Pacific Northwest
NATIONAL LABORATORY

Benchmarking Modern C++ Abstraction Penalty

February 25th, 2019

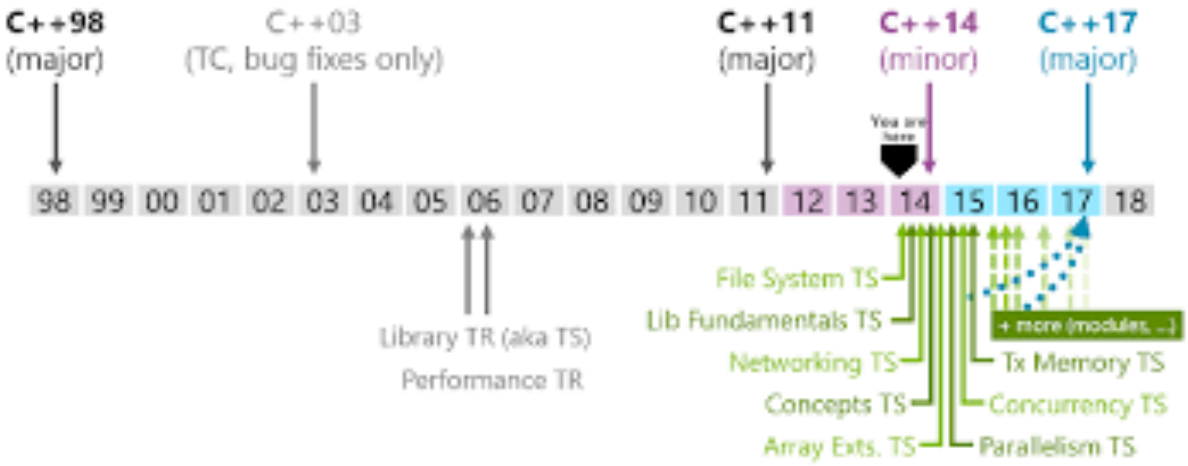
Marcin Zalewski, Andrew Lumsdaine

U.S. DEPARTMENT OF ENERGY
BATTELLE

PNNL is operated by Battelle for the U.S. Department of Energy

The Good, The Bad, and The Ugly

- Modern C++ is here!
  - C++98/C++03 was better than most
  - We waited for something new for a very long time
  - Now the compilers and the community caught up
  - Abstraction is (or should be) piling on



- However…
  - Have you ever heard how C must be faster because it is lower level?
  - More abstraction must mean that your code will be slower?
  - Of course, we C++ programmers know how wrong all of the above is*
- Oh no
  - Compilers vary widely
  - Who knows how much does abstraction cost?
  - How often do programmers think they are smarter then the compiler?

* It may take hours to compile

So What Is That Abstraction You Are Talking About?

- Abstraction
  - Low-Level language constructs
    - ✓ Move semantics
    - ✓ Type deduction
    - ✓ Lambdas
    - ✓ Structured bindings
    - ✓ Range for loops
    - ✓ Variadic templates
    - ✓ Template aliases
    - ✓ Class enums
    - ✓ Static assert
  - Higher-level abstraction
    - ✓ Tuple types
    - ✓ Hash tables
    - ✓ Smart pointers
    - ✓ Threading
    - ✓ Polymorphic function object wrappers
    - ✓ Type traits for metaprogramming
  - DSLs
    - ✓ Exploit C++ features to create an embedded DSL
    - ✓ Library interfaces that closely correspond to the problem domain
    - ✓ Modern feel: use modern C++ features

What Do We Want to Find Out?

- Should we avoid abstraction?
  - Ideal abstraction
  - May be pretty far from hand-optimized code
  - Often there is a temptation to pre-optimize
  - Lack of confidence in the compiler's ability
  - Compilation time
  - Compile-time errors
- How to measure abstraction penalty?
  - In general, this topic receives little attention
  - There are few benchmarks available
  - Compilers differ widely (even between versions)
- Let's look at some benchmarks

Example: Abstract Sparse Matvec

- Let's start with a simple familiar loop nest:

```
zeroize(y);

std::vector<size_t> indices(N*N+1);
std::vector<size_t> indexed(0);
std::vector<double> values(0);
piscetize(indices, indexed, values, N, N);

for (size_t i = 0; i < N*N; ++i) {
  for (size_t j = indices[i]; j < indices[i+1]; ++j) {
    y[i] += values[j] * x[indexed[j]];
  }
}
```

Matvec: What's The Intent?

- y=A*x: For every element A(i, j), multiply it with x[j] and accumulate to y[i]

```
edge_list<directed, double> A(0);
piscetize(A, N, N);
adjacency<0, double> C(A);

for (auto&& [i, j, v] : spmv_range(C)) {
  y[i] += v * x[j];
}
```

- Can this possibly work well?

How About This?

