Mandelbrot Set Image Generation

Maths, Data, and Machines

Gareth Lloyd

ACCU York

2nd July 2025

Outline

1. Introduction
2. Generating the Image
3. Optimization Journey
4. Conclusion

What is the Mandelbrot Set?

- A famous mathematical set of complex numbers
- Boundary forms a fractal pattern
- High-quality visualization made by Benoit Mandelbrot in 1980
- First defined and drawn by Robert W. Brooks and Peter Matelski in 1978
- Generated by iterating a simple mathematical formula

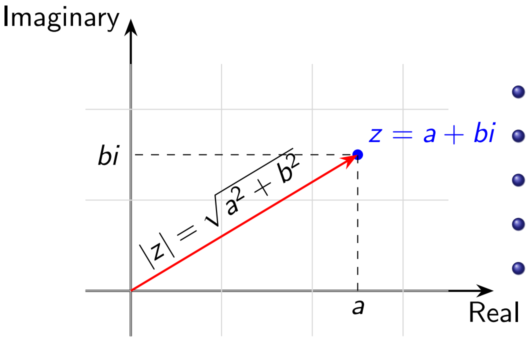
The Mathematical Foundation

The Mandelbrot Formula

$$z_{n+1} = z_n^2 + c$$

- Starting with $z_0 = 0$
- c is a point in the complex plane
- If the sequence remains bounded, the point is in the set
- A sequence diverges to infinity if $|z_n| > 2$, hence iteration can be stopped

Complex Numbers



- Complex number $z = a + bi$
- a: real part (x-axis)
- b: imaginary part (y-axis)
- $|z| = \sqrt{a^2 + b^2}$: magnitude
- $i^2 = -1$

Algorithm Overview

1. Choose a region in the complex plane
2. For each pixel:
   - Map to complex number c
   - Iterate the formula
   - Color based on escape time (number of iterations)

Mandelbrot Set Visualization

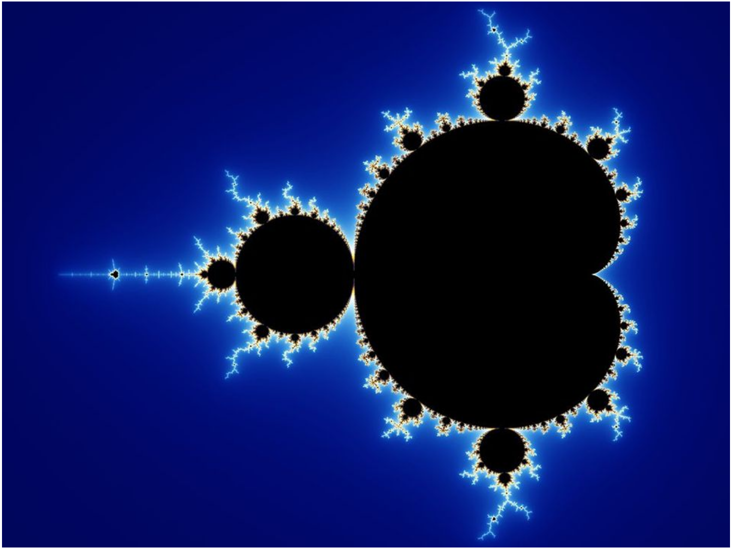


Image by Wolfgang Beyer (CC BY-SA 3.0)

Naïve

Using std::complex

- Clean, readable code using standard library
- `std::complex` handles complex arithmetic
- Easy to understand and maintain

Implementation

```
auto mandelbrot(std::complex<double> c) -> std::size_t {
  auto iter = std::size_t{};
  auto z = std::complex<double>{};

  while (std::abs(z) <= 2.0 and iter < MAX_ITER) {
    z = z * z + c;
    ++iter;
  }
  return iter;
}
```

Engineering Mindset

Evidence-Driven Development

- Avoid premature optimization based on assumptions
- Let evidence guide your efforts
- Similar to TDD: Know when something is wrong because the tests fail

Performance Optimization Cycle

1. Profile and measure current performance
2. Identify actual bottlenecks (not assumed ones)
3. Make targeted improvements
4. Verify with measurements
5. Repeat

Benchmarking Setup

Google Benchmark Code

```cpp
static void BM_Mandelbrot_V1(benchmark::State &state) {
  for (auto _ : state) {
    auto result = mandelbrot<MAX_ITER>(std::complex{0.0, 0.0});
    benchmark::DoNotOptimize(result);
  }
  state.counters["calc"] = benchmark::Counter(1, benchmark::Counter::kIsIterationInvariantRate);
}
BENCHMARK(BM_Mandelbrot_V1);
```

