Efficient Programming

Michael Hanke

School of Engineering Sciences

Program construction in C++ for Scientific Computing

Outline

Intro

- In Scientific Computing, efficiency with respect to memory and execution time is an issue.
- In this lecture, we will give a very short introduction to programming principles enhancing the performance of a code.

Instruction Execution: Pipelining

Every instruction is carried out in different stages. It could be something like:

- Instruction fetch (IF)
- Instruction decode (ID)
- Execute (EX)
- Memory access (MEM)
- Register write back (WB)

Schematically:

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

A real processor has around 15 – 20 stages!

Pipelining Stalling

Problem

The pipeline may *stall*.

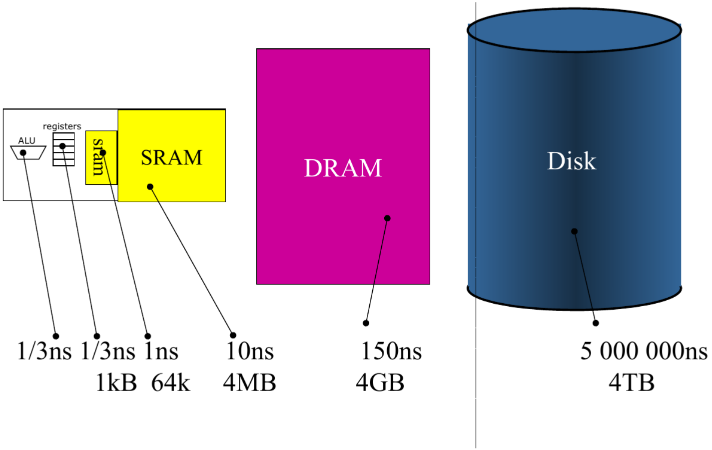
Reasons:

- Data dependencies: An instruction needs data which a previous instruction did not yet deliver.
- Interrupt of the sequential execution by branches.
- The data is not available.

Pipelining: Hardware Optimizations

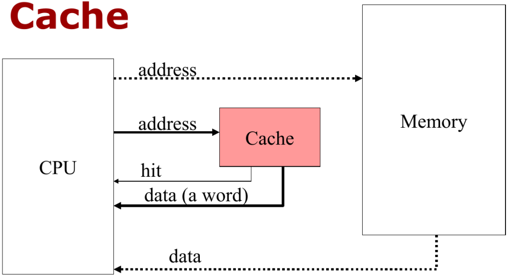
- Out-of-order execution (A good optimizing compiler does it, too, during code generation)
- Speculative execution
- Prefetching (in connection with caches, even a good compiler does it)
- Branch prediction
- Superscalar architecture (more than one execution pipeline)
    - may lead to another problem if the number of identical execution units is less than the number of pipelines)

Memory Hierarchies



Memory Access (Schematic)

Cache



- Hit: Use data provided from the cache
- No-Hit: Use data from memory and also store it in the cache
- Data are moved to memory in cache lines (architecture dependent, typically 64 bytes).
- n-way associativity

Conclusions

- *Space locality*: Access data located as close as possible to each other
  - Avoid indirect addressing
- *Time locality*: Identical data shall be accessed as short as possible consecutively
  - Reuse data if possible
- Avoid branches in loops.
- If there is a branch in a loop, the most often used alternative should follow subsequently

Consequences of Pipelining

Function for computing x_i^k, where $k = 2, 3$:

```
void f1(int n, double x[], int k) {
  for (int i = 0; i < n; i++)
    if (k == 2) x[i] = pow(x[i],2);
    else x[i] = pow(x[i],3);
}
void f2(int n, double x[], int k) {
  if (k == 2)
    for (int i = 0; i < n; i++)
      x[i] = pow(x[i],2);
  else for (int i = 0; i < n; i++)
      x[i] = pow(x[i],3);
}
```

`f1` and `f2` perform the same calculations.
Execution time of `f2` is usually faster than that of `f1` (heavily compiler dependent!)

Array Indexing

C++ Traditional 2D arrays are stored in row-wise order, although the language standard does not guarantee this.

```
x = new double[10][5]
```

allocates 10 arrays of 5 elements each.

Fortran 2D arrays are stored in column-wise order (guaranteed by the language standard).

