Seminar Presentation

Expression Templates
vs.
Smart Expression Templates

by Christian Fischer

Overview

- General overview of ET
- ET and Op-Overloading explained
- Performance of ET and Op-Overload in comparison
- Limits and Problems, which comes with ET's
- Introduction of SET to handle them
- Two examples of SET-Solutions of ET-Problems
- Comparison and Benchmark of SET-using libraries

What are ET's?

- Introduced by Veldhuizen in 1995
- Optimization for array-based operations
- Intents to avoid heavily and unnecessary use of temporary variables
- Able to reach similar performance as C-like-ops:

```
for(int i = 0; i < a.size(); i++) {
  d[i] = a[i] + b[i] + c[i];
}
```

The “Classic” Way

- Addition realised by operator-overloading
- Needs a temporary variable for addition
  - → De-/allocation and copy-operation needed
  - → Slow on large vectors
  - → Used memory blocked for other operations
- Improvement needed!

```
1  template<typename T>
2  class Vector {
3    public:
4      // Constructor, Destructor, other basics...
5
6      Vector& operator=(const Vector& rhs) {
7        if(&rhs == this) return *this;
8        std::copy(rhs._value, rhs._value + size(), this->_value);
9        return *this;
10     }
11     int size() const { return this->_size; }
12     T& operator[](int i) { return _value[i]; }
13     const T& operator[](int i) const { return _value[i]; }
14
15   private:
16       int _size; T* _value;
17 };
18
19 template<typename T>
20 const Vector<T> operator+(const Vector<T>& left,
21                           const Vector<T>& right) {
22   Vector<T> tmp(left.size());
23   for(int i = 0; i < right.size(); i++)
24     tmp[i] = left[i] + right[i];
25   return tmp;
26 }
```

Solution: Expression-templates (ET)

- Don't execute expression until assigned to target

  → prevents expensive memory-operations
- create a “placeholder” for the expression → syntaxtree
- temporaries just references
- only one for-loop, even with lot additions in row

  → classic: #adds = #loops

```
template<typename T>
class Vector {
public:
  // Constructor, Destructor, other basics...
  Vector& operator=(const Vector& rhs) {
    if(&rhs == this) return *this;
    std::copy(rhs._value, rhs._value + size(), this->_value);
    return *this;
  }
  int size() const { return this->_size; }
  T& operator[](int i) { return _value[i]; }
  const T& operator[](int i) const { return _value[i]; }
private:
  int _size; T* _value;
};
```

