GCC and Intel Compiler Optimization on Hot, Vectorizable Function Calls

[redacted]

Abstract

Here we present our findings on the optimization decisions and subsequent runtime performance impact of GCC and Intel Classic Compiler on a easily vectorizable cartesian rotation coded as a scalar looped function call given different function qualifiers and compile-time options.

Based on our findings we propose that inlining, Algebraic Simplification and SIMD/Vectorization to be the primary impacting optimization strategies both compilers have utilized in such context. Additionally, we did not observe significant performance difference between the best performing code the two compilers have yielded.

Contents

Objective

This experiment is designed to observe the difference in compiler decisions and overall runtime performance in hot function optimization across the following three compile time variances:

- How does the general optimization level (`-ON`) option.
- Use of GNU vs Intel Classic compiler.
- How does qualifiers like `static` and `inline` affect compiler decisions in hot function optimization.

Experiment Design

The project codebase is split into three modules, a benchmark target, a benchmark driver and a shell tester.

Benchmark Target

The benchmark target consists of a short status function `rotate()` which applies a rotation on a 2D cartesian coordinate, and a wrapper function `rotateV()` which takes a vector of cartesian coordinates and rotate them by calling `rotate()` in a loop, giving compiler the possibility to perform optimization on it. It exports a structure containing:

- Information about which "variant" this object is as a string constant.
  The "variant" string takes the form of `$COMPILER_O${ONUMBER}_${ROTATE_QUALIFIER// /.}`.
- A function pointer to `rotateV()` so the benchmark driver could find it.

Benchmark Driver

The benchmark driver opens each target as a shared object at run-time and performs the following:

1. Locate the exported structure and record the "variant" of this benchmark case.
2. Generate a pseudorandom sequence of 50000 coordinates.
3. Perform rotation using the benchmark target for 5000 times, recording the time (ns) for each full operation.
4. Calculate the average and stdev of time taken during the middle 4000 repeats.
5. Compute the CRC32 of resulting vector, ensuring each variation is computationally equivalent.
6. Print the result in comma separated format.

The driver is compiled with CMake but it will compile with `g++ -ldl driver.cxx`.

Tester Script

The shell tester script will iterate over all experiment cases, compile the benchmark target, write a function disassembly with GDB then submit a slurm task to run the test driver to load each of the variant and benchmark them.

Results and Observations

After running the tester script and waiting for the job to complete we obtained the following benchmark output:

SPEC	N	REP	AVGTIME	SDTIME	CRC32A	CRC32B
g++_O0_static.inline.void	50000	4000	2.62988e+06	4224.99	86c0a9dc	2f865e2e
g++_O0_static.void	50000	4000	2.6309e+06	80602.2	86c0a9dc	2f865e2e
g++_O0_void	50000	4000	2.63854e+06	7996.5	86c0a9dc	2f865e2e
g++_O1_static.inline.void	50000	4000	53450.2	2045.72	86c0a9dc	2f865e2e
g++_O1_static.void	50000	4000	53430.9	445.855	86c0a9dc	2f865e2e
g++_O1_void	50000	4000	1.54815e+06	980.674	86c0a9dc	2f865e2e
g++_O2_static.inline.void	50000	4000	53615.1	461.254	86c0a9dc	2f865e2e
g++_O2_static.void	50000	4000	53456.4	473.097	86c0a9dc	2f865e2e
g++_O2_void	50000	4000	1.55251e+06	1881.69	86c0a9dc	2f865e2e
g++_O3_static.inline.void	50000	4000	27471.9	355.08	86c0a9dc	2f865e2e
g++_O3_static.void	50000	4000	27360.9	367.615	86c0a9dc	2f865e2e
g++_O3_void	50000	4000	1.55263e+06	1918.48	86c0a9dc	2f865e2e
icpc_O0_static.inline.void	50000	4000	2.64653e+06	12287.1	86c0a9dc	2f865e2e
icpc_O0_static.void	50000	4000	2.64614e+06	2331.95	86c0a9dc	2f865e2e
icpc_O0_void	50000	4000	2.65577e+06	2525.7	86c0a9dc	2f865e2e
icpc_O1_static.inline.void	50000	4000	2.48498e+06	2332.75	86c0a9dc	2f865e2e
icpc_O1_static.void	50000	4000	2.48518e+06	2363.09	86c0a9dc	2f865e2e
icpc_O1_void	50000	4000	2.55634e+06	2809.54	86c0a9dc	2f865e2e

