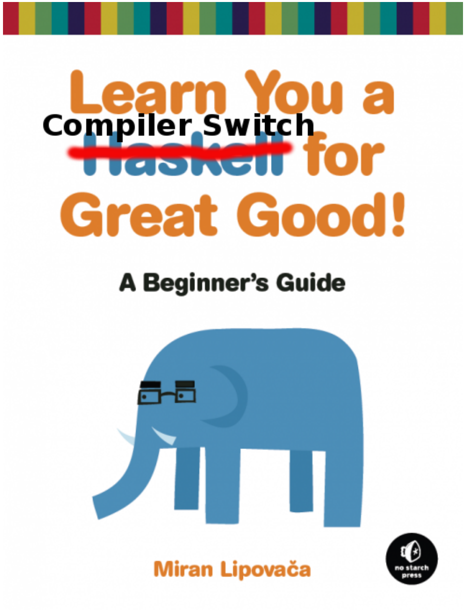
Improving Performance Through Compiler Switches- Examples from Scientific Computing

CppCon 2016

Tim Haines

University of Wisconsin - Madison
Department of Astronomy

September 23, 2016


Learn You a
Compiler Switch
~~Haskell~~ for
Great Good!
A Beginner's Guide
Miran Lipovača
no starch press

Scientific Computing

- Application of high performance computing (HPC) to solving scientific problems
- Today's standard is petaflop computing (10^{15} FLOPS)
- Usually done on 10s of thousands of CPU cores
  - More accelerators are being used every year
- Performance matters... enormously

  Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. ***Yet we should not pass up our opportunities in that critical 3%***

- In SC, it's more like 80/20 where we spend 90+% of our time in the 20% of code

Why Performance Matters

Machine	nCores	Peak TFLOPS	Power (kW)
Sunway TaihuLight	10,649,600	125,435.9	15,371
Tianhe-2	3,120,000	54,902.4	17,808
Titan	560,640	27,112.5	8,209
Sequoia	1,572,864	20,132.7	7,890
Fujitsu	705,024	11,280.4	12,660

1MW can power $\sim$ 1000 homes

In 2012, US Department of Energy pushed back Exascale Computing (10^{18} FLOPS) to 2023 (at the earliest) because of power needs (enough to power Bellevue).

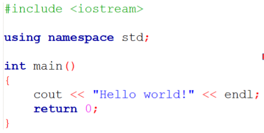
Why Performance Matters (cont.)

What does this mean for us today?

- Density must go up!
  - Accelerators (GPUs, APUs, Xeon Phi, etc.)
  - Intel Purley Skylake server CPU: 28 cores, 54 threads
  - AMD Opteron "Zen" CPU: 32 cores, 64 threads
- Making better utilization of today's machines is imperative
  - This extends well beyond just SC (see WG14 and attendance at this year's low-latency class)
- There is so much existing software, how do we get started making it better?
  - Two options:
    - Re-write the code to make better use of hardware
    - Instruct the compiler to generate better code
  - **These are not mutually exclusive!**