- Use std::for_each

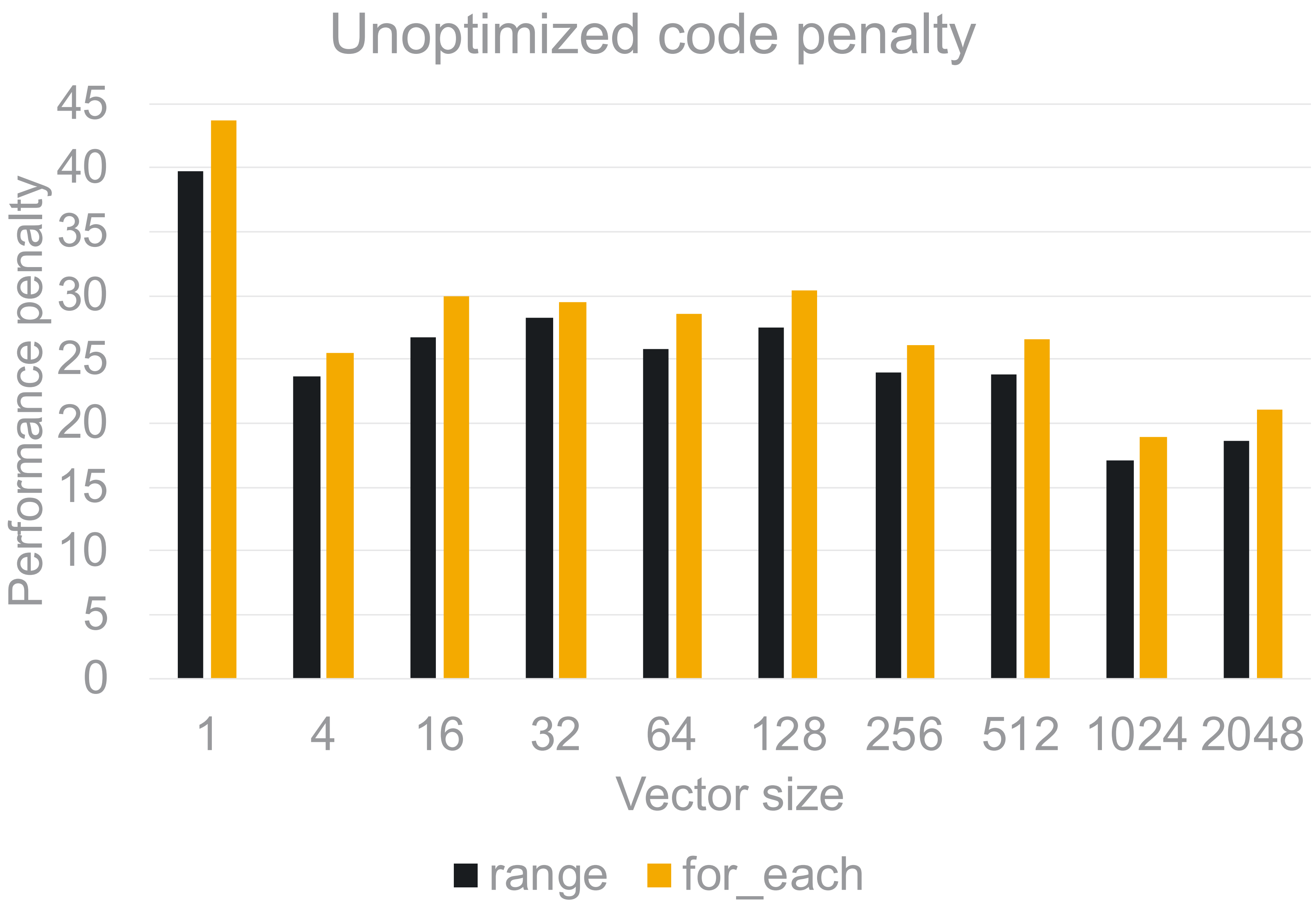
```
edge_list<directed, double> A(0);
piscetize(A, N, N);
adjacency<0, double> C(A);

std::for_each(D.begin(), D.end(),
  [&](std::tuple<size_t, size_t, double>&& a) {
    y[std::get<0>(a)] += std::get<2>(a) * x[std::get<1>(a)];
  }
);
```

What is spmv_range?

