Computation with C++

Keith O'Hara

01/17/2018

`C` & `C++`

- `C` is a low-level programming language above assembly
  - Examples: Gurobi, **R**, the Linux kernel, git, embedded systems.

C & C++

- C is a low-level programming language above assembly
  - Examples: Gurobi, **R**, the Linux kernel, git, embedded systems.
  - Can inline assembly code (see OpenBLAS)

C & C++

- C is a low-level programming language above assembly
  - Examples: Gurobi, **R**, the Linux kernel, git, embedded systems.
  - Can inline assembly code (see OpenBLAS)
  - TIOBE Index: C was the 'fastest grower of 2017' (!?!).

`C` & `C++`

- `C` is a low-level programming language above assembly
  - Examples: Gurobi, **R**, the Linux kernel, git, embedded systems.
  - Can inline assembly code (see OpenBLAS)
  - TIOBE Index: C was the 'fastest grower of 2017' (!?!).
- `C++` is sometimes described as '`C` with classes.'
  - But note: `C` $\not\subset$ `C++`.

`C` & `C++`

- `C` is a low-level programming language above assembly
  - Examples: Gurobi, **R**, the Linux kernel, git, embedded systems.
  - Can inline assembly code (see OpenBLAS)
  - TIOBE Index: C was the 'fastest grower of 2017' (!?!).
- `C++` is sometimes described as '`C` with classes.'
  - But note: `C` $\not\subset$ `C++`.
  - Memory management (constructors and destructors)

`C` & `C++`

- `C` is a low-level programming language above assembly
  - Examples: Gurobi, **R**, the Linux kernel, git, embedded systems.
  - Can inline assembly code (see OpenBLAS)
  - TIOBE Index: C was the 'fastest grower of 2017' (!?!).
- `C++` is sometimes described as '`C` with classes.'
  - But note: `C` $\not\subset$ `C++`.
  - Memory management (constructors and destructors)
  - A lot of Machine Learning libraries are written in `C++`

C & C++

- C is a low-level programming language above assembly
  - Examples: Gurobi, **R**, the Linux kernel, git, embedded systems.
  - Can inline assembly code (see OpenBLAS)
  - TIOBE Index: C was the 'fastest grower of 2017' (!?!).
- C++ is sometimes described as 'C with classes.'
  - But note: C $\not\subset$ C++.
  - Memory management (constructors and destructors)
  - A lot of Machine Learning libraries are written in C++
  - Modern C++: C++11 (and beyond)

`C` & `C++`

- `C` is a low-level programming language above assembly
  - Examples: Gurobi, **R**, the Linux kernel, git, embedded systems.
  - Can inline assembly code (see OpenBLAS)
  - TIOBE Index: C was the 'fastest grower of 2017' (!?!).
- `C++` is sometimes described as '`C` with classes.'
  - But note: `C` $\not\subset$ `C++`.
  - Memory management (constructors and destructors)
  - A lot of Machine Learning libraries are written in `C++`
  - Modern `C++`: `C++11` (and beyond)
- Both are static **type-defined** languages.

  `double a = 8.0;` compared to **R**'s '`a = 8`'