Here be Dragons

This talk will focus on instructing the compiler to generate better code



```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

There are numerous excellent previous (and current!) CppCon, GoingNative, MeetingC++, and C++Now! talks that discuss how compilers try to make the best of your code (see the appendix for a brief list).

Taming the Dragons

To get the best code, the compiler needs to know

- allowed transformations of your code
- target architecture
  - Possibly most important, but hardest to do generically
- space/time tradeoffs you are willing to allow

To do this, we use lots of compiler switches

Quick Quiz

- How many switches does gcc-6.2 support?
  - 2257

Compiler Switches 101

Switches come in essentially three variants (these are made up by me)

1. Compile-time static analysis
   - 1.1 -Wall and friends
2. Profiling and analysis
   - 2.1 PGO (gprof/gcov), -fsanitizer=X
3. Code generation
   - 3.1 -Ox, -march, -ffast-math
   - 3.2 Also LTO, but I don't have time to talk about it today

Is -O3 -OK?

There are several reasons not to use -O3, including

- Correctness of your code depends on UB
  - e.g., you break the aliasing rules
- You don't have unit, coverage, or load tests
- FUD

In scientific computing, it is said that high levels of optimization shouldn't be used because

- It will cause errors in calculations
- The code bloat isn't worth the possible speedup

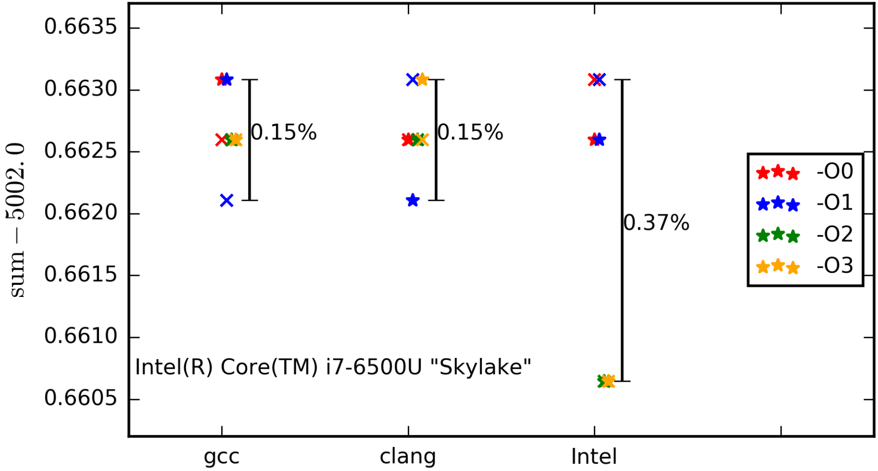
Let's test these "hypotheses."

Testing the Optimizer

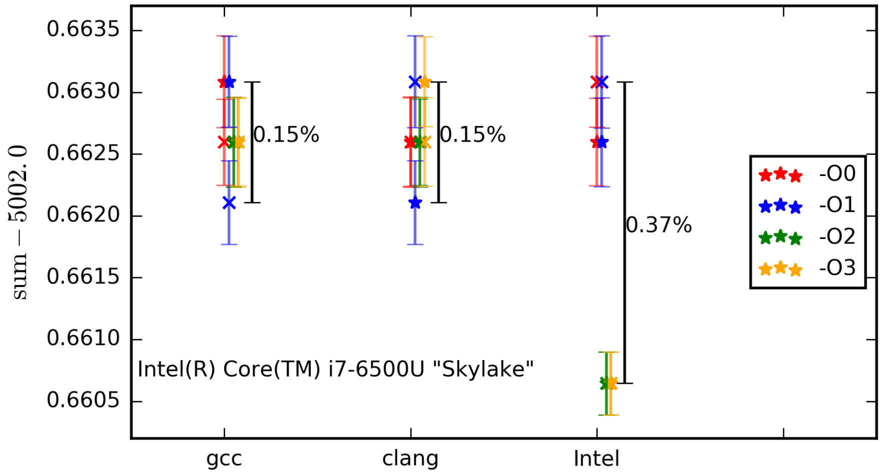
A simple reduction operation test

- Recall, FP arithmetic is not associative
  - $x + (y + z) \neq (x + y) + z$ (see John Farrier's excellent *Demystifying Floating Point* from CppCon2015)
  - This means ILP (SIMD) and out-of-order execution will likely affect the result
- 10,000 single-precision floats uniformly generated in $[0.1, 0.9)$ (Why?)
  - Maximizes precision
- Three industry-standard compilers: gcc-6.2, clang-3.9, Intel® 17.0.0
  - VC doesn't exist in my world
- Two algorithms: `std::accumulate` and `operator[]`
- Four optimization levels: -O0 to -O3

Testing the Optimizer: Arithmetic Errors



Testing the Optimizer: Arithmetic Errors



Testing the Optimizer: Code Size vs Runtime

Program	Compiler	.text Size (b)			Time		
		-Os	-O2	-O3	-Os	-O2	-O3
HPCG	gcc	54322	+91%	+133%	0.0545s	-12%	-28%
	clang	65058	+41%	+39%	0.0395s	-0.05%	+8%
	Intel	92882	+122%	+123%	0.0381s	-7%	-8%
Template	gcc	674	-19%	-19%	340ns	-38%	-39%
	clang	3074	-17%	-77%	517ns	-40%	-40%
	Intel	5762	+2%	+2%	1687ns	-85%	-85%

GCC has the largest range in performance and size based on optimization flags. We will see why this is so, shortly. The Intel compiler isn't very good at dealing with templates.

Under the Hood

Let's look at what's going on when we use `-O3`.
(gcc only for now because it has the best documentation)

- clang and gcc default to `-O0`, but Intel defaults to `-O2`
- `-O3` enables all of `-O2` which enables all of `-O1` ***and***
- -finline-functions
  - Consider more functions for inlining (even if not declared inline)
  - clang and Intel enable this at `-O2`
- -ftree-loop-vectorize
  - Perform loop vectorization on trees (very important!)
- -ftree-slp-vectorize
  - Perform basic block vectorization on trees
- -funswitch-loops
  - Move branches with loop invariant conditions out of the loop

Under the Hood

- -ftree-loop-distribute-patterns
  - Perform loop distribution of patterns that can be generated with calls to a library

This converts