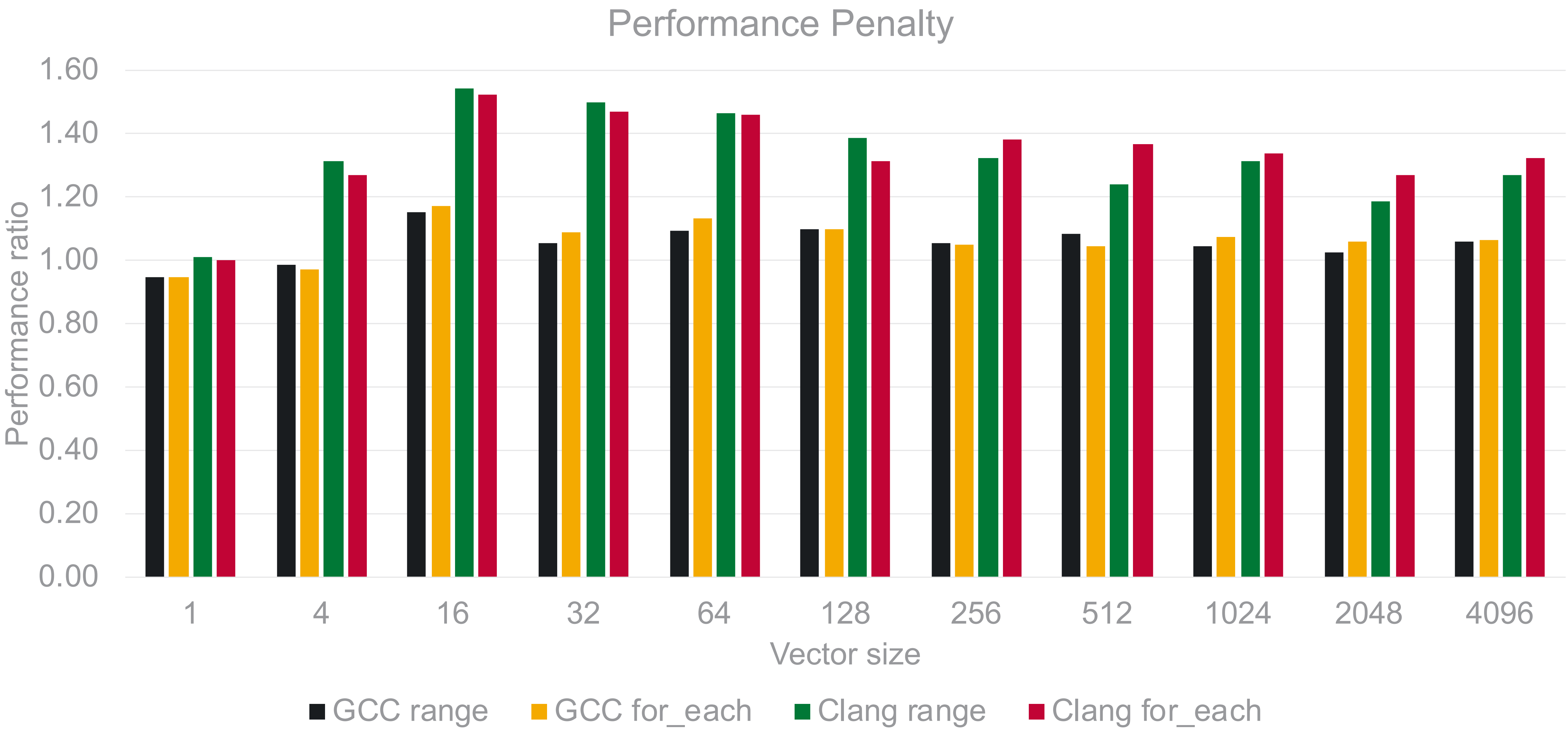
```
v_range_iterator& operator++() {
  ++u_begin;
  if (u_begin == u_end) {
    if (++first != last) {
      u_begin = (*first).begin();
      u_end = (*first).end();
    }
  }
  return *this;
}

auto operator*() {
  return std::tuple{first - the_range_.the_graph_.begin(),
                    std::get<0>(*u_begin),
                    std::get<1>(*u_begin)};
}

auto operator==(const v_range_iterator& b) const
  { return first == b.first; }
```

```
template<typename Graph>
class spmv_range {
public:
  spmv_range(Graph& g) : the_graph_(g) {}
  spmv_range(const spmv_range& b) : the_graph_(b.the_graph_) {}
  auto& operator=(const spmv_range& b) {
    the_graph_ = b.the_graph_;
    return *this;
  }

  class v_range_iterator {
  private:
    spmv_range<Graph>&              the_range_;
    typename Graph::outer_iterator first, last;
    typename Graph::inner_iterator u_begin, u_end;

  public:
    v_range_iterator(spmv_range<Graph>& range, size_t offset = 0)
        : the_range_{range}, first{the_range_.the_graph_.begin() + offset}, last{the_range_.the_graph_.end()} {
      if (first != last) {
        u_begin = (*first).begin();
        u_end   = (*first).end();
      }
    }

    v_range_iterator(const v_range_iterator& b)
        : the_range_(b.the_range_), first(b.first), last(b.last), u_begin(b.u_begin), u_end(b.u_end) {}

    v_range_iterator& operator++() {
      ++u_begin;
      if (u_begin == u_end) {
        if (++first != last) {
          u_begin =
              (*first).begin();    // FIXME:  This was commented out -- why?  Might not need it for CSR -- but in general, we do
          u_end = (*first).end();
        }
      }

      return *this;