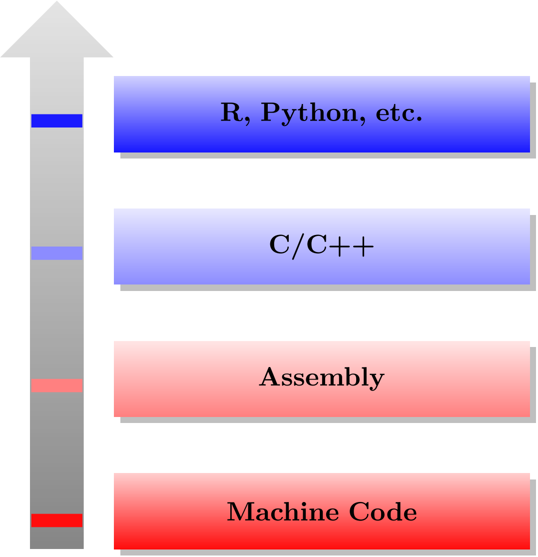
`C` & `C++`

- `C` is a low-level programming language above assembly
  - Examples: Gurobi, **R**, the Linux kernel, git, embedded systems.
  - Can inline assembly code (see OpenBLAS)
  - TIOBE Index: C was the 'fastest grower of 2017' (!?!).
- `C++` is sometimes described as '`C` with classes.'
  - But note: `C` $\not\subset$ `C++`.
  - Memory management (constructors and destructors)
  - A lot of Machine Learning libraries are written in `C++`
  - Modern `C++`: `C++11` (and beyond)
- Both are static **type-defined** languages.

  `double a = 8.0;` compared to **R**'s '`a = 8`'
- Since `C++11`:

  `auto a = 8.0;`

Compiled vs Interpreted Languages



Intro to C++

```cpp
#include <iostream>

int main()
{
    double a = 3.0, b = 2.0;
    double c = a + b;
    std::cout << "a+b=" << c << std::endl;

    return 0;
}
```

- Put into a file called `my_prog.cpp`
- Need a compiler! `gcc` vs `clang` vs `icc`.
  - macOS users: get Xcode from the App Store
  - Windows users: Rtools includes the `gcc` chain
- Open a terminal and type:

  ```
  g++ my_prog.cpp -o my_prog.out
  ```
- Run with: `./my_prog.out`

Intro to C++

- So what is the compiler actually doing?

  `g++ my_prog.cpp -o my_prog.out`

- Compilation process:

  (1) **Preprocessor**: cleanup, evaluate preprocessor directives (e.g., `-DDO_IF_DEF`), and include files
  (2) **Compiler** will generate assembly code
  (3) **Assembler** will generate machine code
  (4) The **linker** will connect any required outside libraries to your program

  Or

  $$\texttt{C/C++} \rightsquigarrow \text{Cleaned } \texttt{C/C++} \rightsquigarrow \text{S} \rightsquigarrow \text{Machine Code}$$

- To stop at (2) use -E; for (3) use -S; and for (4) use -c. E.g.,

  `g++ -S my_prog.cpp -o my_prog.S`

Example: Dijkstra

Algorithm 1 Dijkstra

procedure DIJKSTRA($s, \mathcal{X}, \mathcal{A}$) ▷ Solve for $\mathbf{d}^*$

Define the pair $(u_j, d_j) \in (\mathcal{X}, \mathbb{R})$, where $d_j := \text{dist}(u_s, u_j)$.

$u_s \in \mathcal{X}$, $\mathcal{V} = \mathcal{X}$; $\mathbf{d}^* = \boldsymbol{\infty}$, $d_s^* = 0$ ▷ Initialization

while $\mathcal{V} \neq \{\emptyset\}$ **do**

Choose $u_i := \{u_j \in \mathcal{V} : d_j^* = \min_k\{d_k^*\}\}$.

if $d_j^* = \infty$ **then**

break;

$\mathcal{V} = \mathcal{V} \backslash \{u_i\}$ ▷ Remove u_i

Define the adjacent network to u_i by $N(u_i)$.

For each $v_j \in N(u_i) \cap \mathcal{V}$, with arc weights $\{a_{i,j}\}$, calculate

$$c_j = d_i^* + a_{i,j}$$

if $c_j < d_j^*$ **then**

$d_j^* = c_j$ ▷ Update d_j^*

return $\mathbf{d}^* = \{d_j^*\}_{j \in \mathcal{X}}$. ▷ Solution

Example: Dijkstra

- Recall the NYC roads example: approx 67k nodes, with 170k arcs

Example: Dijkstra

- Recall the NYC roads example: approx 67k nodes, with 170k arcs
- Gurobi takes around 4 secs to solve for the min-dist path

Example: Dijkstra

- Recall the NYC roads example: approx 67k nodes, with 170k arcs
- Gurobi takes around 4 secs to solve for the min-dist path
- SPR takes around 1/100 of a second

Example: Dijkstra

- Recall the NYC roads example: approx 67k nodes, with 170k arcs
- Gurobi takes around 4 secs to solve for the min-dist path
- SPR takes around 1/100 of a second
- Why the difference?