SPEC	N	REP	AVGTIME	SDTIME	CRC32A	CRC32B
icpc_O2_static.inline.void	50000	4000	26742.5	279.696	86c0a9dc	2f865e2e
icpc_O2_static.void	50000	4000	26742.5	279.923	86c0a9dc	2f865e2e
icpc_O2_void	50000	4000	698813	824.445	86c0a9dc	2f865e2e
icpc_O3_static.inline.void	50000	4000	26744.8	285.114	86c0a9dc	2f865e2e
icpc_O3_static.void	50000	4000	26751.8	357.391	86c0a9dc	2f865e2e
icpc_O3_void	50000	4000	698810	826.997	86c0a9dc	2f865e2e

We can make the following trivial observations:

- Every variant returned the same result.
- The fastest and slowest variant on both compilers performed at the same level of magnitude (2e6 to 3e4 ns).
- `static inline` did not result in faster code compared to `static` in any variant.
- On g++ compilers:
  - There is a 2-order speed up between g++ static O0 and static O1.
  - Non static version only received a ~50% speed increase between O0 and O1.
  - There is a small but noticable speed up between static O2 and O3.
- On icpc compilers:
  - O0 and O1 performed about the same across every variant.
  - There is a 2-order speed up between static O1 and O2, but only 75% speed up for non static variant.
  - O3 did not result in any noticable speed up compared to O2. This is expected as the official documentation suggest that `O3` deals with complex operations and nested loops which are not present in this simple example.

Analysis

By analyzing the assembler code for the compiled targets we propose the following three mechanisms to be the main contributing factors to the speed-ups observed above.

Inlining

Inlining resulted in the most significant performance increase at about 20X. By comparing `g++` O1 static vs non static variant we could see the non static variant calls `rotate()` every iteration while in the static variant the function was inlined.

We did not expect the difference to be so significant, maybe using a profiler can shed some light on the reason behind this.

```
for s in g++_O1_{,static.}void.so; gdb -batch -ex "disas rotateV" obj/$s ; end
Dump of assembler code for function rotateV(size_t, double*, double*, double):
   0x00000000000011a2 <+0>: test   rdi,rdi
   0x00000000000011a5 <+3>: je     0x11e6 <rotateV(size_t, double*, double*,
 ↪  double)+68>
   0x00000000000011a7 <+5>: push   r12
   0x00000000000011a9 <+7>: push   rbp
   0x00000000000011aa <+8>: push   rbx
   0x00000000000011ab <+9>: sub    rsp,0x10
   0x00000000000011af <+13>:    movsd  QWORD PTR [rsp+0x8],xmm0
   0x00000000000011b5 <+19>:    mov    rbx,rsi
   0x00000000000011b8 <+22>:    mov    rbp,rdx
   0x00000000000011bb <+25>:    lea    r12,[rsi+rdi*8]
   0x00000000000011bf <+29>:    movsd  xmm0,QWORD PTR [rsp+0x8]
   0x00000000000011c5 <+35>:    mov    rsi,rbp
```

```
   0x00000000000011c8 <+38>:    mov    rdi,rbx
   0x00000000000011cb <+41>:    call   0x1030 <_Z6rotateRdS_d@plt>
   0x00000000000011d0 <+46>:    add    rbx,0x8
   0x00000000000011d4 <+50>:    add    rbp,0x8
   0x00000000000011d8 <+54>:    cmp    rbx,r12
...
Dump of assembler code for function rotateV(size_t, double*, double*, double):
   0x0000000000001135 <+0>: test   rdi,rdi
   0x0000000000001138 <+3>: je     0x11b0 <rotateV(size_t, double*, double*,
   ↪  double)+123>
   0x000000000000113a <+5>: push   r12
   0x000000000000113c <+7>: push   rbp
   0x000000000000113d <+8>: push   rbx
   0x000000000000113e <+9>: sub    rsp,0x10
   0x0000000000001142 <+13>:    mov    rbp,rdx
   0x0000000000001145 <+16>:    mov    rbx,rsi
   0x0000000000001148 <+19>:    mov    r12,rdi
   0x000000000000114b <+22>:    lea    rdi,[rsp+0x8]
   0x0000000000001150 <+27>:    mov    rsi,rsp
   0x0000000000001153 <+30>:    call   0x1050 <sincos@plt>
   0x0000000000001158 <+35>:    movsd  xmm4,QWORD PTR [rsp]
   0x000000000000115d <+40>:    movsd  xmm3,QWORD PTR [rsp+0x8]
   0x0000000000001163 <+46>:    mov    eax,0x0
   0x0000000000001168 <+51>:    movsd  xmm0,QWORD PTR [rbx+rax*8]
   0x000000000000116d <+56>:    movsd  xmm1,QWORD PTR [rbp+rax*8+0x0]
   0x0000000000001173 <+62>:    movapd xmm2,xmm0
   0x0000000000001177 <+66>:    mulsd  xmm2,xmm4
...
```