    }

    // auto operator*() { return std::tuple<vertex_index_t, vertex_index_t&&>(last - first, std::get<0>(*u_begin)); }

    auto operator*() {
      return std::tuple<typename std::iterator_traits<typename Graph::outer_iterator>::difference_type, vertex_index_t, double&>(
          first - the_range_.the_graph_.begin(), std::get<0>(*u_begin), std::get<1>(*u_begin));
    }

    auto operator==(const v_range_iterator& b) const { return first == b.first; }
    auto operator!=(const v_range_iterator& b) const { return first != b.first; }

    auto operator<(const v_range_iterator& b) const { return first < b.first; }
    auto operator-(const v_range_iterator& b) const { return first - b.first; }
    auto operator+(unsigned long step) const { return v_range_iterator(the_range_, step); }

    auto& operator=(const v_range_iterator& b) {
      the_range_ = b.the_range_;
      first      = b.first;
      last       = b.last;
      u_begin    = b.u_begin;
      u_end      = b.u_end;
      return *this;
    }

    //    typedef std::random_access_iterator_tag            iterator_category;
    typedef std::forward_iterator_tag                  iterator_category;
    typedef size_t                                     difference_type;
    typedef std::tuple<vertex_index_t, vertex_index_t> value_type;
    typedef value_type*                                pointer;
    typedef value_type&                                reference;
  };

  typedef v_range_iterator iterator;

  auto begin() { return v_range_iterator(*this); }
  auto end() { return v_range_iterator(*this, the_graph_.size()); }

private:
  Graph& the_graph_;
};
```

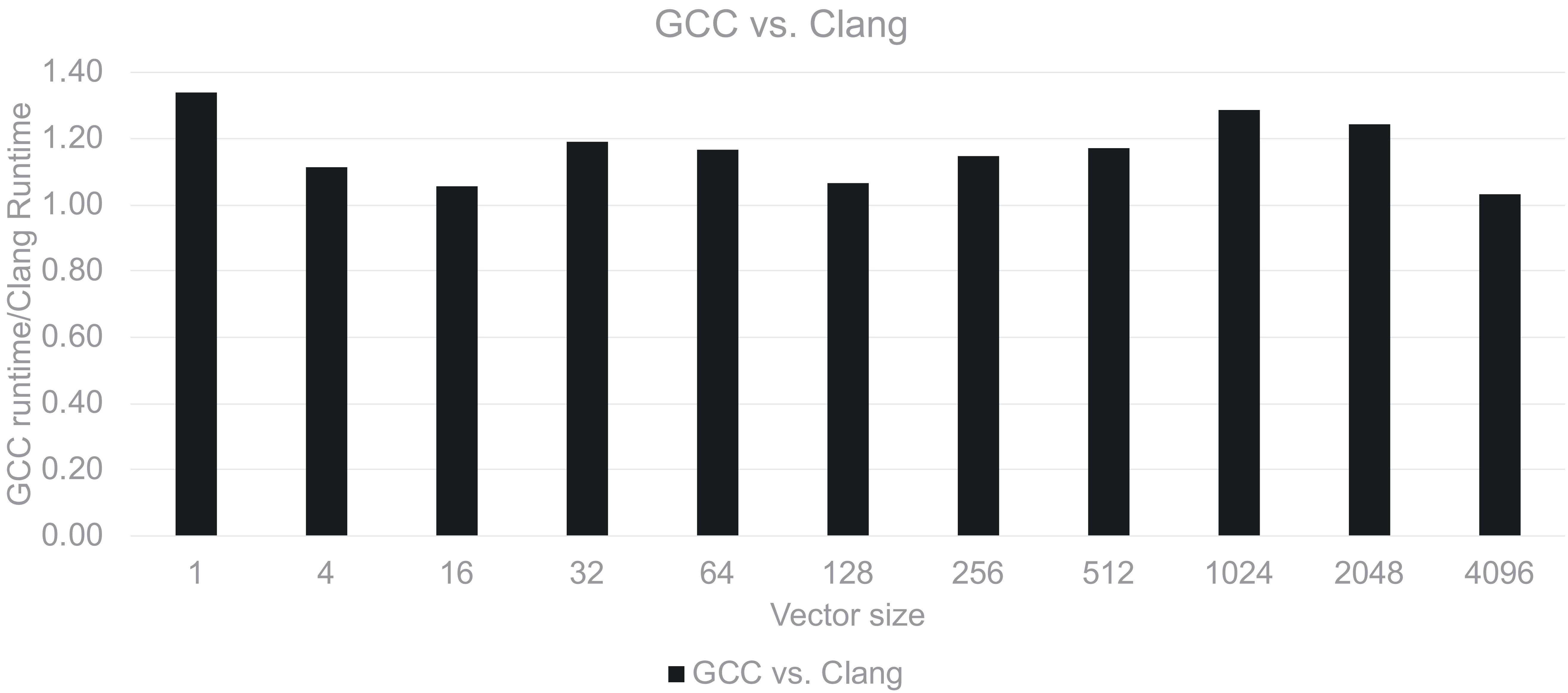
So How Does It Fare?