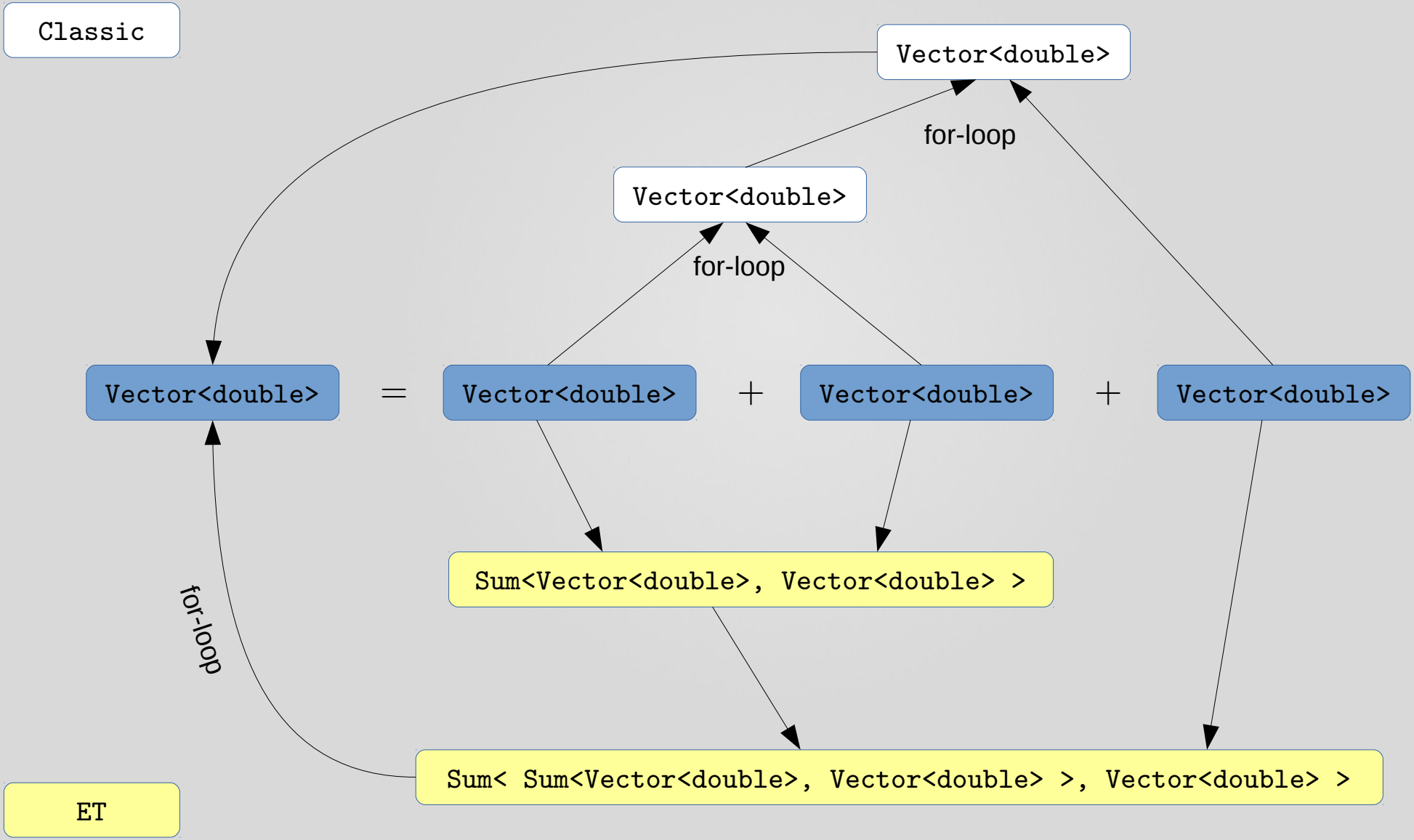
```
template<typename T>
const Vector<T> operator+(const Vector<T>& left,
                          const Vector<T>& right) {
  Vector<T> tmp(left.size());
  for(int i = 0; i < right.size(); i++)
    tmp[i] = left[i] + right[i];
  return tmp;
}
```

```
template<typename A, typename B>
Sum<A, B> operator+(const A& a, const B& b) {
  return Sum<A, B>(a, b);
}
```

```
template<typename A, typename B>
class Sum {
  public:
    explicit Sum(const A& l, const B& r) :
                 _left(l), _right(r) {}
    int size() const   { return this->_left.size(); }
    double operator[](int i) const
      { return this->_left[i] + this->_right[i]; }
  private:
    const A& _left; const B& _right;
};
```