Example: Dijkstra

- Recall the NYC roads example: approx 67k nodes, with 170k arcs
- Gurobi takes around 4 secs to solve for the min-dist path
- SPR takes around 1/100 of a second
- Why the difference?
  - Efficient containers using the standard library: sparse data formats and efficiently ordered `std::set`.
  - `C++` move instructions
  - Loop unrolling

Loop Unrolling

Loop Unrolling

- Loop 'unrolling' is touted as a major benefit of `C/C++`
- But takes some work to get right
- Parallelism on modern computers:
  - Single instruction, multiple data (SIMD) ; Streaming SIMD Extensions (SEE); Advanced Vector Extensions (AVX).
  - OpenMP and SIMD
- Compiler optimization flags:

  `-O3` vs `-Ofast`; `-march=native`; `-funroll-loops` with gcc
- Generally you need to inspect the **assembly code**!

Loop Unrolling: Example

- Consider the loop calculating: $a(i) = a(i) + k \times b(i)$, $i \in [n]$.
- In `C++` we could write this as:

```
void add_vecs(float* a, float* b, float k, int n)
{
    for (int i=0; i<n; i++) {
        a[i] += k*b[i];
    }
}
```

- Suppose $n = 1\text{E}05$ and repeat 1000 times.
- (Optimized) **R** takes around 12 seconds.

Loop Unrolling: Example

- Modify the code slightly with a hint to the compiler (`__restrict`) and an OpenMP pragma.

```
void add_vec(float* __restrict a, float* __restrict b,
             const float k, int n) {
    #pragma omp parallel for
    for (int i=0; i<n; i++) {
        a[i] += k*b[i];
    }
}
```

Loop Unrolling: Example

- Modify the code slightly with a hint to the compiler (`__restrict`) and an OpenMP pragma.

```
void add_vec(float* __restrict a, float* __restrict b,
             const float k, int n) {
    #pragma omp parallel for
    for (int i=0; i<n; i++) {
        a[i] += k*b[i];
    }
}
```

- Let's inspect the assembly code! Compiled with

```
clang++-mp-5.0 -std=c++11 -Ofast -march=native -fopenmp \
simple_float_run.cpp -o simple_float.out
```

Loop Unrolling: Example

- Modify the code slightly with a hint to the compiler (`__restrict`) and an OpenMP pragma.

```
void add_vec(float* __restrict a, float* __restrict b,
             const float k, int n) {
    #pragma omp parallel for
    for (int i=0; i<n; i++) {
        a[i] += k*b[i];
    }
}
```

- Let's inspect the assembly code! Compiled with

  ```
  clang++-mp-5.0 -std=c++11 -Ofast -march=native -fopenmp \
  simple_float_run.cpp -o simple_float.out
  ```

- Without OpenMP: 0.026 secs. With OpenMP: 0.01 secs.

Loop Unrolling: Example

- Modify the code slightly with a hint to the compiler (`__restrict`) and an OpenMP pragma.

```
void add_vec(float* __restrict a, float* __restrict b,
             const float k, int n) {
    #pragma omp parallel for
    for (int i=0; i<n; i++) {
        a[i] += k*b[i];
    }
}
```

- Let's inspect the assembly code! Compiled with

```
clang++-mp-5.0 -std=c++11 -Ofast -march=native -fopenmp \
simple_float_run.cpp -o simple_float.out
```

- Without OpenMP: 0.026 secs. With OpenMP: 0.01 secs.
- tl;dr: **R** takes (>) 1000 times longer than `C++`.