Compilers Can Do Amazing Things



Clang vs. GCC


GCC vs. Clang
GCC runtime/Clang Runtime
1.40
1.20
1.00
0.80
0.60
0.40
0.20
0.00
1
4
16
32
64
128
256
512
1024
2048
4096
Vector size
GCC vs. Clang

Matvec: Abstraction For The Win

- While there is a small performance loss, the abstraction is very close
- Writing loops by hand is much more error prone
- Saving time in productiveness may be worth a small performance hit
  - Future compilers should do better or at least not worse
  - Future readers of the application code will be grateful
  - The library on the other hand is a bit more complicated
- Optimization takes time: O3 is 3-4x slower to compile than O0, but let's not worry about it
  - Improvements in the language (e.g., modules)
  - Improvements in compiler technology

Another Example: Tensor Template Library (TTL)

- Joint work with Luke Dalessandro and Alexander Winter

```
Tensor<4,3,double> A;
A[1][2][0][1] = 1;
double a[3][3][3][3];
a[1][2][0][1] = 1;
```

```
constexpr const Index<'i'> i;
constexpr const Index<'j'> j;
constexpr const Index<'k'> k;
constexpr const Index<'l'> l;

Tensor<2,2,double> A{}, B{};
Tensor<4,2,double> C;

C(i,j,k,l) = 1.9 * A(i,j) * B(k,l);
```

```
Tensor<2,3,double> A{}, B{};
Tensor<2,3,double> C;

auto AxB = A(i,j) * B(j,k);
auto BxA = B(i,j) * A(j,k);

C(i,k) = 0.5 * (AxB + BxA);

for (int i = 0; i < 3; ++i) {
  for (int k = 0; k < 3; k++) {
    C(i,k) = 0.0;
    for (int j = 0; i < 3; ++j) {
      C(i,k) += 0.5 * (A(i,j) * B(j,k) +
                       B(i,j) * A(j,k));
    }
  }
}
```

Benchmarking TTL

- Inner products

```
cArr[ii]() = A(i) * B(i);
```

```
for(size_t aa = 0; aa < iter; aa++){
  for (size_t ii = 0; ii < D; ii++) {
    arrFL[aa] += arrA[ii] * arrB[ii];
  }
}
```

```
for(size_t aa = 0; aa < iter; aa++){
  for (size_t ii = 0; ii < D; ii++){
    for(size_t jj = 0; jj < D; jj++){
      for(size_t kk = 0; kk < D; kk++){
        for(size_t ll = 0; ll < D; ll++){
          arrFL[aa] += arrA[ii][jj][kk][ll] *
                       arrB[ii][jj][kk][ll];
        }
      }
    }
  }
}
```

```
cArr[ii]() = A(i,j,k,l) * B(i,j,k,l);
```