Storage and Efficiency

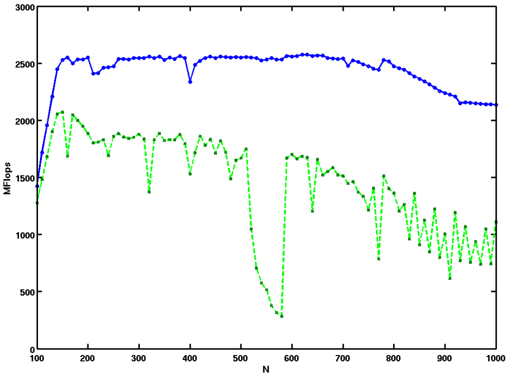
Storage order is irrelevant for efficiency. Implementation of numerical methods must be optimized depending on order!

Example: Matrix-Vector Multiplication

```
double A[N][N], x[N], y[N];
// initialize A, x; set y to zero
// Order: Traverse A continuously
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    y[i] += A[i][j]*x[j];
// Order: ‘‘Jump’’ through A
for (int j = 0; j < N; j++)
  for (int i = 0; i < N; i++)
    y[i] += A[i][j]*x[j];
```

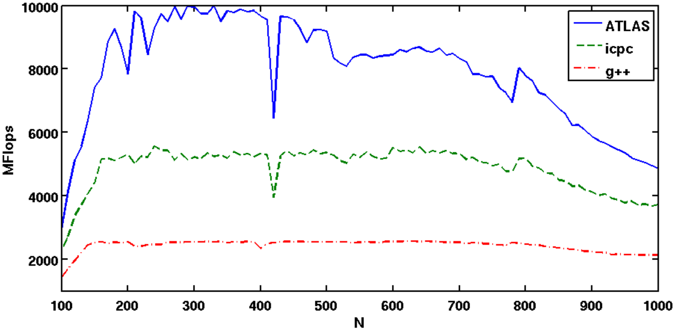
Both versions are mathematically equivalent.

Example (cont)



- Compiler: g++ 4.8.1, -O3
- Machine: My laptop (Intel 2720QM@2.20, 6 MB level 3 cache)

Example (cont)



- Compiler: g++ 4.8.1, ATLAS 3.10.1, icpc 14.0
- Machine: My laptop (Intel 2720QM@2.20, 6 MB level 3 cache)
- What is going on??

Example: Matrix-Matrix Multiplication

- *Problem*: For $C = A \cdot B$, we must evaluate

$$c_{ij} = \sum_{k=0}^{N} a_{ik} b_{kj}$$

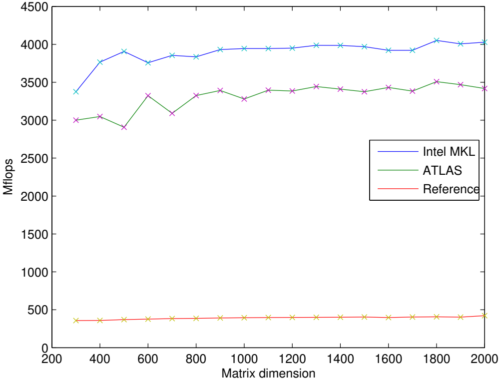
  For forming c_{ij}, the matrices must be traversed in different order (A row-oriented, B column-oriented)

    - How to organise an efficient memory access pattern?

- *Solution*: Implement a block-wise algorithm which uses cache efficiently!

    - Nontrivial
    - Hardware- and compiler-dependent

Example (cont)



- Compiler: ifort 8.1 (?), -O2
- Machine: Desktop, AMD Athlon XP

Use Libraries

Moral: Small mistakes can ruine performance.
Use optimized numerical libraries whenever possible!

