Loop Fusion

Andre Meyering, Niklas Korz, David Hector

Jan 28 2020

C++ Practice

Agenda

Development goals

Implementation

Testing

Benchmarks

Usage & Installation

Used Features

Outlook

Development goals

Development goals

```
for (size_t i = 0; i < a.size(); ++i) {
    a[i] = b[i] + c[i];
}
for (size_t i = 0; i < a.size(); ++i) {
    d[i] = a[i] + f[i];
}
```

```
for (size_t i = 0; i < a.size(); ++i) {
    a[i] = b[i] + c[i];
    d[i] = a[i] + f[i];
}
```

- Possible Cache inefficiency
- Hard to read

- Variables in cache may be reused
  $\Rightarrow$ Possibly faster execution

Implementation

How do we fuse loops? - Overview

1. User provides invocables (e.g. lambda functions) and loop boundaries to loopers
2. `looper_union` object is created which contains all functions
3. All overlapping parts of the loop boundaries are put in individual `looper` objects
4. Loopers are executed one after the other, keeping the function sequence

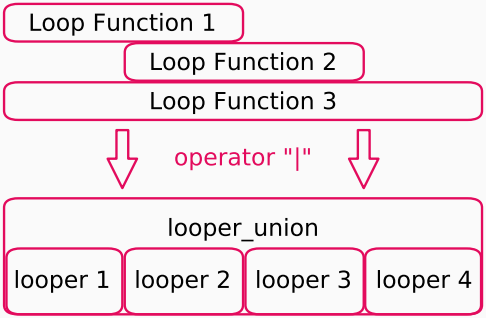


Figure 1: Schematic Loop Fusion

Detailed Implementation - Usage

```
std::array<std::vector<size_t>, 4> vec;
const auto fill_0 = [&vec](size_t) { vec[0].push_back(0); };
const auto fill_1 = [&vec](size_t) { vec[1].push_back(1); };
const auto fill_2 = [&vec](size_t) { vec[2].push_back(2); };
const auto fill_3 = [&vec](size_t) { vec[3].push_back(3); };

auto l1 = loop_to(100) | fill_0;
auto l2 = loop_from_to(10, 90, fill_1) | fill_2;
auto l3 = loop(range{0, 100}) | fill_3
auto merged = l1 | l2 | l3;
merged.run();
```

Detailed Implementation - The `looper` class

General Idea

- `looper` object contains functions with the same loop boundaries
- Functions are stored in `std::tuple`
- Tuple is “expanded” in the for loop

Detailed Implementation - The `looper` class (simplified code)

```
template <typename Iterator, typename... F>
class looper {
public:
    constexpr looper(range<Iterator> _range, std::tuple<F...> _functions) noexcept
        : bounds { _range }
        , functions(_functions) {};

    constexpr void run()
    {
        run_loops(std::index_sequence_for<F...> {}, bounds.start, bounds.end);
    }
    // [...] | operators for merging
public:
    // [...] sanity checks, types, etc.
    common::basic_range<Iterator> bounds;
    std::tuple<F...> functions;
private:
    template <std::size_t... Idx>
    constexpr void run_loops(std::index_sequence<Idx...>, Iterator start, Iterator end)
    {
        for (Iterator i = start; i != end; ++i) {
            (std::get<Idx>(functions)(i), ...);
        }
    }
};
```

Parameter pack stored as std::tuple

Common loop boundaries

Fold expression

3 Implementations

- Compile-Time-Only Merging
- Runtime Merging
- Main Range Merging

Common

- `looper_union` object contains individual ranges and function indexes
- Appending a function with | creates a new type

Compile-Time-Only Implementation

`looper`

- Range is template parameter
- Functions are template parameters

`looper_union`

- Merging in `|` operator
- Class contains functions and ranges with indexes for the used functions in the tuple

Runtime Implementation

`looper`

- Runtime `range` passed as constructor argument
- `run()` method similar to compile time `looper`

`looper_union`

- Merging in `|` operator
- `looper` objects are stored in vector containing `std::variant`
- Running individual `looper` with `std::visit`