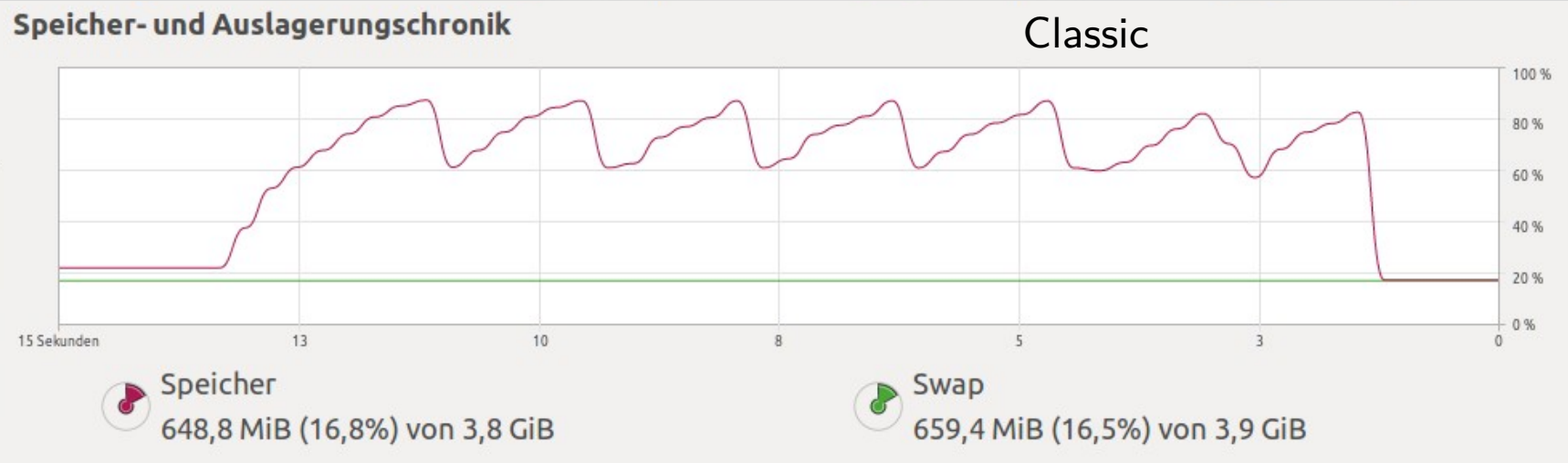
```
class Vector {
    ...
    // ET assignment-operator
    template<typename A>
    Vector& operator=(const A& expr) {
      for(long i = 0; i < expr.size(); i++)
        this->_value[i] = expr[i];
      return *this;
    }
    ...
};
```