Benchmarking TTL

- Outer products

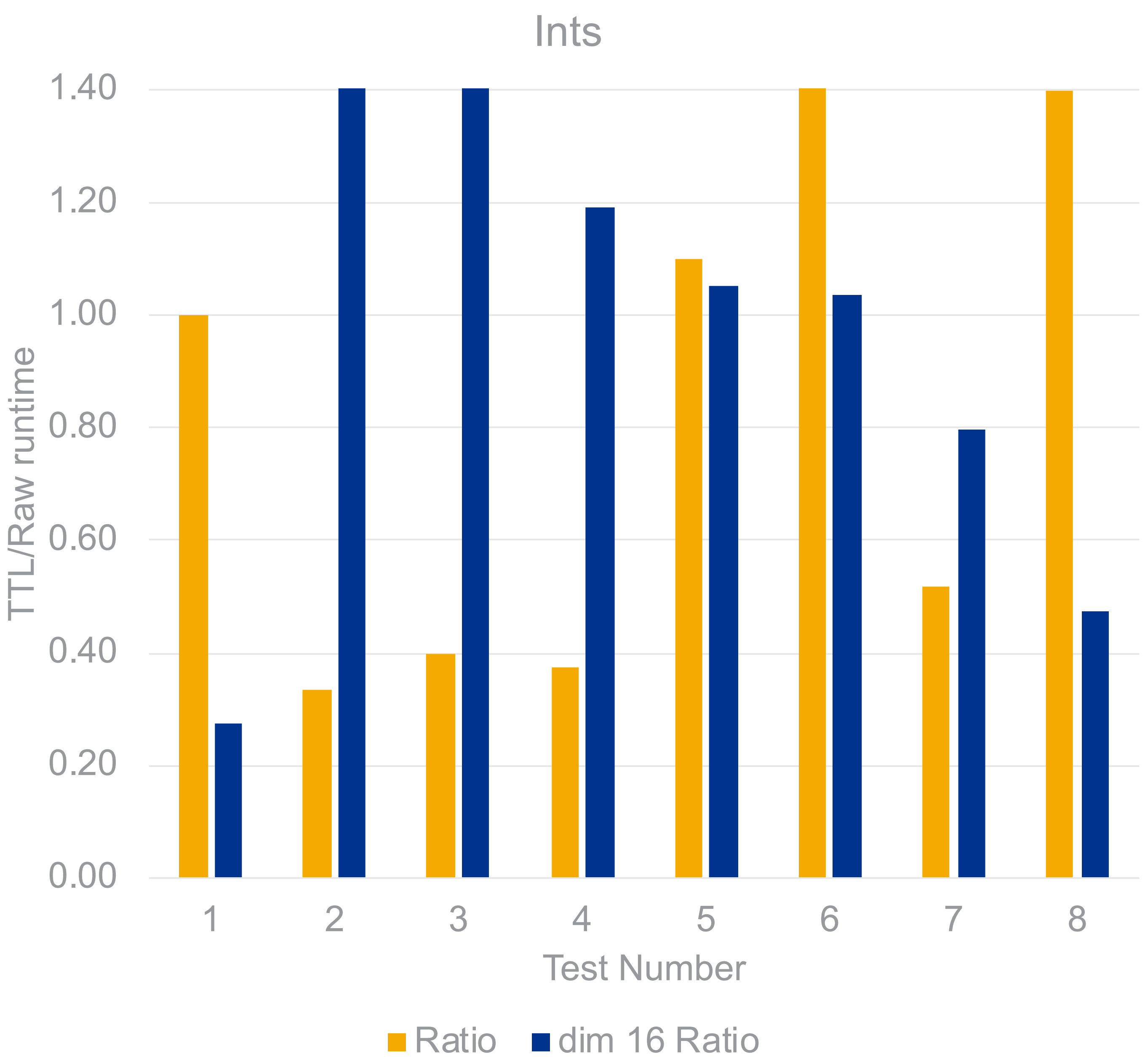
```
cArr[ii](i,j,k,l) = A(i,j) * B(k,l);
```

```
for(size_t aa = 0; aa < iter; aa++){
  for(size_t ii = 0; ii < D; ii++){
    for(size_t jj = 0; jj < D; jj++){
      for(size_t kk = 0; kk < D; kk++){
        for(size_t ll = 0; ll < D; ll++)
          arrFL[aa][ii][jj][kk][ll] =
            arrA[ii][jj] * arrB[kk][ll];
      }
    }
  }
}
```

TTL Benchmark Results

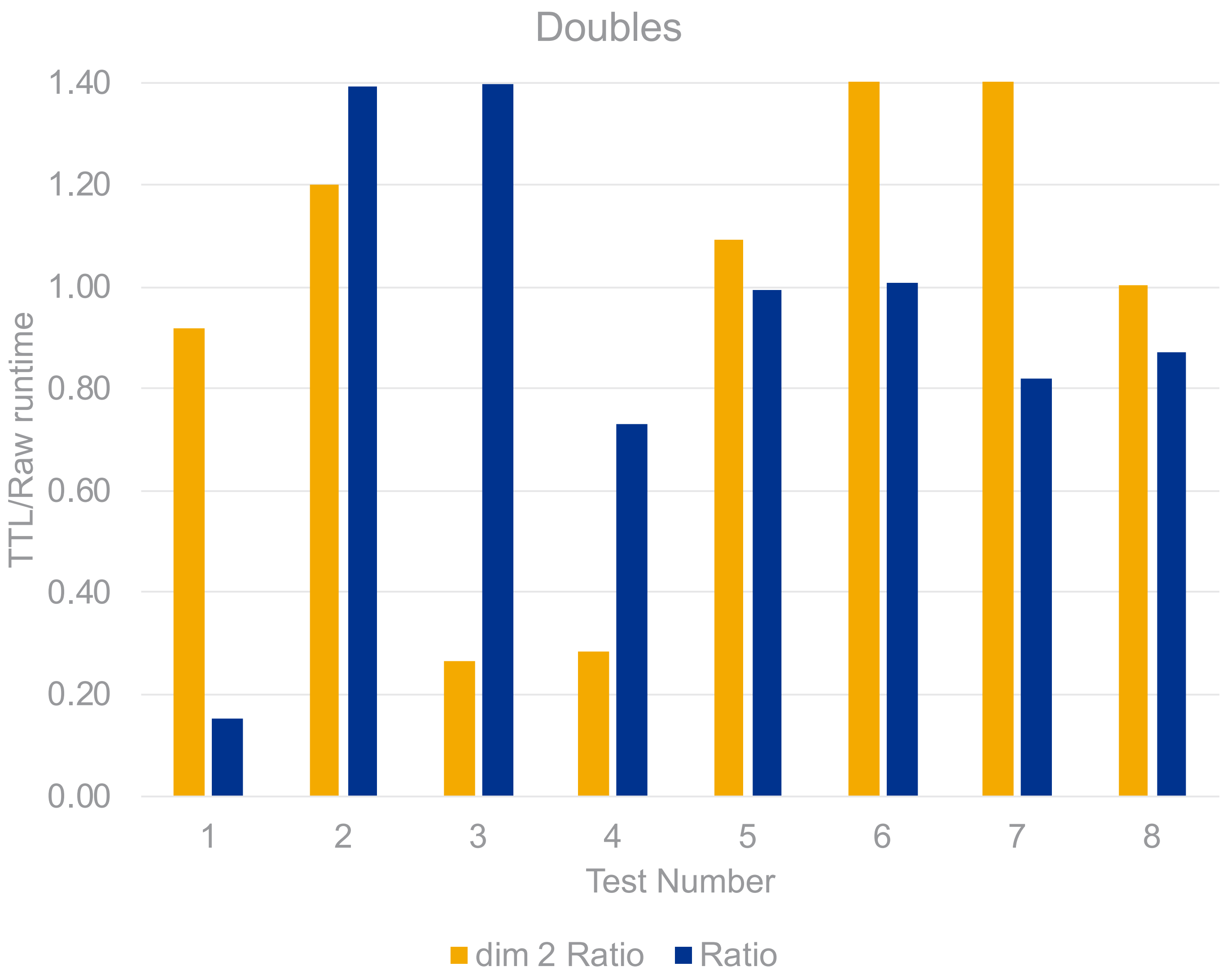
```
Operation          Rank     Format   Runtime (s)
Dimension: 2
Inner Product      1        TTL      6e-06
Inner Product      1        Raw      5e-06
Inner Product      2        TTL      5e-06
Inner Product      2        raw      8e-06
Inner Product      3        TTL      5e-06
Inner Product      3        raw      1e-05
Inner Product      4        TTL      4e-06
Inner Product      4        raw      3.2e-05
Outer Product      1->2     TTL      1.9e-05
Outer Product      1->2     Raw      1.8e-05
Outer Product      1,2->3   TTL      3.9e-05
Outer Product      1,2->3   Raw      3.8e-05
Outer Product      1s->3    TTL      4.1e-05
Outer Product      1s->3    Raw      3.8e-05
Outer Product      2->4     TTL      8.2e-05
Outer Product      2->4     raw      7.5e-05
```