`looper_union`

- Works with arbitrary ranges computed at runtime
- Runtime overhead due to iteration over `std::vector`
- But: only when switching between ranges, functions per looper are still inlined
- `std::variant` becomes more complex with each added looper function
  $\Rightarrow 2^N - 1$ types in the `looper` variant
  $\Rightarrow$ All popular C++ compilers implement `std::visit` with constant time
- May result in larger binaries and more expensive calls to `std::visit`

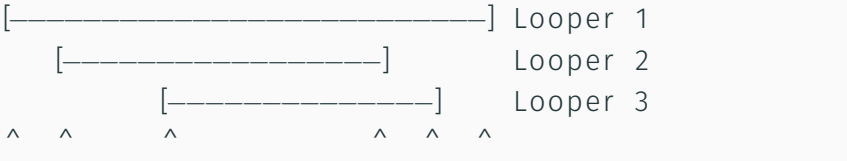
Main Range Implementation

`looper`

- Runtime `range` passed as constructor argument
- `run()` method similar to compile time `looper`

`looper_union`

- Merging in `run()` method
- Only largest common range (intersection) is merged



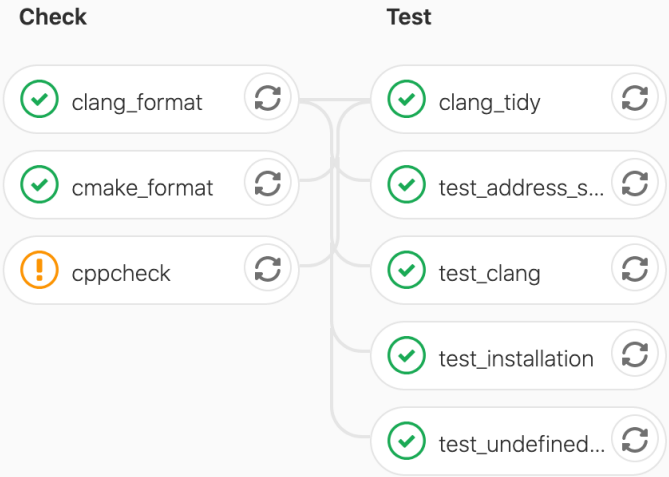
Testing

Testing

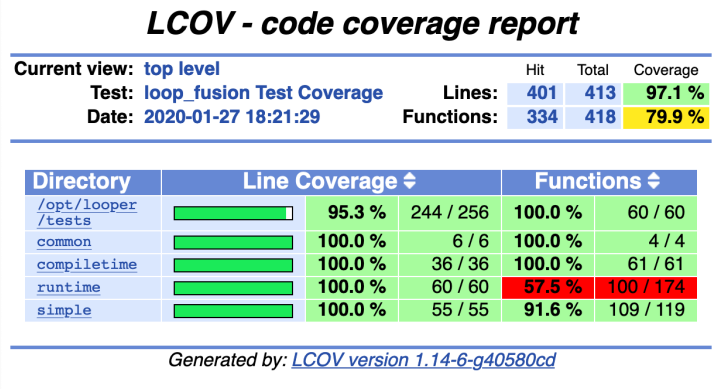
We check every Merge Request against multiple tools using GitLab CI:

- Compile using GCC + Clang
- cppcheck
- Address & Undefined Behaviour Sanitizer
- clang-tidy, clang-format
- Unit Tests using Catch2

GitLab CI


Check
Test
clang_format
cmake_format
cppcheck
clang_tidy
test_address_s...
test_clang
test_installation
test_undefined...

Coverage



LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level				
Test:	loop_fusion Test Coverage	**Lines:**	401	413	97.1 %
Date:	2020-01-27 18:21:29	**Functions:**	334	418	79.9 %

Directory	Line Coverage		Functions	
/opt/looper/tests	95.3 %	244 / 256	100.0 %	60 / 60
common	100.0 %	6 / 6	100.0 %	4 / 4
compiletime	100.0 %	36 / 36	100.0 %	61 / 61
runtime	100.0 %	60 / 60	57.5 %	100 / 174
simple	100.0 %	55 / 55	91.6 %	109 / 119

Generated by: LCOV version 1.14-6-g40580cd

Note: Harder than anticipated

GCC 9.2 does not work with lcov 1.13: Had to use latest master from GitHub!