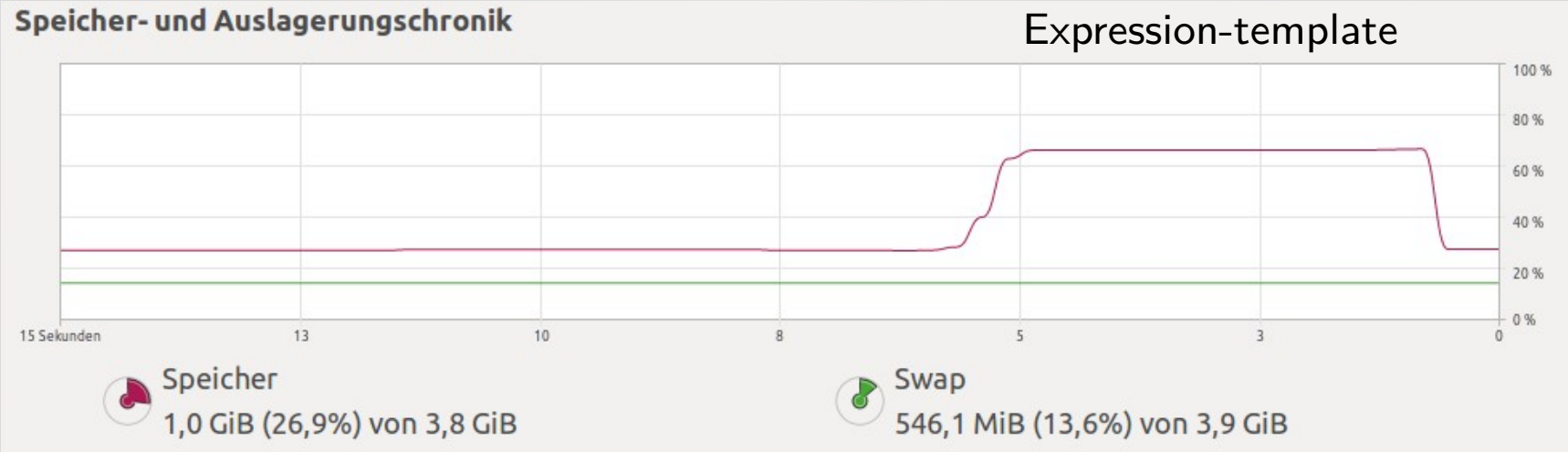
Example: d = a + b + c



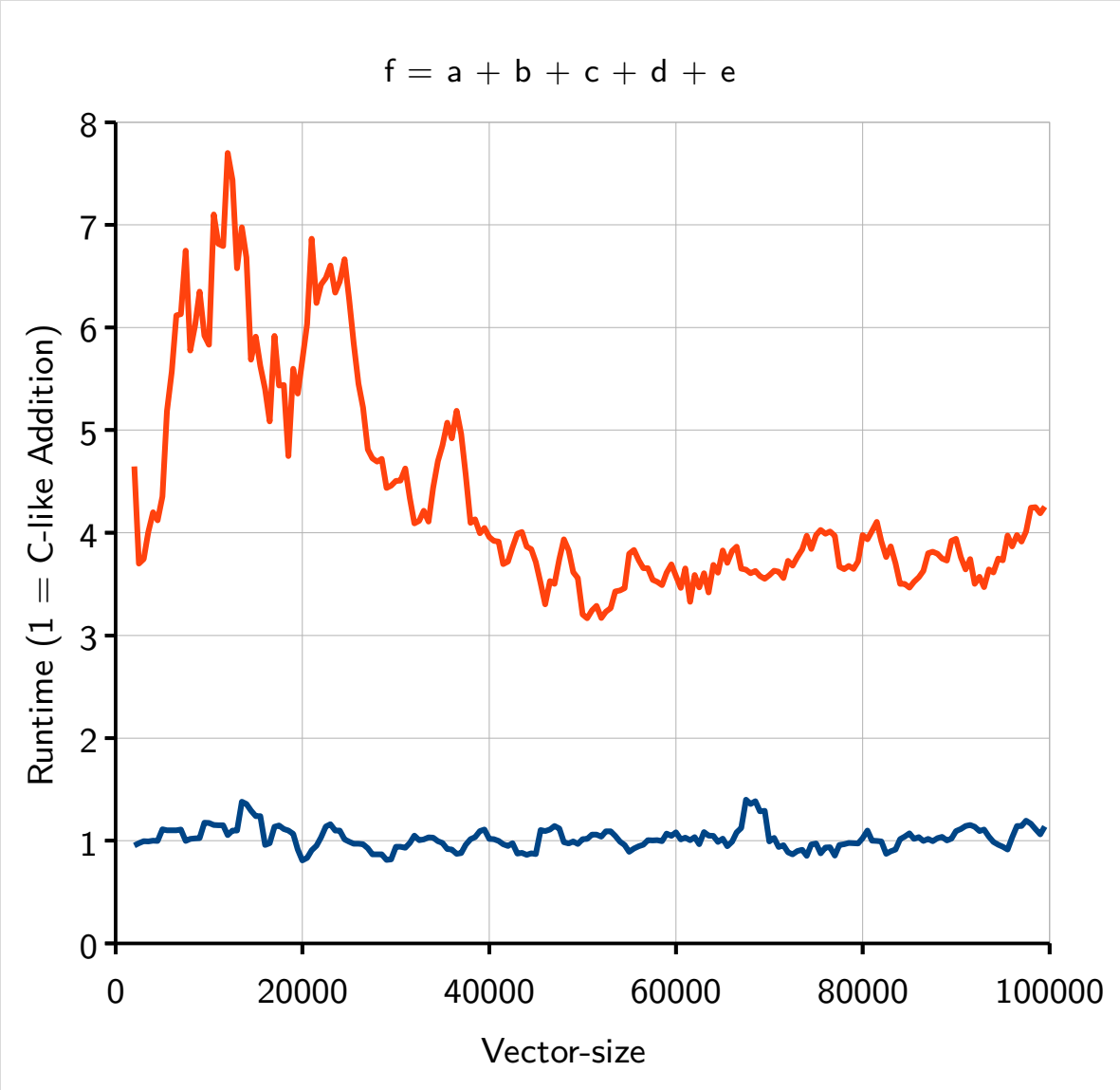