Templates

Template Programming

- Templates are an approach to generic programming in a type-defined language.
- Frequently used to avoid tedious function overloading.

  Example: calculate the maximum of two numbers.

```
int max(int a, int b)
{
    int res = (a > b) ? a : b;
    return res;
}

// template version
template <typename T>
T max(T a, T b)
{
    return (a > b) ? a : b;
}
```

Template Metaprogramming

- Second use: compile-time computation (vs run-time).

```
template <int n> struct Factorial {
    static const int result = n * Factorial<n-1>::result;
};

// specialization in 0! case
template <> struct Factorial<0> {
    static const int result = 1;
};

int main()
{
    std::cout << Factorial<10>::result << std::endl;
    return 0;
}
```

- Pros and Cons?

Template Metaprogramming

- New style with C++11:

```
constexpr
int
factorial(const int x) {
    return ( x == 0 ? 1 : x == 1 ? x : x*factorial(x-1));
}

int main()
{
    constexpr int x = 10;
    constexpr int res = factorial(x);

    return 0;
}
```

Compile-time Computation

- Compilation with `-O0`:

```
_main:                                          ## @main
        .cfi_startproc
## BB#0:
        push    rbp
Lcfi0:
        .cfi_def_cfa_offset 16
Lcfi1:
        .cfi_offset rbp, -16
        mov     rbp, rsp
Lcfi2:
        .cfi_def_cfa_register rbp
        xor     eax, eax
        mov     dword ptr [rbp - 4], 0
        mov     dword ptr [rbp - 8], 10
        mov     dword ptr [rbp - 12], 3628800
        pop     rbp
        ret
        .cfi_endproc
```

Compile-time Computation

- You might be thinking...

Compile-time Computation

- You might be thinking...
  That's a really silly example, Keith!

Compile-time Computation

- You might be thinking...
  That's a really silly example, Keith!
  How many other examples could there be?

Compile-time Computation

- You might be thinking...

  That's a really silly example, Keith!

  How many other examples could there be?
- Answer: A lot of functions possess recursive representations

Compile-time Computation

- You might be thinking...

  That's a really silly example, Keith!

  How many other examples could there be?
- Answer: A lot of functions possess recursive representations
- Example:

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \exp(-t^2) dt$$

  can be (well-) approximated using a continued fraction representation

$$\text{erf}(x) = \frac{2x}{\sqrt{\pi}} \exp(-x^2) \cfrac{1}{1 - 2x^2 + \cfrac{4x^2}{3 - 2x^2 + \cfrac{8x^2}{5 - 2x^2 + \cfrac{12x^2}{7 - 2x^2 + \ddots}}}}$$

Deeper Down the Rabbit Hole...
OpenBLAS and Optimized GEMM

OpenBLAS

- BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra PACKage)
- OpenBLAS is a fork of GotoBLAS (written by Kazushige Goto)
- **Hand-optimized assembly code** for `gemm`

OpenBLAS

- BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra PACKage)
- OpenBLAS is a fork of GotoBLAS (written by Kazushige Goto)
- **Hand-optimized assembly code** for `gemm`
  - Optimization focused on use of L1 and L2 caches, TLB misses, and specialized architecture instructions

OpenBLAS

- BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra PACKage)
- OpenBLAS is a fork of GotoBLAS (written by Kazushige Goto)
- **Hand-optimized assembly code** for `gemm`
  - Optimization focused on use of L1 and L2 caches, TLB misses, and specialized architecture instructions
- Get using:

```
git clone -b develop --single-branch \
https://github.com/xianyi/OpenBLAS ~/Desktop/OBLAS
```