```
for(int i=0; i<N; i++) {
    A[i] = 0;
    B[i] = A[i] + i;
}
```

to

```
for(int i=0; i<N; i++) A[i] = 0;
for(int i=0; i<N; i++) B[i] = A[i] + i;
```

and replaces the first loop with a call to `memset`.

Under the Hood

- And a few more switches that enable technical details such as register scheduling and memory load/store motion

If you aren't sure what your compiler is doing, they provide query mechanisms

- gcc: `gcc -O2 -Q --help=optimizers`
- clang: `llvm-as < /dev/null | opt -O2 -disable-output -debug-pass=Arguments`
- Intel: ?

From Dragons to Demons

Let's talk about `-ffast-math`

- QUIZ: How many think this is safe to turn on all the time?
  - The Intel compiler has this on **by default**
- It's really many switches rolled into one
- -fno-math-errno
  - Don't set `errno` when using a single instruction, e.g., `sqrt`
- -ffinite-math-only
  - Assume arguments and results are not `NaN`s or `+-Inf`s
- -fno-rounding-math
  - Don't assume a default FP rounding behavior (on by default)
- -fno-signaling-nans
  - Assume IEEE sNaNs don't generate user-visible traps
- -funsafe-math-optimizations
  - this is really even more switches!

How Unsafe is unsafe-math-optimizations?

Allow FP arithmetic optimizations that (a) assume that arguments and results are valid and (b) may violate IEEE or ANSI standards.

- -fno-signed-zeros
  - Allow FP arithmetic that ignores the signedness of zero
  - `-(a - b) == b - a`
- -fno-trapping-math
  - Assume FP operations cannot generate user-visible traps
- -fassociative-math
  - Allow re-association of operands in series of FP operations
- -freciprocal-math
  - Allow `a / b => a * (1 / b)`
  - `(v)rcp(p|s)s` guarantees this is good to $1.5 \times 2^{-12} \sim 10^{-5.7}$ (recall $\epsilon \sim 1.2 \times 10^{-7}$)

Know Your Architecture

Knowing your architecture can be hard, but offers allows the compiler to generate better code.

- It is hard because you can't always know the architecture your code will run on
  - The world of pre-built binaries makes this hard!
  - Open Source makes this very easy!
- Knowing your target architecture allows the compiler to target your ISA
  - -m(SSE|SSE2|SSE3|SSSE3|SSE4.1|SSE4.2)
  - -m(AVX|AVX2|FMA3|FMA4|AVX512)
  - All three compilers support `-march=native`
- Timur Doumler's talk from Tuesday entitled *Want fast C++? Know your hardware!*
- Matt P. Dziubinski's talk from Monday entitle *Computer Architecture, C++, and High Performance*

A Few Miscellanea

- -ftree-loop-distribution (gcc only)