Down the rabbit hole...

- We even found a compiler bug in GCC 9.2 (which is fixed in trunk)
- Our CI warned us before committing to master
- Clang and MSVC didn't crash!
- Demo: `https://godbolt.org/z/Q64owR`

```
during GIMPLE pass: fre
<source>: In lambda function:
<source>:307:1: internal compiler error: Segmentation fault
  307 | }
      | ^
Please submit a full bug report,
with preprocessed source if appropriate.
See <https://gcc.gnu.org/bugs/> for instructions.
Compiler returned: 1
```

Benchmarks

Benchmarks for same range loop fusion

- Three different loops
- Three equal ranges
- Implementations use very similar looper class for equal range fusion
- Comparison: Handwritten loops vs execution through looper class

```
for (size_t i = 0; i < 1000000; ++i) {
    c[i] = a[i] + b[i];
}

for (size_t i = 0; i < 1000000; ++i) {
    d[i] = c[i] * a[i];
}

for (size_t i = 0; i < 1000000; ++i) {
    c[i] = d[i] - a[i];
}
```

Benchmarks for same range loop fusion

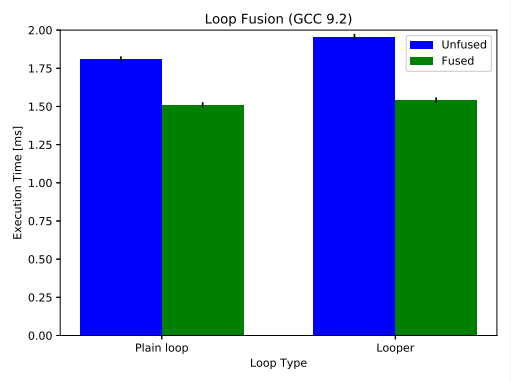


Figure 2: GCC 9.2 (mp-skl2s24c)

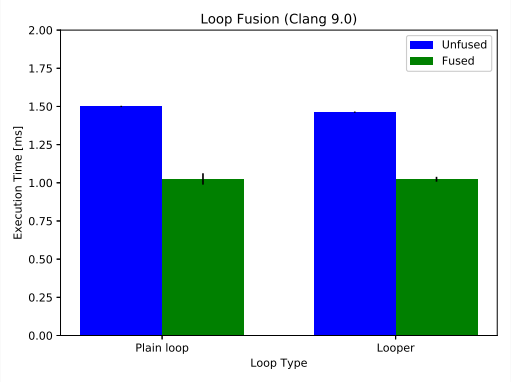


Figure 3: Clang 9.0 (mp-skl2s24c)

Benchmarks for looper unions of different ranges

- Four different loops, four partially overlapping ranges
- Seven ranges after complete merging
- Six ranges after main-range merging (only intersection of all four ranges is added)
- Comparison: Handwritten loops vs execution through united looper classes

```
for (size_t i = 0; i < 1000000; ++i) {
    d[i] = b[i];
    c[i] = a[i] + b[i];
}
for (size_t i = 5000; i < 400000; ++i) {
    d[i] = c[i] * a[i];
}
for (size_t i = 50000; i < 600000; ++i) {
    c[i] = d[i] - a[i];
}
for (size_t i = 300000; i < 900000; ++i) {
    d[i] += a[i];
}
```

Benchmarks for looper unions of different ranges

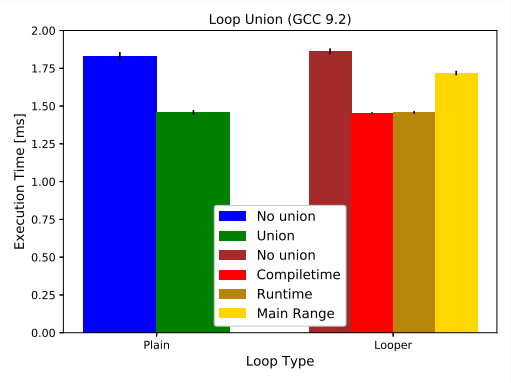


Figure 4: GCC 9.2 (mp-skl2s24c)