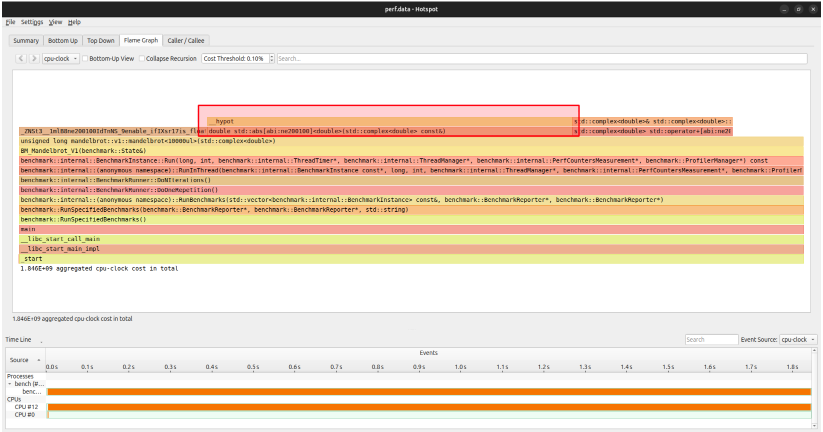
Profiling Tools

- Google Benchmark for microbenchmarking
- Linux `perf` for performance analysis
- `Hotspot` as a convenient GUI for `perf`

Why $0.0 + 0.0i$?

- This is the worst case, it is in the set and can not escape
- Are there other cases we should profile?

Performance Analysis



- Profiling reveals significant time spent in `std::abs()`

The Problem with `std::abs`

Performance Issues

- `std::abs` for `std::complex` calls `__hypot`
- `__hypot` is a non-inlinable C library call
- `__hypot` internally performs an expensive square root operation

Mathematical Insight

$$
\begin{aligned}
\mathrm{hypot}(z) \leq 2 &\Leftrightarrow \sqrt{a^2 + b^2} \leq 2 \\
&\Leftrightarrow a^2 + b^2 \leq 4 \\
&\Leftrightarrow \mathrm{norm}(z) \leq 4
\end{aligned}
$$

Optimized

Using std::norm

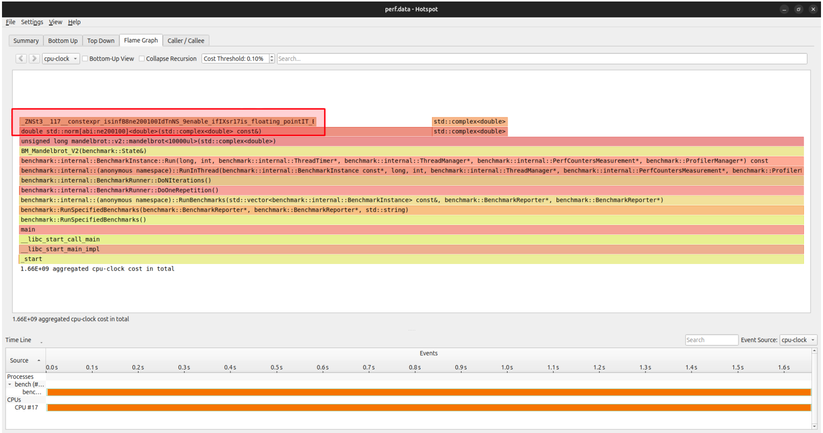
- Replaces `std::abs(z) <= 2` with `std::norm(z) <= 4`
- Avoids expensive square root operation, maintains mathematical equivalence

Implementation

```
auto mandelbrot(std::complex<double> c) -> std::size_t {
  auto iter = std::size_t{};
  auto z = std::complex<double>{};

  while (std::norm(z) <= 4.0 and iter < MAX_ITER) {
    z = z * z + c;
    ++iter;
  }
  return iter;
}
```

Performance Analysis



- Small amount of work to handle infinities
- Our values stay small we don’t need to handle infinities

Manual

Avoiding `std::norm`

- Directly compute $x^2 + y^2$ instead of using `std::norm`
- No need to handle special cases (infinities, NaNs)
- More explicit control over the computation

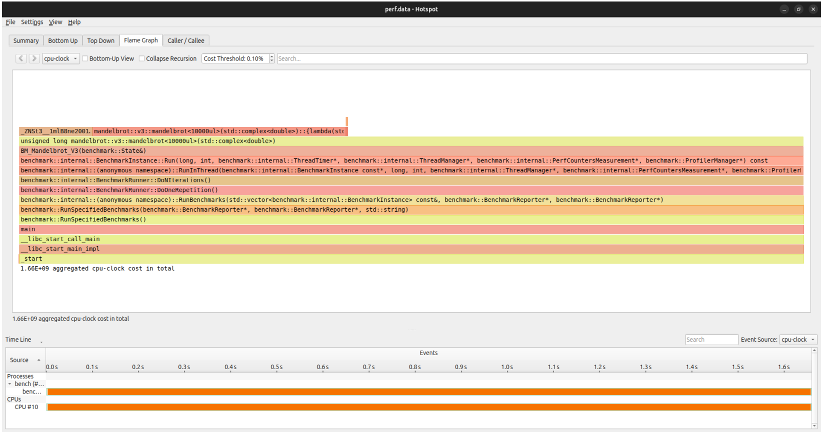
Implementation

```
auto mandelbrot(std::complex<double> c) -> std::size_t {
  auto iter = std::size_t{};
  auto z = std::complex<double>{};

  auto not_escaped = [](std::complex<double> z) {
    auto x = z.real();
    auto y = z.imag();
    return x * x + y * y <= 4.0;
  };

  while (not_escaped(z) and iter < MAX_ITER) {
    z = z * z + c;
    ++iter;
  }
  return iter;
}
```

Performance Analysis



- Both implementations have nearly identical performance
- `std::norm` templated code is fully inlined by the compiler

Removing Abstractions

Why Remove `std::complex`?