  ```
  for(int i=0; i<N; i++) {
      A[i] = B[i] + c;
      D[i] = E[i] * f;
  }
  ```

  becomes

  ```
  for(int i=0; i<N; i++) A[i] = B[i] + c;
  for(int i=0; i<N; i++) D[i] = E[i] * f;
  ```

- -funroll-loops
  - Statically unroll loops with known trip counts
  - This is a very large space/time trade-off
  - On by default in Intel and at `-O2` in clang
- -fstrict-aliasing
  - Enabled at -O2 in gcc; on by default in clang and icc

A More Detailed Example

- The famous SAXPY from the Fortran BLAS library
  - 33M single-precision floats $\gg$ all caches
- Three algorithms
  - expression templates with std::transform
  - expression templates with a C-style accumulator
  - expression templates with AVX intrinsics
- Combinations of many switches
  - `-O2`, `-O3`, `-O3 -ffast-math`
  - `-march=x86-64`, `-march=native`
- Two machines
  - Intel(R) Core(TM) i7-6500U CPU “Skylake” Dual Core + Hyperthreading (Q3 2015)
  - AMD FX(tm)-8350 “Bulldozer v2” 8-core (Q3 2012)
- Using one thread

A Quick Diversion

Anatomy of a SIMD instruction

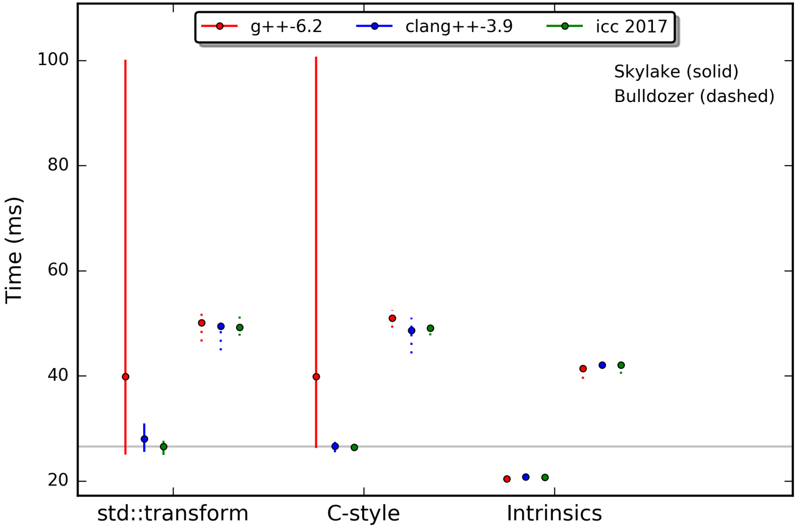
V	ADD/SUB/MUL		P/S	S/D
V	MOV	U/A	P/S	S/D

ADDSS - SSE add ***one*** scalar SP float

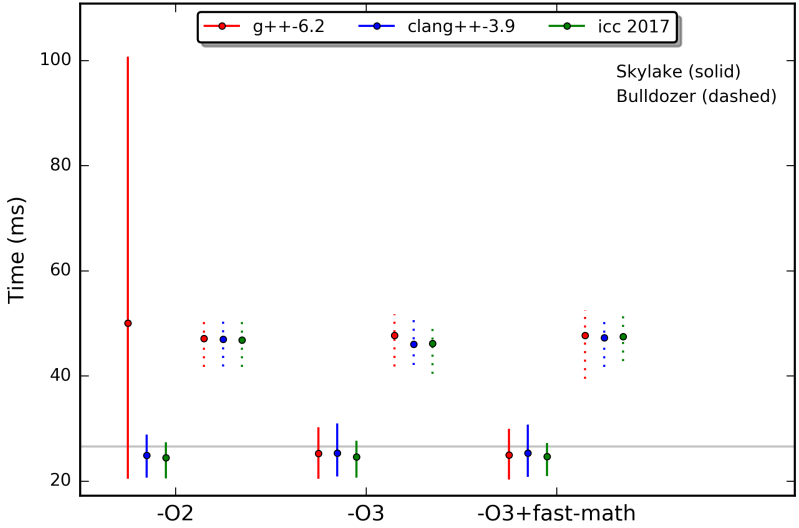
VMOVAPD - AVX move aligned packed DP floats

VFMADD123PS - Fuse Multiply-Add packed SP floats

A More Detailed Example (results)



A More Detailed Example (results)



Compilers Gone Wild

gcc