Algebraic Simplification

Since the `rotate()` function repeatedly use `sin(alpha)` and `cos(alpha)`. The compiler can use `sincos()` to calculate both values in one go and reuse them cross the whole function. The following example `icpc` was able to reduce 4 calls to library trigonometry functions to a single SSE2 `sincos()` across O1 and O2 non static variant. This is consistent with the observed ~4X speed increase between the two variants during benchmark.

```
$ for s in icpc_O{1,2}_void.so; gdb -batch -ex "disas _Z6rotateRdS_d" obj/$s ; end
Dump of assembler code for function rotate(double&, double&, double):
...
   0x000000000000223f <+38>:    call   0x2050 <cos@plt>
   0x0000000000002244 <+43>:    movsd  QWORD PTR [rsp+0x18],xmm0
   0x000000000000224a <+49>:    movsd  xmm0,QWORD PTR [rsp]
   0x000000000000224f <+54>:    call   0x2030 <sin@plt>
   0x0000000000002254 <+59>:    movsd  xmm1,QWORD PTR [rsp+0x18]
   0x000000000000225a <+65>:    mulsd  xmm1,QWORD PTR [rsp+0x10]
   0x0000000000002260 <+71>:    mulsd  xmm0,QWORD PTR [rsp+0x8]
   0x0000000000002266 <+77>:    subsd  xmm1,xmm0
   0x000000000000226a <+81>:    movsd  xmm0,QWORD PTR [rsp]
   0x000000000000226f <+86>:    movsd  QWORD PTR [rbx],xmm1
   0x0000000000002273 <+90>:    call   0x2030 <sin@plt>
   0x0000000000002278 <+95>:    movsd  QWORD PTR [rsp+0x18],xmm0
   0x000000000000227e <+101>:   movsd  xmm0,QWORD PTR [rsp]
   0x0000000000002283 <+106>:   call   0x2050 <cos@plt>
   0x0000000000002288 <+111>:   movsd  xmm2,QWORD PTR [rsp+0x10]
   0x000000000000228e <+117>:   movsd  xmm1,QWORD PTR [rsp+0x8]
```

```
   0x0000000000002294 <+123>:   mulsd  xmm2,QWORD PTR [rsp+0x18]
   0x000000000000229a <+129>:   mulsd  xmm1,xmm0
   0x000000000000229e <+133>:   addsd  xmm2,xmm1
   0x00000000000022a2 <+137>:   movsd  QWORD PTR [rbp+0x0],xmm2

Dump of assembler code for function rotate(double&, double&, double):
...
   0x000000000000220e <+14>:    movsd  xmm2,QWORD PTR [r15]
   0x0000000000002213 <+19>:    movsd  xmm1,QWORD PTR [r14]
   0x0000000000002218 <+24>:    movsd  QWORD PTR [rsp+0x8],xmm1
   0x000000000000221e <+30>:    movsd  QWORD PTR [rsp],xmm2
   0x0000000000002223 <+35>:    call   0x2080 <__libm_sse2_sincos@plt>
   0x0000000000002228 <+40>:    movsd  xmm5,QWORD PTR [rsp+0x8]
   0x000000000000222e <+46>:    movaps xmm2,xmm0
   0x0000000000002231 <+49>:    movsd  xmm3,QWORD PTR [rsp]
   0x0000000000002236 <+54>:    movaps xmm4,xmm1
   0x0000000000002239 <+57>:    movaps xmm0,xmm5
   0x000000000000223c <+60>:    movaps xmm1,xmm3
   0x000000000000223f <+63>:    mulsd  xmm0,xmm4
   0x0000000000002243 <+67>:    mulsd  xmm1,xmm2
   0x0000000000002247 <+71>:    mulsd  xmm5,xmm2
   0x000000000000224b <+75>:    mulsd  xmm4,xmm3
   0x000000000000224f <+79>:    subsd  xmm0,xmm1
   0x0000000000002253 <+83>:    addsd  xmm5,xmm4
   0x0000000000002257 <+87>:    movsd  QWORD PTR [r14],xmm0
   0x000000000000225c <+92>:    movsd  QWORD PTR [r15],xmm5
```