OpenBLAS

- BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra PACKage)
- OpenBLAS is a fork of GotoBLAS (written by Kazushige Goto)
- **Hand-optimized assembly code** for `gemm`
  - Optimization focused on use of L1 and L2 caches, TLB misses, and specialized architecture instructions
- Get using:

```
git clone -b develop --single-branch \
https://github.com/xianyi/OpenBLAS ~/Desktop/OBLAS
```

- Build using:

```
cd ~/Desktop/OBLAS
make CC=gcc FC=gfortran
sudo make PREFIX=/usr/local install
```

OpenBLAS

- BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra PACKage)
- OpenBLAS is a fork of GotoBLAS (written by Kazushige Goto)
- **Hand-optimized assembly code** for `gemm`
  - Optimization focused on use of L1 and L2 caches, TLB misses, and specialized architecture instructions
- Get using:

```
git clone -b develop --single-branch \
https://github.com/xianyi/OpenBLAS ~/Desktop/OBLAS
```

- Build using:

```
cd ~/Desktop/OBLAS
make CC=gcc FC=gfortran
sudo make PREFIX=/usr/local install
```

- Can easily link with **R** (replace Rblas or build **R** from source)

Example: FMA Optimization

- One important aspect of hand-optimized assembly code relates to advanced processor instructions, such as fused multiply-add (FMA):

$$a \leftarrow b + c \times d$$

- In code:

Example: FMA Optimization

- One important aspect of hand-optimized assembly code relates to advanced processor instructions, such as fused multiply-add (FMA):
$$a \leftarrow b + c \times d$$
- In code:

```
double add_m3(double x1, double x2, double x3)
{
    return x1 + x2*x3;
}
```

Example: FMA Optimization

- One important aspect of hand-optimized assembly code relates to advanced processor instructions, such as fused multiply-add (FMA):

$$a \leftarrow b + c \times d$$

- In code:

```
double add_m3(double x1, double x2, double x3)
{
    return x1 + x2*x3;
}
```

- Using -O3

```
mulsd   xmm1, xmm2
addsd   xmm0, xmm1
pop     rbp
ret
```

Example: FMA Optimization

- One important aspect of hand-optimized assembly code relates to advanced processor instructions, such as fused multiply-add (FMA):

$$a \leftarrow b + c \times d$$

- In code:

```c
double add_m3(double x1, double x2, double x3)
{
    return x1 + x2*x3;
}
```

- Using `-Ofast -march=native`

```
vfmadd231sd    xmm0, xmm2, xmm1
pop     rbp
ret
```

Linear Algebra with `C++`: Armadillo

Armadillo

- Armadillo is a templated `C++` linear algebra library
- Written by Conrad Sanderson at NICTA/Data61
- Template metaprogramming & **static polymorphism**
- Syntax similar to Matlab's

Armadillo	Matlab
C = A.t() * B	C = A' * B
C = A%B	C = A. * B
C = solve(A,B)	C = A\B
int n = A.n_rows	n = size(A,1)

- Link against **OpenBLAS** or some other system BLAS/LAPACK

Armadillo + OpenBLAS

- Consider

$$Z = A \times B \times C \times D$$

where the dimensions are

$$A = 1000 \times 800, \;\; B = 800 \times 600$$
$$C = 600 \times 400, \;\; D = 400 \times 200$$

Armadillo + OpenBLAS

- Consider
$$Z = A \times B \times C \times D$$
where the dimensions are
$$A = 1000 \times 800, \;\; B = 800 \times 600$$
$$C = 600 \times 400, \;\; D = 400 \times 200$$
- How do we compute this efficiently?

Armadillo + OpenBLAS

- Consider

$$Z = A \times B \times C \times D$$

where the dimensions are

$$A = 1000 \times 800, \;\; B = 800 \times 600$$
$$C = 600 \times 400, \;\; D = 400 \times 200$$

- How do we compute this efficiently?
- Do this 1000 times.

Run Time (sec.)

R	**R** & OpenBLAS	`C++` & OpenBLAS
475.208	18.01	3.96