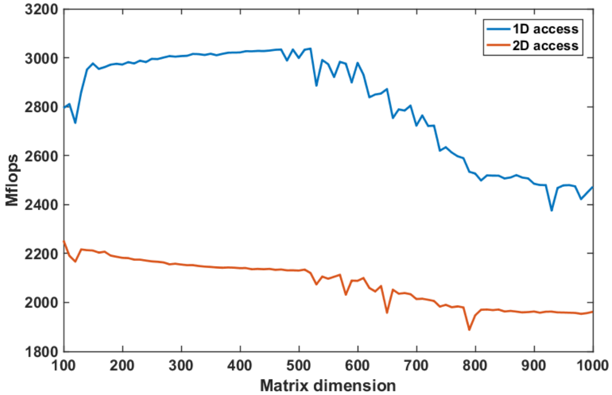
+ good performance with little effort
+ less programming, i.e. debugging and testing
+ one can focus on essentials, e.g. PDEs instead of linear algebra
- not all libraries are good, choose carefully
- must complain to certain storage formats

Recommandation: Replace `X[m][n]` by `x[m*n]` and map `X[i][j] = x[i+j*m]` (column major)

Example: Matrix-Vector Multiplication

```
double A[N][N], a[N*N], x[N], y[N]
// Initialize A, a, x, set y to zero
// 2D access
for (i=0 ; i<n ; i++)
  for (j=0 ; j<n ; j++)
    y[i] += A[i][j]*x[j];
// 1D access (columnwise)
idx=0;
for (j=0 ; j<n ; j++)
  for (i=0 ; i<n ; i++) {
    y[i] += a[idx]*x[j];
    idx++;
}
```

Example (cont)



- Compiler: g++ 4.8.3, -O6
- Machine. My laptop (Intel i7-5600U @ 2.60GHz, 4 MB cache)

Standard Libraries

- De-Facto standard in Scientific Computing: (C)BLAS, LAPACK for basic linear algebra routines (full and banded matrices)
- Fast Fourier transforms: FFTW
- Sparse linear algebra: PETSc (your milage may vary), Trilinos
- Sparse LU etc: MUMPS, SuperLU, SuiteSparse
- Many, many, many more

Use vendor-supplied libraries whenever possible!
Examples: Intel MKL, AMD ACML, SPARC sunperf
Public domain replacements: ATLAS, OpenBLAS

A Simple Matrix Class

Our aim is to construct a simple matrix class which behaves like matrices in matlab:

- All reasonable operations should be allowed if they are mathematically legal.
- Matrices with one dimension equal to 1 are considered to be vectors.
- Matrices of dimensions (1,1) are scalars.

We intend to show performance issues. Therefore:

- We will not use generic programming.
- We will not use C++'s standard libraries (in particular containers).

The Basics

```
class Matrix {
  int m, n;  // should be size_t
  double *A;
public:
  Matrix(int m_ = 0, int n_ = 0) : m(m_), n(n_), A(nullptr) {
    if (m*n > 0) {
      A = new double[m*n];
      std::fill(A,A+m*n,0.0);
      // cblas_dcopy may be faster
    }
  }

  ~Matrix() { if (A != nullptr) delete [] A; }
  double& operator()(int i, int j) { return A[i+j*m]; }
  const double operator()(int i, int j) const { return A[i+j*m]; }
};
```

Notes:

- We used column-major for storing the matrix.
- Copy and move constructors will be needed, too.

Additional Constructors

```
Matrix(const Matrix& B) : m(B.m), n(B.n), A(nullptr) {
  if (n*m > 0) {
    A = new double[n*m];
    std::copy(B.A,B.A+m*n,A);
  }
}

Matrix(Matrix&& B) noexcept : m(B.m), n(B.n), A(B.A) {
  B.m = 0; B.n = 0; B.A = nullptr;
}
```

Overloaded Operators I

```
Matrix& operator=(const Matrix& B) {
  if (this != &B) {
    if (m*n != B.m*B.n) {
      if (A != nullptr) delete [] A;
      if (B.A != nullptr) A = new double[B.m*B.n];
    }
    m = B.m; n = B.n;
    std::copy(B.A,B.A+m*n,A); // ?
  }
  return *this;
}
Matrix& operator=(Matrix&& B) {
  m = B.m; n = B.n;
  if (A != nullptr) delete [] A;
  A = B.A;
  B.m = B.n = 0;
  B.A = nullptr;
}
```

Overloaded Operators II

```
const Matrix operator*(const Matrix& B) const {
  if (n != B.m) error();
  Matrix tmp(m,B.n);
  if (tmp.A == nullptr) return tmp;
  for (int i = 0; i < m; i++)
    for (int j = 0; j < B.n; j++) {
      tmp.A[i+j*m] = 0.0;
      for (int k = 0; k < n; k++)
        tmp.A[i+j*m] += A[i+k*m]*B.A[k+j*m];
      }
  return tmp;
}
```

This implementation is extremely slow as we have seen before!