SIMD/Vectorization

icpc moved to vectorization at O2 and g++ moved to vectorization at O3.

In the following example between **g++** static O2 and O3 variant we can observe scalar operations were replaced with SSE2 packed operations. Since xmm registers fits two doubles at a time that corroborates with the observed ~50% bump in speed. Since non static version function calls were not inlined it was not possible to vectorize, thus there is no observable speed up between **g++** non static O2 and O3 variants.

```
for s in g++_O{2,3}_static.void.so; gdb -batch -ex "disas rotateV" obj/$s ; end
```

```
Dump of assembler code for function rotate(double&, double&, double):
...
   0x00000000000011f2 <+18>:    sub    rsp,0x10
   0x00000000000011f6 <+22>:    lea    rdi,[rsp+0x8]
   0x00000000000011fb <+27>:    mov    rsi,rsp
   0x00000000000011fe <+30>:    call   0x1060 <sincos@plt>
   0x0000000000001203 <+35>:    movsd  xmm4,QWORD PTR [rsp]
   0x0000000000001208 <+40>:    movsd  xmm3,QWORD PTR [rsp+0x8]
   0x000000000000120e <+46>:    xor    eax,eax
   0x0000000000001210 <+48>:    movsd  xmm0,QWORD PTR [rbx+rax*8]
   0x0000000000001215 <+53>:    movsd  xmm1,QWORD PTR [rbp+rax*8+0x0]
   0x000000000000121b <+59>:    movapd xmm2,xmm0
   0x000000000000121f <+63>:    movapd xmm5,xmm1
   0x0000000000001223 <+67>:    mulsd  xmm2,xmm4
   0x0000000000001227 <+71>:    mulsd  xmm5,xmm3
   0x000000000000122b <+75>:    mulsd  xmm0,xmm3
   0x000000000000122f <+79>:    mulsd  xmm1,xmm4
   0x0000000000001233 <+83>:    subsd  xmm2,xmm5
```

```
   0x0000000000001237 <+87>:    addsd  xmm0,xmm1
   0x000000000000123b <+91>:    movsd  QWORD PTR [rbx+rax*8],xmm2
   0x0000000000001240 <+96>:    movsd  QWORD PTR [rbp+rax*8+0x0],xmm0

Dump of assembler code for function rotateV(size_t, double*, double*, double):
...
0x0000000000001202 <+34>:   call   0x1060 <sincos@plt>
   0x0000000000001207 <+39>:    lea    rax,[rbp+0x10]
   0x000000000000120b <+43>:    movsd  xmm4,QWORD PTR [rsp]
   0x0000000000001210 <+48>:    movsd  xmm5,QWORD PTR [rsp+0x8]
   0x0000000000001216 <+54>:    cmp    rbx,rax
   0x0000000000001219 <+57>:    lea    rax,[rbx+0x10]
   0x000000000000121d <+61>:    setae  dl
   0x0000000000001220 <+64>:    cmp    rbp,rax
   0x0000000000001223 <+67>:    setae  al
   0x0000000000001226 <+70>:    or     dl,al
   0x0000000000001228 <+72>:    je     0x12f0 <rotateV(size_t, double*, double*,
   ↪  double)+272>
   0x000000000000122e <+78>:    lea    rax,[r12-0x1]
   0x0000000000001233 <+83>:    cmp    rax,0x1
   0x0000000000001237 <+87>:    jbe    0x12f0 <rotateV(size_t, double*, double*,
   ↪  double)+272>
   0x000000000000123d <+93>:    mov    rdx,r12
   0x0000000000001240 <+96>:    movapd xmm6,xmm4
   0x0000000000001244 <+100>:   movapd xmm0,xmm5
   0x0000000000001248 <+104>:   xor    eax,eax
   0x000000000000124a <+106>:   shr    rdx,1
   0x000000000000124d <+109>:   unpcklpd xmm6,xmm6
   0x0000000000001251 <+113>:   unpcklpd xmm0,xmm0
   0x0000000000001255 <+117>:   shl    rdx,0x4
   0x0000000000001259 <+121>:   nop    DWORD PTR [rax+0x0]
   0x0000000000001260 <+128>:   movupd xmm1,XMMWORD PTR [rbp+rax*1+0x0]
   0x0000000000001266 <+134>:   movupd xmm2,XMMWORD PTR [rbx+rax*1]
   0x000000000000126b <+139>:   movapd xmm3,xmm1
   0x000000000000126f <+143>:   movapd xmm7,xmm2
   0x0000000000001273 <+147>:   mulpd  xmm3,xmm6
   0x0000000000001277 <+151>:   mulpd  xmm7,xmm0
   0x000000000000127b <+155>:   mulpd  xmm1,xmm0
   0x000000000000127f <+159>:   mulpd  xmm2,xmm6
   0x0000000000001283 <+163>:   subpd  xmm3,xmm7
   0x0000000000001287 <+167>:   addpd  xmm1,xmm2
...
```