```
movss  xmm0,DWORD PTR [rip+0x850]
mulss  xmm0,DWORD PTR [rdx-0x4]
addss  xmm0,DWORD PTR [rcx-0x4]
movss  DWORD PTR [rax-0x4],xmm0
```

Compilers Gone Wild

clang

```
movaps xmm0,XMMWORD PTR [rip+0xba7]
movups xmm1,XMMWORD PTR [rbx+rsi*4]
movups xmm2,XMMWORD PTR [r15+rsi*4]
mulps  xmm1,xmm0
addps  xmm1,xmm2
movups XMMWORD PTR [r14+rsi*4],xmm1
movups xmm1,XMMWORD PTR [rbx+rsi*4+0x10]
movups xmm2,XMMWORD PTR [r15+rsi*4+0x10]
mulps  xmm1,xmm0
addps  xmm1,xmm2
movups XMMWORD PTR [r14+rsi*4+0x10],xmm1
```

Compilers Gone Wild

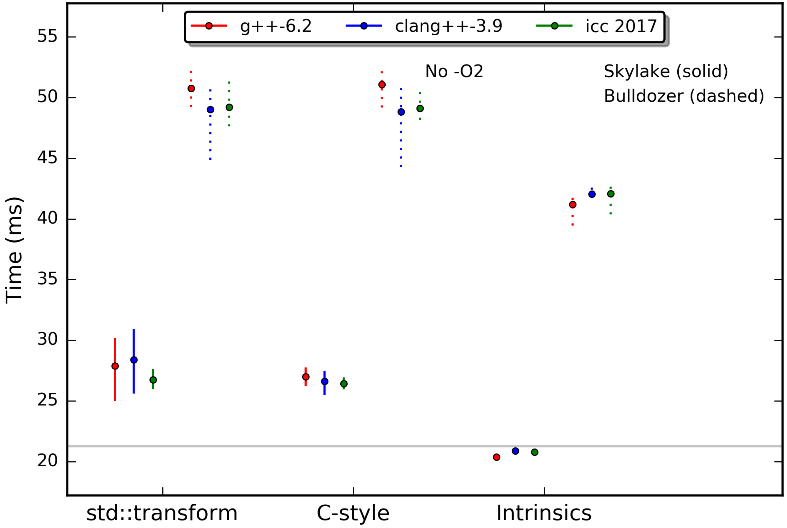
Intel

```
movups xmm3,XMMWORD PTR [rdx+r8*4]
movups xmm2,XMMWORD PTR [rsi+r8*4]
mulps  xmm3,xmm0
addps  xmm3,xmm2
movups XMMWORD PTR [rcx+r8*4],xmm3
movups xmm5,XMMWORD PTR [rdx+r8*4+0x10]
movups xmm4,XMMWORD PTR [rsi+r8*4+0x10]
mulps  xmm5,xmm0
addps  xmm5,xmm4
movups XMMWORD PTR [rcx+r8*4+0x10],xmm5
```

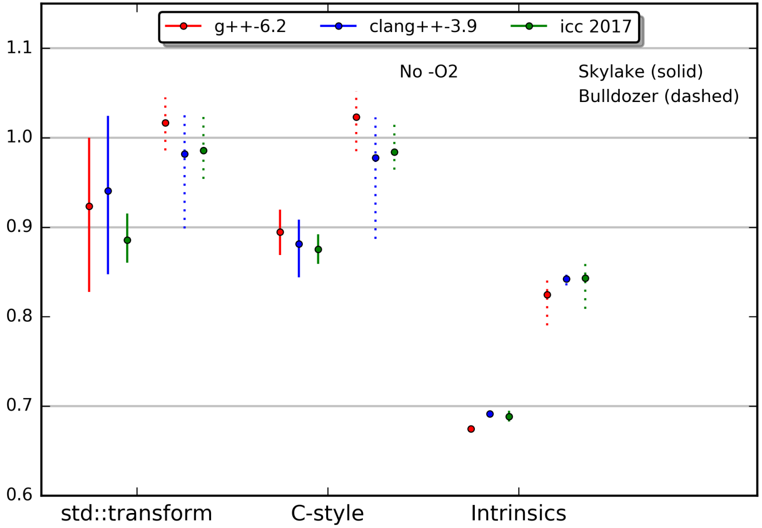
A More Detailed Example (results)

