A short course on object-oriented numerics

Sylwester Arabas
Faculty of Physics, University of Warsaw

local organisers:
Juan A. Añel, **Orlando García**

EPhysLab, Universidade de Vigo, Ourense

day 2 (June 4^{th} 2014)

plan for today: getting acquainted with Blitz++

Veldhuizen & Jernigan 1997

Will C++ be faster than Fortran?

Todd L. Veldhuizen and M. Ed Jernigan

Department of Systems Design Engineering, University of Waterloo,
Waterloo Ontario N2L 3G1 Canada
tveldhui@monet.uwaterloo.ca

Abstract. After years of being dismissed as too slow for scientific computing, C++ has caught up with Fortran and appears ready to give it stiff competition. We survey the reasons for the historically poor performance of C++ (pairwise expression evaluation, the abstraction penalty, aliasing ambiguities) and explain how these problems have been resolved. C++ can be faster than Fortran for some applications, due to template techniques (such as expression templates and template metaprograms) which permit optimizations beyond the ability of current Fortran compilers.

documentation (0.9)

`http://blitz.sf.net/resources/blitz-0.9.pdf`

packages (0.10)

Debian/Ubuntu: libblitz-dev
Fedora: blitz-devel

Example 3: Blitz++ API basics

```cpp
#include <blitz/array.h>

int main()
{
  blitz::Array<int, 2> a(4, 4);
  // dimensionality-^    ^^^^-size

  // basic loop-free arithmetics
  a = 0; // before - undefined values!
  a += 2;
  std::cerr << a << std::endl;

  { // array-index arithmetics
    using namespace blitz::tensor;
    a = (i == j); // 1 on diagonal, else 0
  }
  std::cerr << a << std::endl;

  { // element addresing
    a(0, 0) = 2;

    blitz::Range i(1, 2), j(1, 2);
    a(i, j) = 8;

    blitz::RectDomain<2> ij({i, j});
    a(ij) -= 4;
  }
  std::cerr << a << std::endl;

  { // reductions
    using namespace blitz::tensor;
    blitz::Array<int, 1> s(sum(a, j));
    std::cerr << s << std::endl;
  }
}
```

```
$ clang++ -std=c++11 -lblitz -Ofast\
    -march=native ex2.cpp
$ ./a.out
(0,3) x (0,3)
[ 2 2 2 2
  2 2 2 2
  2 2 2 2
  2 2 2 2 ]

(0,3) x (0,3)
[ 1 0 0 0
  0 1 0 0
  0 0 1 0
  0 0 0 1 ]

(0,3) x (0,3)
[ 2 0 0 0
  0 4 4 0
  0 4 4 0
  0 0 0 1 ]

(0,3)
[ 2 8 8 1 ]
```

Example 3: Blitz++ API basics (debug mode)

what if we change "i(1,2)" into i(1,22)

- compiles fine
- may run fine!
- if lucky, one may get a run-time error (segfault or alike)

Blitz++ debug mode: recompile with -DBZ_DEBUG
(to compile with clang++, see `http://sf.net/p/blitz/bugs/55/`)

```
$ g++ -std=c++11 -lblitz -DBZ_DEBUG ex2.cpp
$ ./a.out

...

Bad array slice: Range(1, 22, 1).  Array is Range(0, 3)
a.out: /usr/include/blitz/array/slicing.cc:340:
  void blitz::Array<P_numtype, N_rank>::slice(int, blitz::Range)
  [with P_numtype = int; int N_rank = 2]: Assertion ‘0’ failed.
Aborted
```

Example 4: Blitz++ API / element-wise fun. evaluation

```cpp
#include <blitz/array.h>
#include "fun.hpp"

int main()
{
  // custom array ranges
  blitz::Range
    x(-3, 3), y(-3, 3);
  blitz::Array<double, 2>
    a(x, y), b(x, y);

  // output settings
  std::cout.precision(2);
  std::cout.setf(std::ios::fixed);

  // built-int fun. evaluation
  {
    using namespace blitz::tensor;
    a = exp(-(pow(i,2) + pow(j,2)));
  }
  std::cout << a << std::endl;

  // user-defined fun. evaluation
  b = fun({.a = -1, .b = 1})(a);
  std::cout << b << std::endl;
}
```

```
$ clang++ -std=c++11 -lblitz ex3.cpp
$ ./a.out
(-3,3) x (-3,3)
[ 0.00 0.00 0.00 0.00 0.00 0.00 0.00
  0.00 0.00 0.01 0.02 0.01 0.00 0.00
  0.00 0.01 0.14 0.37 0.14 0.01 0.00
  0.00 0.02 0.37 1.00 0.37 0.02 0.00
  0.00 0.01 0.14 0.37 0.14 0.01 0.00
  0.00 0.00 0.01 0.02 0.01 0.00 0.00
  0.00 0.00 0.00 0.00 0.00 0.00 0.00 ]

(-3,3) x (-3,3)
[ 1.00 1.00 1.00 1.00 1.00 1.00 1.00
  1.00 1.00 0.99 0.98 0.99 1.00 1.00
  1.00 0.99 0.86 0.63 0.86 0.99 1.00
  1.00 0.98 0.63 0.00 0.63 0.98 1.00
  1.00 0.99 0.86 0.63 0.86 0.99 1.00
  1.00 1.00 0.99 0.98 0.99 1.00 1.00
  1.00 1.00 1.00 1.00 1.00 1.00 1.00 ]
```

