=1}^{d} a_k \cdot b_k = \text{reduce}(+) \left( \text{zip}(\cdot)(a, b) \right)
\]
We can express matrix multiplication using *allpairs* and the dot product using *zip* and *reduce*:

```cpp
skelcl::init();

Zip<float(float, float)> mult
  ("float func(float x, float y) { return x*y; }")

Reduce<float(float, float)> sum_up
  ("float func(float x, float y) { return x+y; }")

Allpairs<float(float, float)> mm(sum_up, mult);

Matrix<float> A(n, d); fill(A);
Matrix<float> B(d, m); fill(B);
Matrix<float> C = mm(A, B);
```

Matrix multiplication is easily expressed in only **9 lines of code**!
• Our \textit{allpairs} implementation is \(\approx 7\) \textbf{times faster} than a naive OpenCL implementation, and close to the performance of BLAS implementations.

• \textit{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 works for multi-GPU systems as well

- SkelCL automatically divides the computation among GPUs using its *distribution* feature
- 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
Medical Imaging Application

- Application study: *List-Mode Ordered Subset Expectation Maximization* (LM OSEM)

- LM OSEM is a time-intensive image reconstruction algorithm

- Up to several hours on common PCs \(\Rightarrow\) not practical

- 3D-images are reconstructed from sets of *events* recorded by a scanner; events are split into *subsets* which are 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 then used to update a *reconstruction image* (\(f\))
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
• 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%
GPUs are increasingly used in distributed systems, e.g., modern supercomputers. **Goal:** also use SkelCL for distributed systems with GPUs

**Problem:** **OpenCL** is limited to stand-alone systems

**Our approach:** **dOpenCL** - implementation of OpenCL for distributed systems

- Preserves uniform programming model
- Allows SkelCL skeletons to automatically use devices of a distributed system
- Allows the programmer to mix skeletons with standard OpenCL (ad-hoc parallelism)
**Distributed OpenCL (dOpenCL):** Extended OpenCL implementation for distributed systems

**Main advantages of dOpenCL:**
- Transparently integrates distributed compute devices into OpenCL API → **compatible** with existing OpenCL applications
- Integrates arbitrary vendor-specific OpenCL implementations → easily **extendable** for arbitrary compute devices

Our implementation of dOpenCL is open source software available at http://dopencl.uni-muenster.de/
OpenCL does not prescribe connection type between host and compute devices → host and devices can be distributed over a network

Figure: König, B., OpenCL platform architecture, de.wikipedia.org/w/index.php?title=Datei:Platform_architecture_2009-11-08.svg
• OpenCL does not prescribe connection type between host and compute devices → host and devices can be distributed over a network

• dOpenCL extends OpenCL’s platform model for distributed systems:
  - **host**: runs *host program* which controls distributed execution
  - **compute node**: executes data-parallel functions (*kernels*) on devices

---

Footnote: Figure based on König, B., OpenCL platform architecture
**SkelCL** (available at http://skelcl.uni-muenster.de)
- High-level library built on top of OpenCL
- Skeletons implicitly express parallel calculations
- Vector data type implicitly transfers data to/from devices
- Distributions simplify parallelization across multiple devices

**dOpenCL** (available at http://dopencl.uni-muenster.de)
- Provides transparent OpenCL implementation for distributed systems
- OpenCL devices across the distributed system are locally available

Using SkelCL and dOpenCL together allows for high-level programming of distributed systems with multiple GPUs!