Expression Templates and OpenCL

Uwe Bawidamann,
Marco Nehmeier

Institute of Computer Science
University of Würzburg
Germany

PPAM 2011

Outline

Outline

Motivation
Computer Architecture

- A change in the design of computer architecture
  - Single-core $\Rightarrow$ multi-core
  - Circumvent physical constraints
  - Parallelize the computation

Motivation
Graphic Devices

- GPGPU
- Performance
- Highly parallel
- CUDA
  - NVIDIA
  - CUDA C, CUDA Fortran or OpenCL
- OpenCL
  - Open standard
  - C like programming language
  - CPU's, GPU's, and many other

GPGPU
Impact onto scientific computing

- Supercomputer
  - 3 of the 5 fastest supercomputer hosting CUDA
- Workstation and desktop computers
- Standard applications supporting CUDA or OpenCL

$\Rightarrow$ Write own application

GPU Programming

- New concepts
- New Techniques
- New pitfalls
  - Inappropriate data transfers
  - Misaligned memory access
  - Bank conflicts
  - Inappropriate load balancing
  - Deadlocks

Our Intention

- Investigate techniques and concepts
  - User-friendly libraries
  - Interface in C++
  - Hiding GPGPU paradigms
- Operator overloading
  - DSL
- Expression templates
  - Code generation
  - Optimize data transfer

Outline

Expression Templates

- Operator overloading

```
Vector v1, v2, v3;
...
Vector res = v1 + v2 + v3;
```

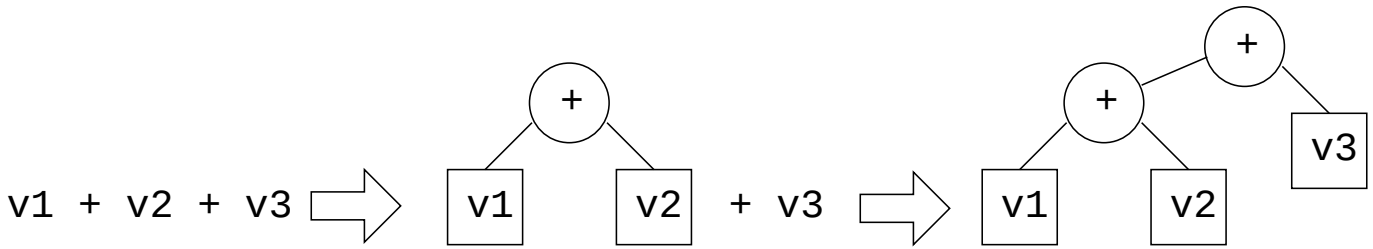
- Pairwise evaluation problem

```
...
Vector tmp = v1 + v2;
Vector res = tmp + v3;
```

- Additional loops
- Additional instances
  - `new`
  - `delete`

Expression Templates

- Introduced by Todd Veldhuizen
- Building parse trees
  - Operator overloading
  - Recursive templates
- Evaluate whole tree
  - Assignment
    - `operator=`
  - Optimized
    - Algorithm
    - Compiler



Expression Templates


```
template<typename T, class OP, typename A, typename B>
class BExpr {
    A const& a_;
    B const& b_;
  public:
    BExpr(A const& a, B const& b) : a_(a), b_(b) { }

    T operator[] (size_t i) const {
        return OP.eval(a_[i], b_[i]);
    }
    ...
};
```

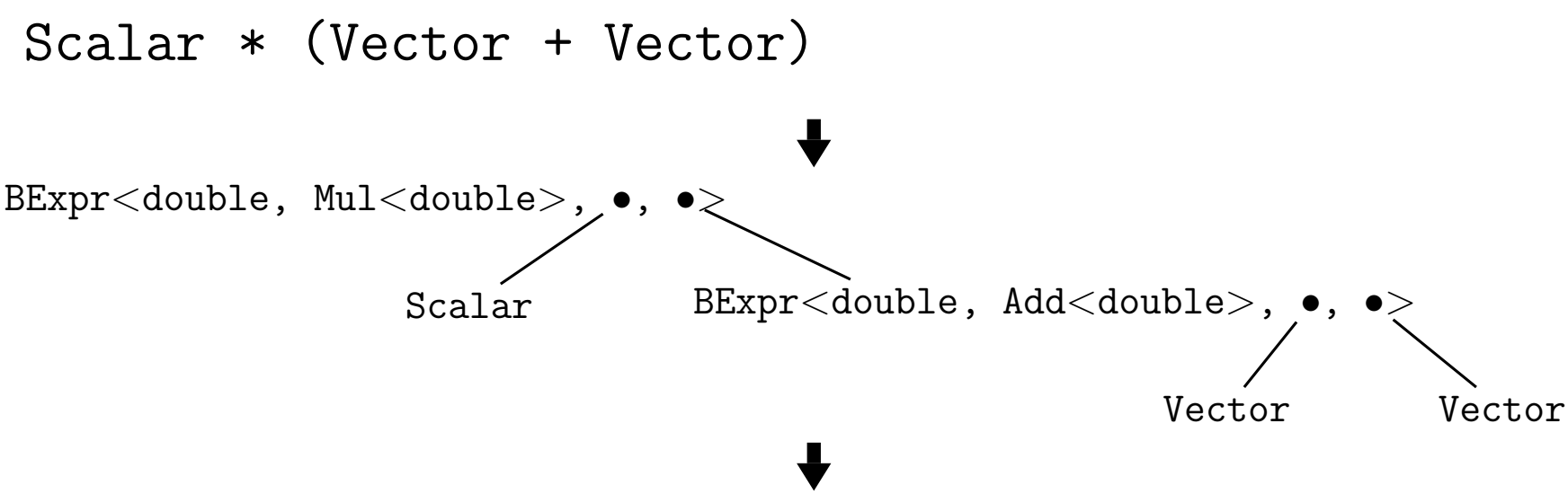
- Abstract representation of a binary operation
  - `OP` : policy class specifying operation
  - `A` and `B` : arguments
  - `T` : type of vector elements
- Overloaded subscript operator
  - Compute $\texttt{i}^{\text{th}}$ element
  - Policy class