Example 4: Blitz++ API / element-wise fun. evaluation

```cpp
#include <blitz/array.h>
#include "fun.hpp"
```

fun.hpp

```cpp
#pragma once

#include <blitz/array.h>

struct fun
{
  const double a, b;

  double operator()(double x) const
  {
    return a * x + b;
  }

  // make it work with Blitz++ types
  BZ_DECLARE_FUNCTOR(fun)
};
```

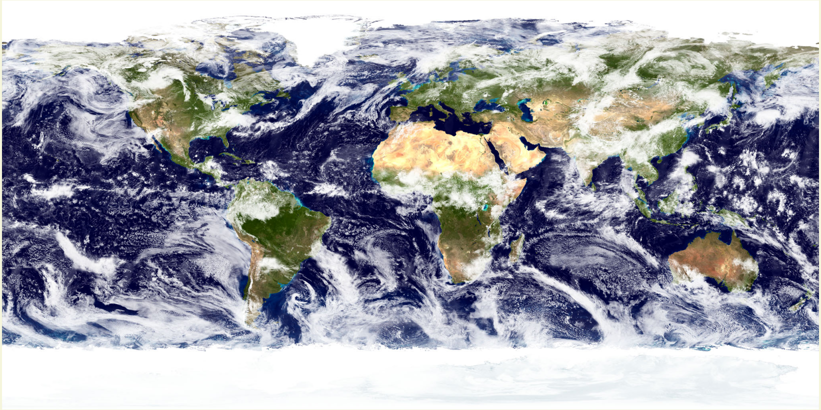
```cpp
  // user-defined fun. evaluation
  b = fun({.a = -1, .b = 1})(a);
  std::cout << b << std::endl;
}
```

```
$ clang++ -std=c++11 -lblitz ex3.cpp
$ ./a.out
(-3,3) x (-3,3)
[ 0.00 0.00 0.00 0.00 0.00 0.00 0.00
  0.00 0.00 0.01 0.02 0.01 0.00 0.00
  0.00 0.01 0.14 0.37 0.14 0.01 0.00
  0.00 0.02 0.37 1.00 0.37 0.02 0.00
  0.00 0.01 0.14 0.37 0.14 0.01 0.00
  0.00 0.00 0.01 0.02 0.01 0.00 0.00
  0.00 0.00 0.00 0.00 0.00 0.00 0.00 ]

(-3,3) x (-3,3)
[ 1.00 1.00 1.00 1.00 1.00 1.00 1.00
  1.00 1.00 0.99 0.98 0.99 1.00 1.00
  1.00 0.99 0.86 0.63 0.86 0.99 1.00
  1.00 0.98 0.63 0.00 0.63 0.98 1.00
  1.00 0.99 0.86 0.63 0.86 0.99 1.00
  1.00 1.00 0.99 0.98 0.99 1.00 1.00
  1.00 1.00 1.00 1.00 1.00 1.00 1.00 ]
```

Example 5: a draft of an advection-equation solver

$$\partial_t \psi + \nabla \cdot (\vec{u}\psi) = 0$$



http://earthobservatory.nasa.gov/Features/BlueMarble/BlueMarble_2002.php

Example 5: a draft of an advection-equation solver

$$\partial_t \psi + \nabla \cdot (\vec{u}\psi) = 0$$

donor-cell algorithm for constant $\vec{u} > 0$

$$C = \left\{ u_x \frac{\Delta t}{\Delta x},\ \ u_y \frac{\Delta t}{\Delta y} \right\}$$

$$\begin{aligned}
\psi^{n+1}_{i,j} = \psi^{n}_{i,j} \ &-\ \left(C_x \psi^{n}_{i,j} - C_x \psi^{n}_{i-1,j} \right) \\
&-\ \left(C_y \psi^{n}_{i,j} - C_y \psi^{n}_{i,j-1} \right)
\end{aligned}$$

(note: basic donor-cell has significant numerical diffusion)

Example 5: a draft of an advection-equation solver

```
#include <blitz/array.h>
#include <array>

int main()
{
  // some compile-time constants
  enum { n_dims = 2, n_tlev = 2, x=0, y=1, n=0 };

  // ``run-time'' parameters
  int nx = 3, ny = 3, nt = 4;
  std::array<double, n_dims> C = {.5, .5};

  // objects representing ``psi'', ``i'' & ``j''
  std::array<blitz::Array<double, n_dims>, n_tlev> psi;
  blitz::Range i(1, nx), j(1, ny);

  // mem alloc + zeroing (C++11 for loop syntax)
  for (auto arr : {&psi[n], &psi[n+1]})
  {
    arr->resize(nx+2, ny+2);
    *arr = 0;
  }