```
vmovups     ymm0,YMMWORD PTR [rdx+r8*1]
vmovaps     ymm7,YMMWORD PTR [rcx+r8*1]
vfmadd132ps ymm0,ymm7,YMMWORD PTR [rip+0xf22]
vmovups     YMMWORD PTR [rsi+r8*1],ymm0
```

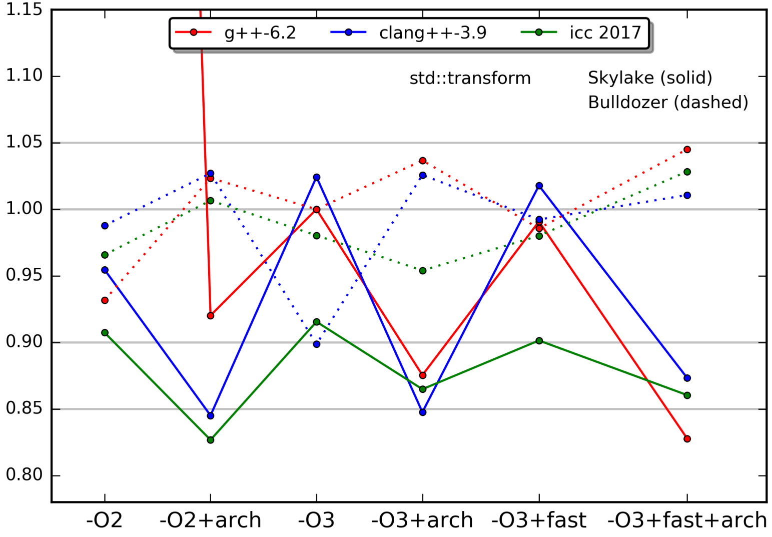
A More Detailed Example (results)



A More Detailed Example (results)



A More Detailed Example (results)



Appendix

- Chandler Carruth. Understanding Compiler Optimizations. Meeting C++, 2015
- Chandler Carruth. Tuning C++: Benchmarks, and CPUs, and Compilers! Oh My!. CppCon, 2015
- Eric Brumer. Compiler Confidential. GoingNative, 2013
- Scott Meyers. CPU Caches and Why You Care. code::dive, 2014
- John Farrier. Demystifying Floating Point. CppCon, 2015
- Timur Doumler. Want fast C++? Know your hardware!. CppCon, 2016
- Matt P. Dziubinski. Computer Architecture, C++, and High Performance. CppCon, 2016

The code is in my github repo:
https://github.com/hainest/Presentations/tree/master/CppCon2016

Extra

```
const __m256 a = _mm256_set1_ps(sp.a);

const auto x = sp.x.data_;
const auto y = sp.y.data_;
auto c = data_;

for (size_t i = 0; i < size; i+=16) {
    const __m256 x1 = _mm256_load_ps(x + i + 0);
    const __m256 x2 = _mm256_load_ps(x + i + 8);
    const __m256 y1 = _mm256_load_ps(y + i + 0);
    const __m256 y2 = _mm256_load_ps(y + i + 8);

    const __m256 r1 = _mm256_fmadd_ps(a, x1, y1);
    const __m256 r2 = _mm256_fmadd_ps(a, x2, y2);

    _mm256_stream_ps(c + i + 0, r1);
    _mm256_stream_ps(c + i + 8, r2);
}
```