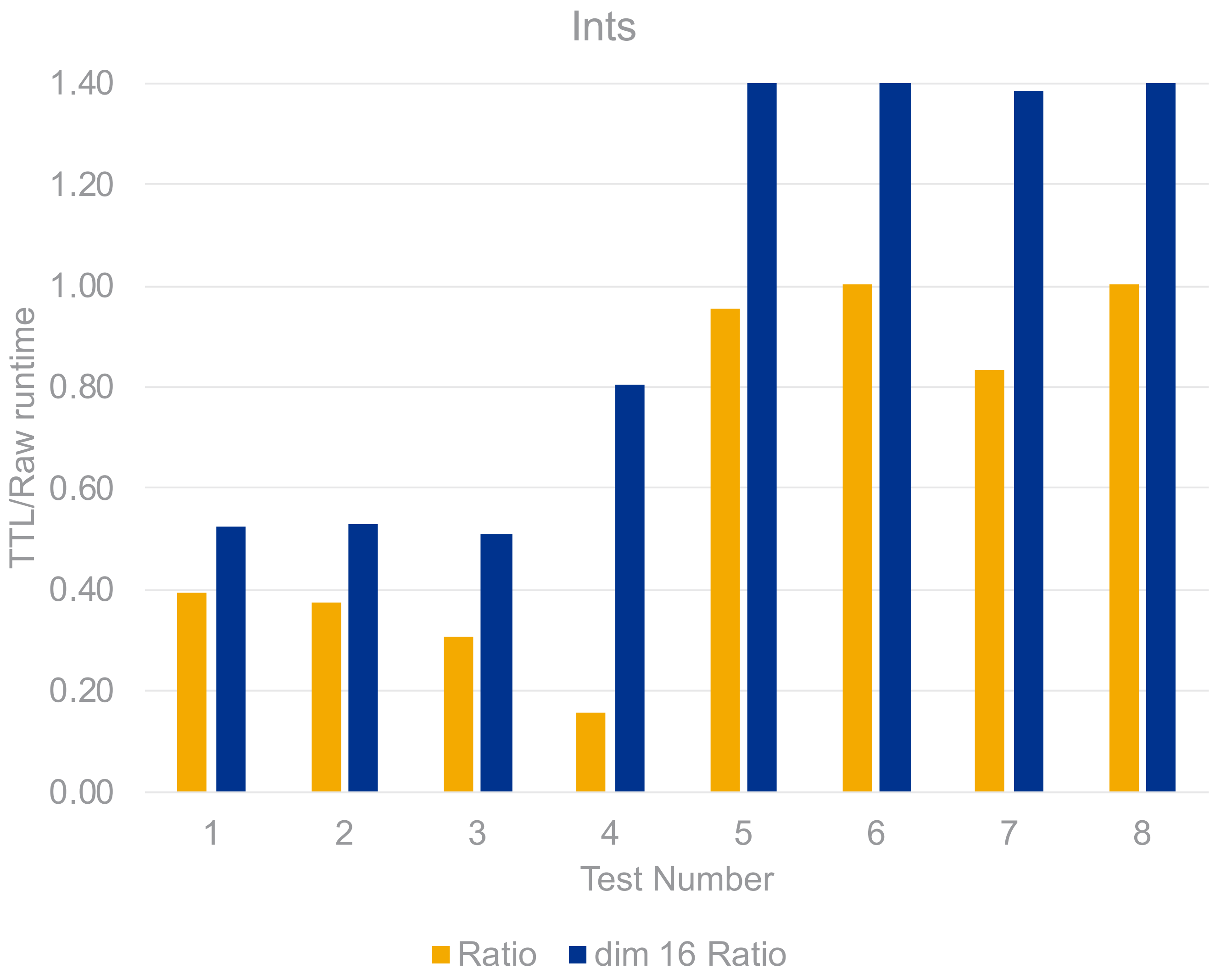
TTL Benchmark Results



GCC

TTL Benchmark Results

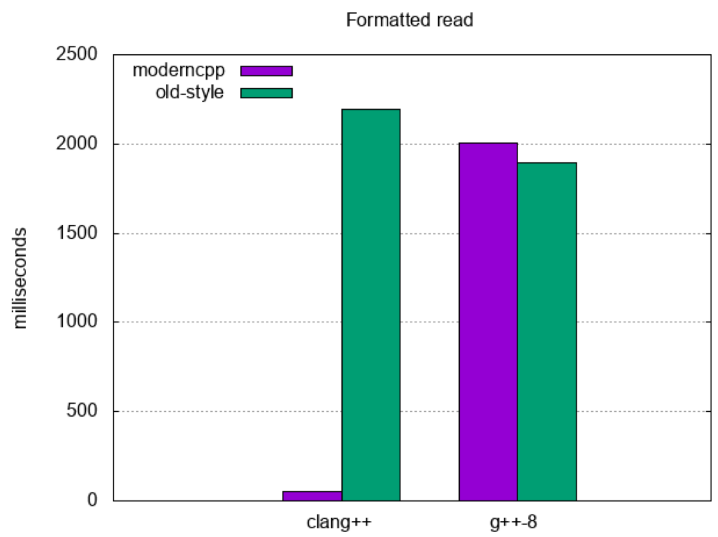


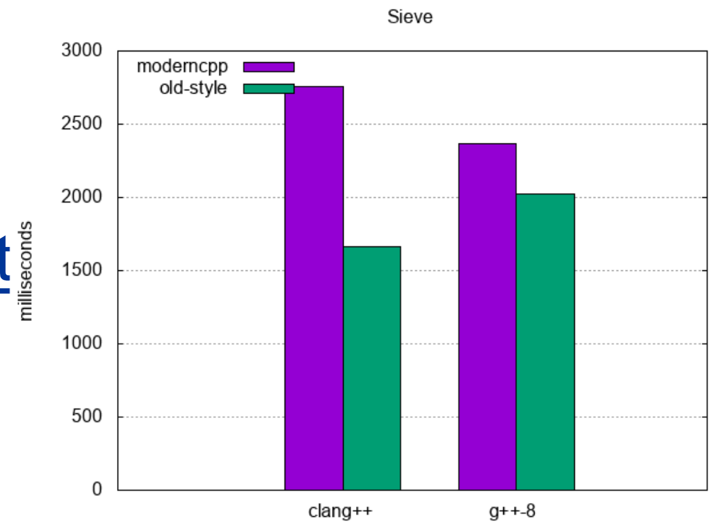


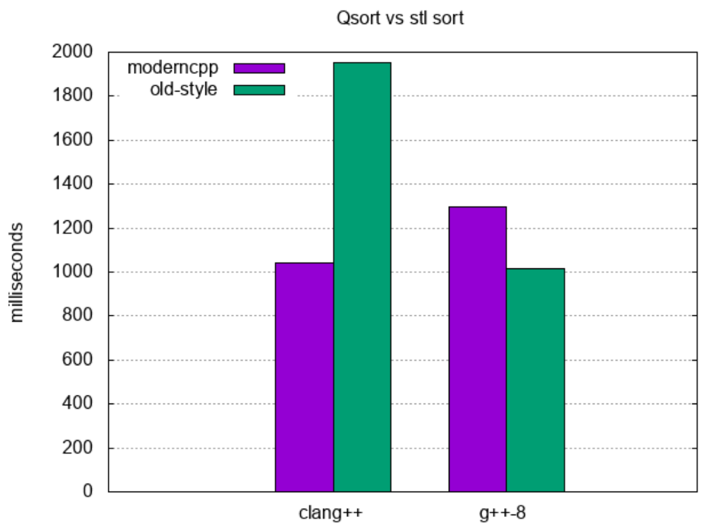
Clang

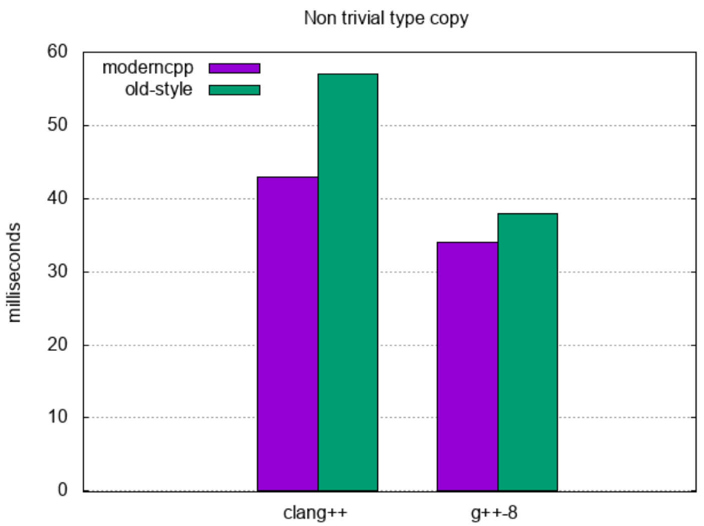
Other Benchmarks

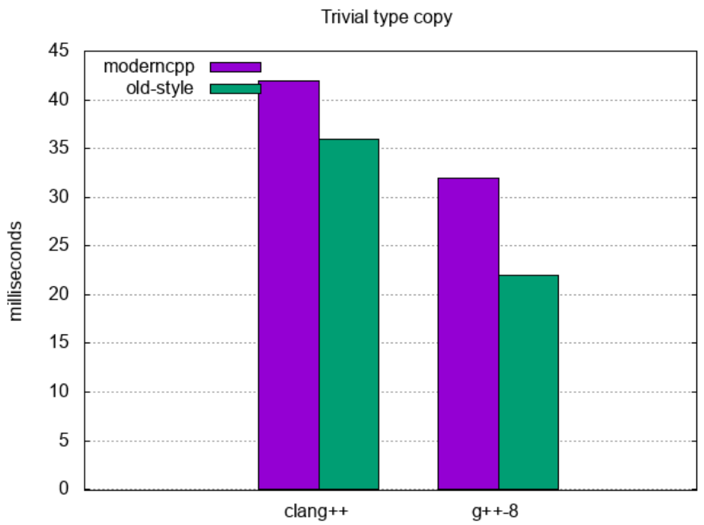
- Chris Cox's CppPerformanceBenchmarks
  - https://gitlab.com/chriscox/CppPerformanceBenchmarks
  - Lots of good tests for low-level language functionality
- German Diago Gomez's the-cpp-abstraction-penalty
  - https://github.com/germandiagogomez/the-cpp-abstraction-penalt
  - Fewer benchmarks, but tests more algorithms