“Classic” vs. ET - Memory

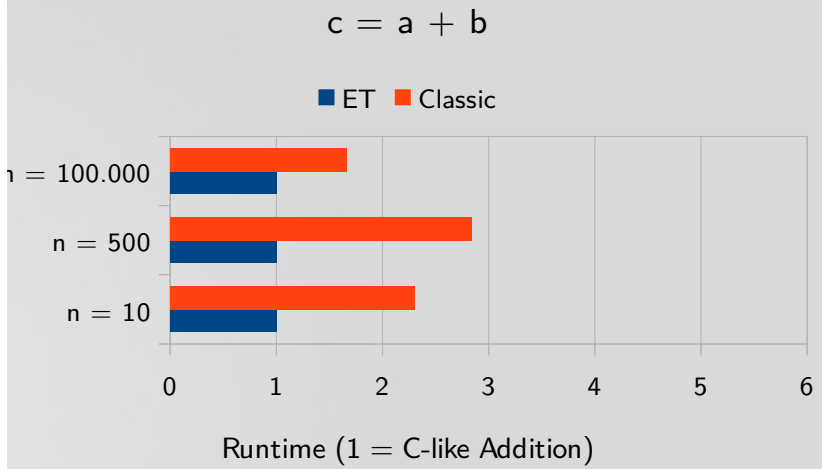
Adding 5 vectors with 33-million entries each (7-times in total)