```

Example 5: a draft of an advection-equation solver

```
  // output settings
  std::cout.setf(std::ios::fixed);
  std::cout.precision(2);

  // initial condition
  psi[n](i, j)(0, 0) = 1;
  std::cout << psi[n](i,j) << std::endl;

  // integration loop
  for (int t = 0; t < nt; ++t)
  {
    // donor-cell advection (constant C>0)
    psi[n+1](i,j) = psi[n](i,j)
      - (C[x] * psi[n](i, j) - C[x] * psi[n](i-1, j))
      - (C[y] * psi[n](i, j) - C[y] * psi[n](i, j-1));

    // output
    std::cout << psi[n+1](i,j) << std::endl;

    // ``old'' psi becomes ``new'' psi (no copying)
    blitz::cycleArrays(psi[n], psi[n+1]);
  }
}
```

Example 5: a draft of an advection-equation solver

```
$ ./a.out
(0,2) x (0,2)
[ 1.00 0.00 0.00
  0.00 0.00 0.00
  0.00 0.00 0.00 ]

(0,2) x (0,2)
[ 0.00 0.50 0.00
  0.50 0.00 0.00
  0.00 0.00 0.00 ]

(0,2) x (0,2)
[ 0.00 0.00 0.25
  0.00 0.50 0.00
  0.25 0.00 0.00 ]

(0,2) x (0,2)
[ 0.00 0.00 0.00
  0.00 0.00 0.38
  0.00 0.38 0.00 ]

(0,2) x (0,2)
[ 0.00 0.00 0.00
  0.00 0.00 0.00
  0.00 0.00 0.38 ]
```

blackboard

$$\begin{aligned}\psi^{n+1}_{i,j} = \psi^{n}_{i,j} \; &- \left(C_x \psi^n_{i,j} - C_x \psi^n_{i-1,j}\right) \\ &- \left(C_y \psi^n_{i,j} - C_y \psi^n_{i,j-1}\right)\end{aligned}$$

Blitz++

```
psi[n+1](i,j) = psi[n](i,j)
  - (C[x] * psi[n](i,   j  ) -
     C[x] * psi[n](i-1, j  ))
  - (C[y] * psi[n](i,   j  ) -
     C[y] * psi[n](i,   j-1));
```

- loop-free code
- execution within a single loop
- no temporary objects

Summary

Blitz++

- C++ multi-dimensional array containers suitable for high-performance computing

Summary

Blitz++

- C++ multi-dimensional array containers suitable for high-performance computing
- loop-free arithmetics

Summary

Blitz++

- C++ multi-dimensional array containers
  suitable for high-performance computing
- loop-free arithmetics
- works on array-expressions rather that arrays
  $\rightsquigarrow$ no temporary objects, single-loop traversals
  (by design, in Fortran it depends on compiler optimisations)

Summary

Blitz++

- C++ multi-dimensional array containers
  suitable for high-performance computing
- loop-free arithmetics
- works on array-expressions rather that arrays
  $\rightsquigarrow$ no temporary objects, single-loop traversals
  (by design, in Fortran it depends on compiler optimisations)
- OOP interface (incl. multi-dim. addressing with single object)
  $\rightsquigarrow$ no analogues in Fortran

Summary

Blitz++

- C++ multi-dimensional array containers
  suitable for high-performance computing
- loop-free arithmetics
- works on array-expressions rather that arrays
  $\rightsquigarrow$ no temporary objects, single-loop traversals
  (by design, in Fortran it depends on compiler optimisations)
- OOP interface (incl. multi-dim. addressing with single object)
  $\rightsquigarrow$ no analogues in Fortran
- C-pointer API for interfacing C, Fortran, NumPy, ...

Summary

Blitz++

- C++ multi-dimensional array containers
  suitable for high-performance computing
- loop-free arithmetics
- works on array-expressions rather that arrays
  $\rightsquigarrow$ no temporary objects, single-loop traversals
  (by design, in Fortran it depends on compiler optimisations)
- OOP interface (incl. multi-dim. addressing with single object)
  $\rightsquigarrow$ no analogues in Fortran
- C-pointer API for interfacing C, Fortran, NumPy, ...
- numerous alternatives:
  - C++: NT2, Armadillo, . . .
  - Python: NumPyPy, numexpr, . . .
  - Fortran: built-in arrays (not OO)

Summary

Blitz++

- C++ multi-dimensional array containers
  suitable for high-performance computing
- loop-free arithmetics
- works on array-expressions rather that arrays
  $\rightsquigarrow$ no temporary objects, single-loop traversals
  (by design, in Fortran it depends on compiler optimisations)
- OOP interface (incl. multi-dim. addressing with single object)
  $\rightsquigarrow$ no analogues in Fortran
- C-pointer API for interfacing C, Fortran, NumPy, ...
- numerous alternatives:
  - C++: NT2, Armadillo, ...
  - Python: NumPyPy, numexpr, ...
  - Fortran: built-in arrays (not OO)
- implemented using template meta-programming
  $\rightsquigarrow$ much easier to use with C++11

That's it for today.
Thanks for your attention!