Expression Templates


```
template<typename T> struct Add {
    static T eval(T a, T b) { return a + b; }
};
```

- Policy class
- Element-wise operation
  - Type `T`

Expression Templates




```
for (int i = 0; i < a.size(); ++i)
    r[i] = a[i] * (b[i] + c[i]);
```

Outline

Background

- Interval arithmetic
  - Expression templates
  - Template meta programming
- DSL
- Utilize GPU's
  - Specific Algorithms
  - Problem Solver

This Work

- Case study
  - Vector and matrix operations
- Two different approaches
  - Code generation
  - Utilize precompiled kernels

OpenCL Code Generation

- Problems
  - Data transfer
  - Mapping
  - Code generation

Data Transfer

- `cl::Buffer`
- Allocate and fill at creation
  - Only one data transfer
  - Permanent use of memory
- Allocate and fill within assignment operator
  - Data transfer for every expression
  - Uses only required memory

Mapping

- Unique Mapping between data and kernel parameters
- Singleton pattern
  - Generate unique id's
- Objects
  - `id_`
  - `getIdent()` $\longrightarrow$ `''v'' + id_`
  - `getCode()` $\longrightarrow$ `''v'' + id_ + ''[index]''`
- Scalar
  - Value

Code Generation

- Computation of required parameters
  - `paramSet : std::set`
  - Recursively filled
    - Vectors
    - Matrices
- Header

```
std::string code = "__kernel␣void␣CLETGenVectorKernel(␣";
for ( it = paramSet.begin() ; it != paramSet.end(); it++ ) {
        code += "__global␣float*␣" +
              (*it).getObject().getIdent() += ",␣";
}
code += "␣const␣unsigned␣int␣size␣)" +
        "\n{␣\n" +
        "␣const␣int␣index␣=␣get_global_id(0);␣\n";
```

Code Generation

- Body

```
code += "␣" + result.getCode() + "␣=␣" +
        expression.getCode() + ";\n};\n␣\n";
```

- getCode()
  - Result type → id
  - Expression tree
    - Replacement of subscript operator

```
std::string getCode() const {
        return std::string("(" + a_.getCode() +
                "␣+␣" + b_.getCode() + ")" );
 }
```

Utilize Precompiled Kernels

- Precompiled Kernels
- Minimize data transfer
  - DSL
  - Operator overloading
- User-friendly

Utilize Precompiled Kernels

- Expression templates are almost similar
- Inner nodes are annotated with id's
  - Mapping between operation and temporaries
- Evaluation
  - Two traversals

Utilize Precompiled Kernels

- First traversal
  - Memory allocation
  - `cl::Buffer`
  - Leafs as well as inner nodes
  - `std::map`
- Second traversal
  - Computation
  - Recursive

Outline

Experimental Results

- Code generation vs operator overloading
  - Without compilation:
    - Speedup of about 2
  - With compilation:
    - Only for hard problems
    - Similar expressions
    - Code mixin

Experimental Results

- Precompiled kernels vs operator overloading

Expression	Precompiled kernels	Operator overloading
`(M1 + M2) + (M3 + M4)`	26 ms	51 ms
`(M1 + M2) + (M3 + M1)`	21 ms	51 ms
`(M1 * M2) * (M3 * M4)`	183 ms	218 ms

Outline

Summary

- DSL
- User-friendly
- Code generation
  - Only for hard problems
  - Similar expressions
  - Code mixin
- Reduce data transfer

Questions ?

Marco Nehmeier

nehmeier@informatik.uni-wuerzburg.de