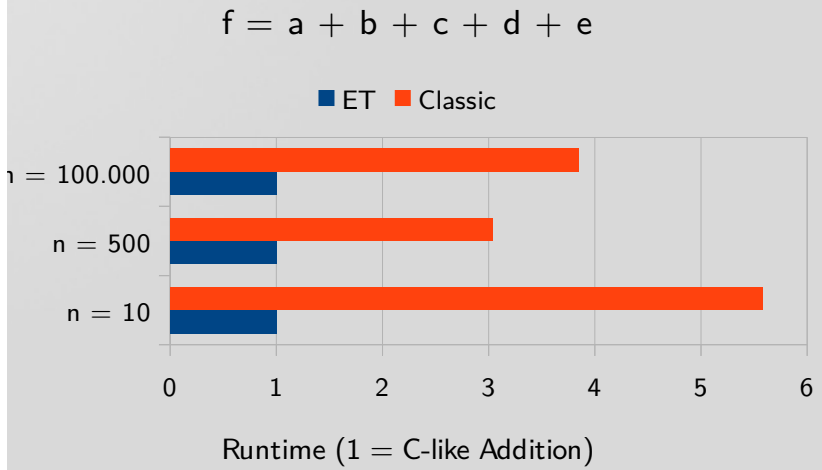




“Classic” vs. ET - Runtime





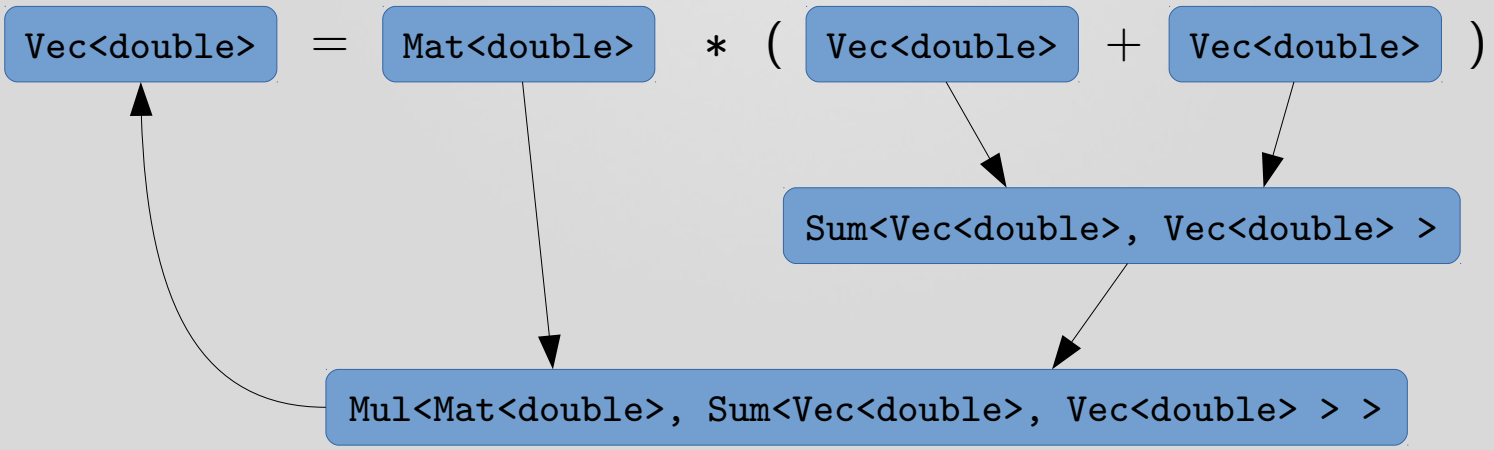


Limits of ET's - Examples

- ET's has big advantages for simple expressions
- More complex expr. may lead to disadvantage

- Expr. evaluated at assignment
- Addition repeated for every row of the matrix
- Avoidable by using temp. var

Necessary temporaries



Limits of ET's - Examples

- ET's has big advantages for simple expressions
- More complex expr. may lead to disadvantage

Evaluation strategies

- Standard-evaluation leads to temp. variable
- Avoidable by using variable, which is used anyways

$$A = B + C * D$$

Left-to-Right-Eval:

1 | Temp = C * D

2 | A = B + Temp

Better strategy:

1 | A = B

2 | A += C * D

Limits of ET's - Examples

- ET's has big advantages for simple expressions
- More complex expr. may lead to disadvantage

- Standard-evaluation leads to Mat.-Mat.-Multiplication
- Avoidable by restructuring the expression

Expression-restructuring

Mat * Mat * vec

Left-to-Right-Eval:

(Mat * Mat) * vec

$\rightarrow$ Mat * vec

$\rightarrow O(n^3) + O(n^2) = O(n^3)$

Restructured:

Mat * (Mat * vec)

$\rightarrow$ Mat * vec

$\rightarrow O(n^2) + O(n^2) = O(n^2)$

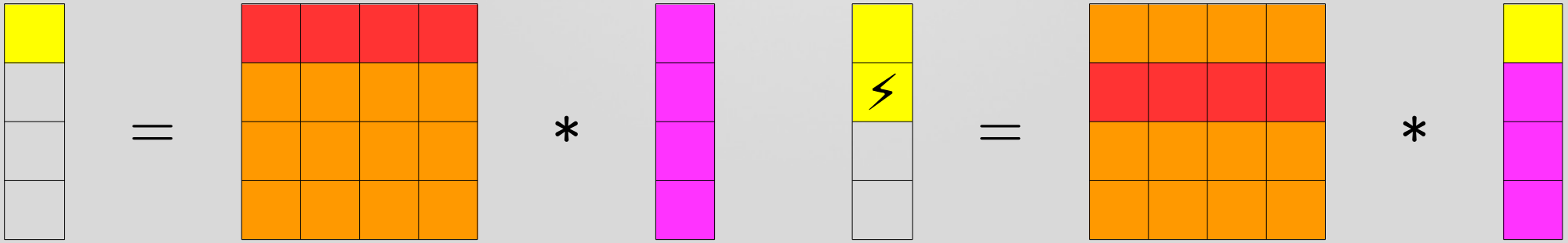
Limits of ET's - Examples

- ET's has big advantages for simple expressions
- More complex expr. may lead to disadvantage

Aliasing

- ET: evaluation at assignment $\rightarrow$ no temp
- Manipulates values, which are still required

$$x = A * x$$



Solution: Smart-Expression Templates (SET)