Discussions

Here we present how GCC and Intel compiler chain can optimize hot and vectorizable function calls. Apart from the three variables discussed there are also numerous factors that may play a role in producing optimal code, such as:

1. We used the compiler default architecture setting, which prevented them from using the full capability of the machine architecture available. `-march=native` may enable further vectorization with more recent SIMD instruction sets.
2. We designed the experiment so that the compiler can not deduce any information about how the function would be used. In reality the compiler may be able to optimize out constants or perform better branch prediction based on how the target function was used in the code context.
3. Optimizations that potentially result in functionally non-equivalent code such as `fast-math` have

not been experimented.

Appendix

Benchmark Target

Source of `rotate/rotate.cxx`:

```
#include "./rotate.hxx"
#include <cmath>

using std::vector;

#ifndef ROTATE_SPEC
#define ROTATE_SPEC "UNKNOWN"
#endif

#ifndef ROTATE_QUALIFIERS
#define ROTATE_QUALIFIERS void
#endif

ROTATE_QUALIFIERS rotate(double &x, double &y, const double alpha)
{
    double x0 = x, y0 = y;
    x = cos(alpha) * x0 - sin(alpha) * y0;
    y = sin(alpha) * x0 + cos(alpha) * y0;
    return;
}

extern "C" void rotateV(size_t n, double *a, double *b, double alpha)
{
    for (size_t i = 0; i < n; i++)
        rotate(a[i], b[i], alpha);
}

extern "C"
{
    rotate_spec rotate_export = rotate_spec{
        ROTATE_SPEC,
        rotateV};
};
```