Optimizing Overloaded Operators

```
#include <cblas.h>
const Matrix operator*(const Matrix& B) const {
  if (n != B.m) error();
  Matrix tmp(m,B.n);
  if (tmp.A == nullptr) return tmp;
  cblas_dgemm(CblasColMajor,CblasNoTrans,
      CblasNoTrans,m,n,B.n,
      1.0,A,m,B.A,n,0.0,tmp.A,m);
  return tmp;
}
```

Note: The `dgemm` routine evaluates a much more complex expression: $C := \alpha AB + \beta C$.

More Complex Expressions

For the following explanations assume that we have defined an addition operation:

```
const Matrix operator+(const Matrix& B) const {
  // Insert tests for correctness and memory management
  Matrix tmp(m,n);
  for (int i = 0; i < m*n; i++) tmp.A[i] = A[i]+B.A[i];
  return tmp;
}
```

Note: The corresponding BLAS routine would be `cblas_daxpy`.

Problem: A temporary is created which is then copy-assigned to the result.

Optimizations: 1

- We have previously seen that a lot of copying can be avoided by using the move-assignment operator:

```
Matrix& operator=(Matrix&& A);
```

- In the assignment `C = A+B;`, this operator *will not be invoked* because `A` is no longer `const`! Hence, the signature of the addition operator must be changed:

```
~~const~~ Matrix operator+(const Matrix& A) const;
```

- A temporary object will be created anyway, but the assignment is “light-weight”.

Optimizations: 2

Define a member function:

```
void add(const Matrix& B, Matrix& C) const;
```

- Here, the creation of temporaries is avoided completely.
- Copy management is handed over to the user.
- However, the notation becomes rather clumsy: Instead of the elegant notation

```
C = A+B;
```

- we have

```
A.add(B,C);
```

- How can we implement `M = A+B+C;` etc??

Even More Complex Expressions

- Consider M = A+B+C;
- With the definitions above, this will be compiled to:

```
t1 = A+B; // Matrix A.operator+(const Matrix& B)
t2 = t1+C; // Matrix t1.operator+(const Matrix& C)
M = t2;  // Matrix& operator=(Matrix&& t2)
```

- *In order to avoid the deep copy we would need an operator which takes temporaries as the first argument.*

Operators With Temporary Expressions

- If the first argument is an rvalue reference, the operator cannot be a member of the class. So we must declare it a friend:

```
friend Matrix operator+(Matrix&& A, const Matrix& B);
```

- So a definition might be:

```
Matrix operator+(Matrix&& A, const Matrix& B) {
  A += B;  // Assumes a standard definition of +=
  return std::move(A); // Invokes the move-constructor
}
```

- The call to the move-constructor could have been replaced by an explicit type cast:

```
    return static_cast<Matrix&&> A;
```

Temporary Expressions (cont)

Our statement M = A+B+C becomes now:

```
t1 = A+B; // Matrix A.operator+(const Matrix& B)
t2 = t1+C; // Matrix operator+(Matrix&& t1, const Matrix& C)
M = t2;  // Matrix& operator=(Matrix&& t2)
```

Temporary Expressions (cont)

A very good compiler would inline the corresponding functions and generate a code like the following:

```
for (int i = 0; i < m*n; i++) t1[i] = A[i]+B[i];
for (int i = 0; i < m*n; i++) M[i] = t1[i]+C[i];
```

However, the optimal implementation would be something like this:

```
for (int i = 0; i < m*n; i++)
   M[i] = A[i]+B[i]+C[i];
```

This is called *loop fusion*.

Expression Templates

- *Basic idea*: Create types which encode complex expressions. In our example, it may be something like

  ```
  Sum< Sum<Matrix, Matrix>, Matrix>
  ```

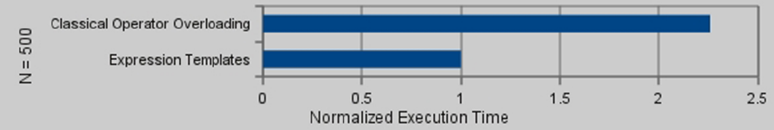
- Applying the index operator to an object of that type reduces to an expression including all operations (in our example: `A[i]+B[i]+C[i]`).
- The assignment operator becomes a type cast. It traverses through all indices.
- Note: *Templates are instantiated during compile time!*
- Metaprogramming