Idea behind SET:

- distinguish between expr. + plain obj. in operations
- ResultType:
  - Type after evaluation
- CompositeType:
  - Type if used in another expression
- SET take advantage of this information e.g. to create temporary variables

```
1  template <typename MT, typename VT>
2  class MatVecMultExpr : public Vector<MatVecMultExpr<MT,VT>>,
3                         private Expression {
4    public:
5      // Result type for expression template evaluations.
6      typedef typename MathTrait<MRT, VRT>::MultType ResultType;
7      // Data type for composite expression templates
8      typedef const MatVecMultExpr& CompositeType;
9      // Resulting element type
10     typedef typename ResultType::ElementType ElementType;
11     // Member data type of the left -hand side dense matrix expression
12     typedef typename MT::CompositeType Lhs;
13     // Member data type of the right -hand side dense vector expression
14     typedef typename SelectType<IsExpression<VT>::value,
15                                 const VRT, const VT&>::Type Rhs;
16     inline const ElementType operator[]( size_t index ) const;
17     inline size_t size() const;
18     // ...
19   private:
20     // Result type of the left -hand side dense matrix expression
21     typedef typename MT::ResultType MRT;
22     // Result type of the right -hand side dense vector expression
23     typedef typename VT::ResultType VRT;
24
25     // Element type of the left -hand side dense matrix expression
26     typedef typename MRT::ElementType MET;
27
28     // Element type of the right -hand side dense vector expression
29     typedef typename VRT::ElementType VET;
30     // ...
31     Lhs mat_; // Left -hand side dense matrix of the mult.-expr.
32     Rhs vec_; // Right -hand side dense vector of the mult.-expr.
33     // ...
34 };
```

Example: Detecting Aliasing

- Potentially dangerous to allow incorrect calcs
- implement isAliased(...) in every vec, mat, expr.
- assignment op. checks if aliasing is present
- swaps the rhs with a temporary variable

```
1  template <typename MT, typename VT>
2  class MatVecMultExpr : public Vector<MatVecMultExpr<MT,VT>>,
3                         private Expression {
4    public:
5      // ...
6      template <typename T>
7      inline bool isAliased(const T* alias) const {
8        return vec_.isAliased(alias);
9      }
10      // ...
11    private:
12      // ...
13      Lhs mat_;
14      Rhs vec_;
15      // ...
16 };
```

```
1  // in Vector-class
2  template <typename T>
3  template <typename O>
4  inline bool Vector<T>::isAliased(const O* alias) const {
5    return static_cast<const void*>(this) ==
6           static_cast<const void*>(alias);
7  }
8  // ...
9  // in the overloaded assignment-operator of Vector-class
10  if(rhs.isAliased(this)) {
11    Vector tmp(rhs);
12    swap(tmp);
13  }
```

Tools using SET's

Overview, which library supports handling of the shown problems wit ET's

	Eigen3	Boost	uBLAS	Blaze	Blitz++
Necessary temps	+++	---	---	+++	---
Evaluation strategies	+++	---	---	+++	---
Expression restructuring	+++	---	---	+++	---
Aliasing	○	○	○	+++	---

+++: Supported, ---: Not supported, ○: just with right handling

Benchmarks between SET-Tools

- A * (a + b)
- Graph with results
- A * (a + b + c)
- (A * B) * (a + b)
- (A * B) + C

Conclusion

- ET are able to improve operations between mathematical objects by avoiding temporaries in contrast to classic operator-overloading
- Avoiding these temps leads to new problems, which can be solved by SET
- Not every library which uses ET/SET handles the problems, which comes with them

Question-and-Answer

Any questions, comments, etc.?

Thanks for your attention!

Literature

http://arxiv.org/pdf/1104.1729.pdf

https://www10.cs.fau.de/publications/reports/TechRep09-17.pdf

https://blogs.fau.de/hager/files/2012/05/ET-SISC-Iglberger2012.pdf