Benchmark Driver

Source of `driver.cxx`:

```
#include "rotate/rotate.hxx"
#include "./sd.hxx"
#include "./crc32.hxx"
#include <chrono>
#include <iostream>
#include <dlfcn.h>
#include <random>

using std::cerr;
using std::cout;
```

```cpp
using std::endl;
using std::vector;

typedef struct
{
    std::string spec;
    long n = 50000;
    long repeat = 5000;

    double avgTime;
    double sdTime;

    std::pair<uint32_t, uint32_t> crc32_result;
} rotate_b_t;

std::ostream &operator<<(std::ostream &out, const rotate_b_t &rotate)
{
    return out << rotate.spec << "," << rotate.n << "," << rotate.repeat << ","
               << rotate.avgTime << "," << rotate.sdTime << ","
               << std::hex << rotate.crc32_result.first << "," << rotate.crc32_result.second << std::dec << endl;
}

void benchmarkRotate(rotate_b_t &arg, RotateV rotateV)
{
    std::mt19937 rng;
    std::uniform_real_distribution<double> rand_angle(0, 3.14);

    vector<unsigned long> times(arg.repeat);
    vector<double> a(arg.n);
    vector<double> b(arg.n);
    for (auto &n : a)
        n = rand_angle(rng);
    for (auto &n : b)
        n = rand_angle(rng);

    for (auto r = arg.repeat - 1; r >= 0; r--)
    {
        auto start = std::chrono::steady_clock::now();
        rotateV(arg.n, a.data(), b.data(), 0.1);
        auto end = std::chrono::steady_clock::now();
        times[r] = std::chrono::duration_cast<std::chrono::nanoseconds>(end - start).count();
    }

    // don't count the first and last 10% for stability
    for (long i = 0; i < arg.repeat / 10; i++)
    {
        times[i] = 0;
        times[arg.repeat - i - 1] = 0;
    }
    arg.repeat -= arg.repeat / 10 * 2;

    auto t = meanAndSd<unsigned long, double>(times);
    arg.avgTime = t.first;
    arg.sdTime = t.second;
```

```
    arg.crc32_result = std::make_pair(crc32a((uint8_t *)a.data(), arg.n * sizeof(double)),
                                      crc32a((uint8_t *)b.data(), arg.n * sizeof(double)));
}

int main(int argc, char *argv[])
{
    cout << "SPEC,N,REP,AVGTIME,SDTIME,CRC32A,CRC32B" << endl;
    for (int i = 1; i < argc; i++)
    {
        void *rotateH = dlopen(argv[i], RTLD_LAZY);
        if (!rotateH)
        {
            cerr << "Cannot load library" << argv[i] << ": " << dlerror() << endl;
            continue;
        }
        rotate_spec *rspec = (rotate_spec *)dlsym(rotateH, "rotate_export");
        const char *dlsym_error = dlerror();
        if (dlsym_error)
        {
            cerr << "Cannot load symbol : " << dlsym_error << endl;
            dlclose(rotateH);
            return 1;
        }

        // cerr << "Testing with: " << rspec->spec << endl;
        rotate_b_t arg;
        arg.spec = rspec->spec;
        benchmarkRotate(arg, rspec->rotateV);
        cout << arg;
        dlclose(rotateH);
    }
    return 0;
}
```

Tester Script

Source of `rotate/flow.fish`:

```
#!/usr/bin/env fish

module load intel

icpc --version
or exit 5

g++ --version
or exit 5

rm -f obj/*.{so, disas}

for i in 0 1 2 3
    for c in icpc g++
        for qualifier in "static void" "static inline void" "void"
                set fish_trace 1
```

```
                set -l outputBase "$c"_O"$i"_(string replace -a " " "." $qualifier)


                $c  -ggdb -shared -fPIC \
                    -DROTATE_SPEC=\"$outputBase\" \
                    -DROTATE_QUALIFIERS=$qualifier \
                    -o obj/$outputBase.so -O"$i" rotate.cxx
                or exit 5

                gdb -ex "set disassembly-flavor intel" \
                    -ex "disas /m rotateV" \
                    -batch obj/$outputBase.so &> obj/$outputBase.disas

                set -e fish_trace
        end
    end
end

srun -pdevelopment -N1 -n1 -t00:15:00 \
     ../../../out/bin/rotate_bench obj/*.so &| tee rotate_bench.out
```