Expression Templates (cont)

- This technique may lead to an efficiency comparable to hand-coded code for vector operations.
- The first implementation is the `blitz++` library by Todd Veldhuizen.
- Expression templates have very high demands on the compiler!
- Cf David Vandevoorde and Nicolai M. Josuttis: *C++ Templates, The Complete Guide*, Pearson 2003, Chapter 18

A Simple Comparison

Evaluation of the expression M = A+B+C with m = 500, n = 1:



Machine: Intel i7 940
Compiler: g++ 4.4.1

Source: PhD Thesis Klaus Iglberger, FAU Erlangen-Nürnberg 2010

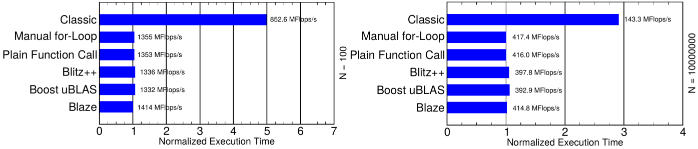
ET: Libraries

- **blitz++**: Todd Veldhuizen (The first implementation of this idea), http://sourceforge.net/projects/blitz/
- **Boost uBLAS**: Joerg Walter and Mathias Koch, http://www.boost.org/ (focus *not* on efficiency)
- **Armadillo**: Conrad Sanderson et al, http://arma.sourceforge.net/
- **MTL4**: Peter Gottschling et al, http://www.simunova.com/de/home
- **Eigen3**: Benoît Jacob, Gaël Guennebaud et al, http://eigen.tuxfamily.org/index.php?title=Main_Page
- **blaze**: Klaus Igelberger (smart ET) https://bitbucket.org/blaze-lib/blaze

and many, many more.
The functionality is usually much larger than simple linear algebra operations.

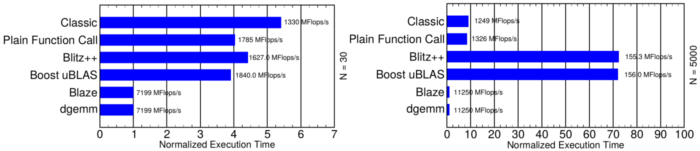
Example: Vector Addition

All the following examples are taken from: K. Igelberger, G. Hager, J. Treibig, U. Rüde: SIAM J Scientific Comp 34(2012), C42-C69. Pictures taken from preprint.



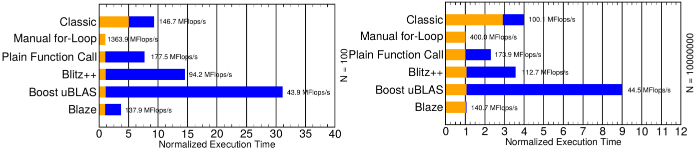
Machine: Intel Westmere@2.93GHz, 12MB cache
Compiler: g++ 4.4.2

Example: Matrix Multiplication



dgemm: Intel MKL

The Importance of Inlining: Vector Addition



Yellow: Complete inlining
Blue: No inlining

Stroustrup's Proposal: Composite Objects

- The previous approach is well-suited for expressions like `y = A*x`.
- However, the expression `x = A*x` cannot be handled this way because a temporary is needed.
- *It cannot be decided at compile time if x and y are aliased!*
- A different approach consists in doing the decision at execution time: An expression is only evaluated if the assignment takes place (lazy evaluation).
- Idea: If an expression like `y = A*x+y` (dgemv) is to be evaluated, the * and + operators create only a structure with information about the operations to be performed. It is `operator=()` which performs the real operation, eg by calling `cblas_dgemv`.
- Cf Suely Oliveira and David Steward: *Writing Scientific Software*, Section 8.6
- Not as flexible as expression templates.

Summary

- Libraries, libraries, libraries
- The design and implementation of an efficient class requires a deep understanding of hard- and software environment.
- Even if designed with efficiency in mind, careless use of C++ may lead to extremely inefficient executables.
- "90% of the computation time are spent in 10% of the code." Identify and optimize hotspots!
- Finally a reference: Agner Fog, Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms. http://www.agner.org/optimize/optimizing_cpp.pdf