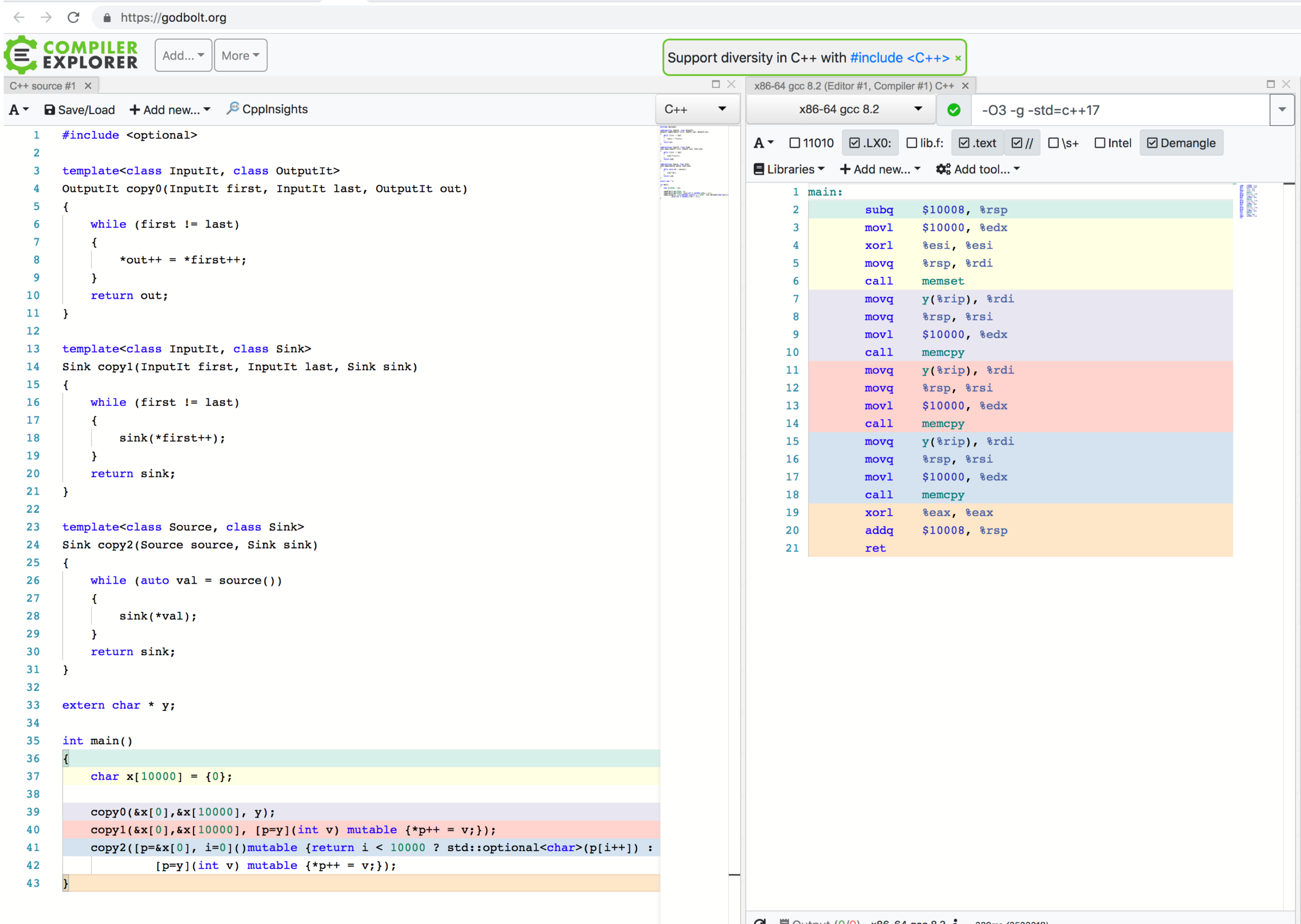


Don’t Give Up: Use Abstraction Unless Proven Wrong

- Abstraction penalty is tough to quantify
- In general, trust the compiler and code the “ideal version”
  - Code readability is worth it
- Identify problems based on need
  - Profile
  - Benchmark abstraction vs. hand written code
  - Unfortunately, maybe look at assembly
  - Maybe the abstraction can be implemented better if it is lagging
- There are very few concentrated efforts to keep current information
- Submit your benchmarks or start your own collection
- Big question: GPU

Thank you

Bonus


https://godbolt.org
COMPILER EXPLORER
Add...
More
Support diversity in C++ with #include <C++>
C++ source #1
Save/Load
Add new...
CppInsights
C++
#include <optional>
template<class InputIt, class OutputIt>
OutputIt copy0(InputIt first, InputIt last, OutputIt out)
{
while (first != last)
{
*out++ = *first++;
}
return out;
}
template<class InputIt, class Sink>
Sink copy1(InputIt first, InputIt last, Sink sink)
{
while (first != last)
{
sink(*first++);
}
return sink;
}
template<class Source, class Sink>
Sink copy2(Source source, Sink sink)
{
while (auto val = source())
{
sink(*val);
}
return sink;
}
extern char * y;
int main()
{
char x[10000] = {0};
copy0(&x[0],&x[10000], y);
copy1(&x[0],&x[10000], [p=y](int v) mutable {*p++ = v;});
copy2([p=&x[0], i=0]()mutable {return i < 10000 ? std::optional<char>(p[i++]) :
[p=y](int v) mutable {*p++ = v;});
}
x86-64 gcc 8.2 (Editor #1, Compiler #1) C++
x86-64 gcc 8.2
-O3 -g -std=c++17
11010
.LX0:
lib.f:
.text
//
\s+
Intel
Demangle
Libraries
Add new...
Add tool...
main:
subq $10008, %rsp
movl $10000, %edx
xorl %esi, %esi
movq %rsp, %rdi
call memset
movq y(%rip), %rdi
movq %rsp, %rsi
movl $10000, %edx
call memcpy
movq y(%rip), %rdi
movq %rsp, %rsi
movl $10000, %edx
call memcpy
movq y(%rip), %rdi
movq %rsp, %rsi
movl $10000, %edx
call memcpy
xorl %eax, %eax
addq $10008, %rsp
ret