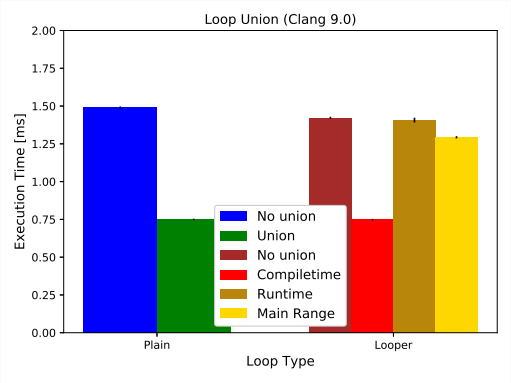


Figure 5: Clang 9.0 (mp-skl2s24c)

Note: Clang does a better job at vectorization and loop-unrolling.

Note: The assembly code for hand-fused and compile-time-fused is the same!

Usage & Installation

Usage & Installation

- We use CMake as our build system generator
- `install` target using GNU standard directories
- `loop_fusionConfig.cmake`, etc. are generated
- `find_package(loop_fusion)` is enough

Godbolt example using our library[1]:

[1] `https://explorer.ameyering.de/z/hc43Tz`

Usage & Installation

```
cmake --build . --target install
```

```
├── include
│   └── loop_fusion
│       ├── basic_looper.hpp
│       ├── ...
└── share
    └── loop_fusion
        └── cmake
            ├── loop_fusionConfig.cmake
            ├── loop_fusionConfigVersion.cmake
            └── loop_fusionTargets.cmake
```

Used Features

Used Features

- Variadic Templates
- Parameter Packs + Fold Expressions
- Partial Template Specialization
- `std::variant` for runtime loop fusion
- `constexpr` for Compile time computation

Outlook

Using Iterators

- Not just integral types
- Also iterators!
- Would work now but issues for merging

```
using iterator = typename std::vector<int>::iterator;
std::vector<int> a(100, 0);
auto l = [&](iterator i) { *i = 10; };
loop(basic_range<iterator>{a.begin(), a.end()}, l).run();
```

More arguments than just the iterator variable

- May be useful for Functors
- Workaround with Lambdas + Captures

```
auto l = [](ArgT1, ArgT2, size_t) { /* [...] */ };
auto arguments = std::tuple<ArgT1, ArgT1>{/* [...] */};
loop(range{0,10}, arguments, func1, func1).run();
```

Thank you for your attention

Backup Slides

C++20

- `std::vector` set to become `constexpr` in C++20 [2]

[2] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1004r2.pdf

Catch2: Custom Generator

RandomRangeGenerator

A `RandomRangeGenerator` with the helper function `random_range(start, end)` allowed easier testing of the mergers.

RandomRangeGenerator

```
const auto range_1 = GENERATE(
    take(50, random_range(0, 1000))
);
```

Doxygen


loop_fusion 0.1.0
loop_fusion
Main Page
Related Pages
Classes
Files
Search
loop_fusion
Loop Fusion
Goal
Compiler Support
Third Party Libraries
CMake Modules
Benchmarks
Code Quality Tools
Code Conventions
Design Decisions
Doxygen Documentation
How to merge loops
Documentation
Terminology
Usage
Classes
Loop Fusion
11 Loop fusion I a.k.a loop_fusion is a project for the course "C++ Practice WS19" from Andre Meyering, Niklas Korz and David Hector.
Goal
We try to create a header-only library that allows users to manually fuse/merge loops.
What is loop fusion?
Assume you have the following code:
for (size_t i = 0; i < a.size(); ++i) {
a[i] = b[i] + c[i];
}
// [...] code that is independent from the loop
Generated by doxygen 1.8.17