- Potential overhead from abstraction
- Want explicit control over operations
- Better optimization opportunities

Mathematical Derivation

Starting with $z = x + yi$ and $c = a + bi$:

$$\begin{aligned}
z^2 + c &= (x + yi)^2 + (a + bi) \\
&= (x^2 + 2xyi + (yi)^2) + (a + bi) \\
&= (x^2 + 2xyi + y^2 i^2) + (a + bi) \\
&= (x^2 + 2xyi - y^2) + (a + bi) \quad \text{since } i^2 = -1 \\
&= x^2 + 2xyi - y^2 + a + bi \\
&= x^2 - y^2 + a + 2xyi + bi \\
&= (x^2 - y^2 + a) + i(2xy + b)
\end{aligned}$$

No Abstraction

Implementation

```
auto mandelbrot(std::complex<double> c) -> std::size_t {
  auto const a = c.real();
  auto const b = c.imag();
  auto iter = std::size_t{};
  auto x = 0.0;
  auto y = 0.0;

  while (x * x + y * y <= 4.0 and iter < MAX_ITER) {
    auto x_next = x * x - y * y + a;
    auto y_next = 2 * x * y + b;
    std::tie(x, y) = std::tie(x_next, y_next);
    ++iter;
  }
  return iter;
}
```

Details

- Uses `std::tie` for clean value updates
- Avoids temporary complex number objects
- Easier for compiler to see operation dependencies
- Better quality codegen, is faster

Computation Reuse

Key Insight / Experiment

- x^2 and y^2 are used in both the escape check and value update
- We can compute them once per iteration and reuse them
- Reduces redundant multiplications

Optimized Implementation

```cpp
auto mandelbrot(std::complex<double> c) -> std::size_t {
  auto const a = c.real();
  auto const b = c.imag();
  auto iter = std::size_t{};
  auto x = 0.0;
  auto y = 0.0;
  auto x2 = 0.0;  // x squared
  auto y2 = 0.0;  // y squared

  while (x2 + y2 <= 4.0 and iter < MAX_ITER) {
    auto x_next = x2 - y2 + a;
    auto y_next = 2 * x * y + b;
    std::tie(x, y) = std::tie(x_next, y_next);
    y2 = y * y;  // store to reuse in the loop check
    x2 = x * x;  // store to reuse in the loop check
    ++iter;
  }
  return iter;
}
```

Beyond Hotspot: Going Deeper with perf

Hotspot's Limits

- Hotspot was good for seeing the big picture, functions and lines of code
- Now we need instruction-level analysis
- Time to use Linux perf directly for detailed insights

Using perf assembly

```
# Record with call graph and debug info
perf record -g --call-graph=dwarf ./bench
    --benchmark_filter=BM_Mandelbrot_V4/0
    --benchmark_min_time=1s
# View annotated assembly
perf annotate
```

Assembly Comparison

```
        nop
1e0:  ┌─mov     %rcx,0x80(%rsp)
      │ dec     %r14
      │ ↓ je    24f
1ed:  │ xorpd   %xmm1,%xmm1
      │ xor     %eax,%eax
      │ xorpd   %xmm2,%xmm2
      │ nop
2.57    200:  │ movapd  %xmm1,%xmm3
3.27          │ movapd  %xmm2,%xmm4
5.11          │ mulsd   %xmm2,%xmm4
5.60          │ mulsd   %xmm1,%xmm1
5.06          │ subsd   %xmm4,%xmm1
5.68          │ addsd   %xmm5,%xmm1
6.21          │ addsd   %xmm3,%xmm3
6.27          │ mulsd   %xmm3,%xmm2
5.37          │ addsd   %xmm6,%xmm2
5.25          │ lea     0x1(%rax),%rcx
5.48          │ movapd  %xmm2,%xmm3
5.69          │ mulsd   %xmm2,%xmm3
5.63          │ movapd  %xmm1,%xmm4
5.60          │ mulsd   %xmm1,%xmm4
5.44          │ addsd   %xmm3,%xmm4
10.31         │ ucomisd %xmm4,%xmm0
3.18          │ ↑ jb    1e0
3.01          │ cmp     $0x270f,%rax
2.43          │ mov     %rcx,%rax
2.83          │ ↑ jb    200
              └─jmp     1e0
        24f:    mov     %rbx,%rdi
```

No Abstraction

```
              cs      nopw 0x0(%rax,%rax,1)
        1e0:  ┌─mov     %rcx,0x80(%rsp)
              │ dec     %r14
              │ ↓ je    247
        1ed:  │ xorpd   %xmm3,%xmm3
              │ xorpd   %xmm2,%xmm2
              │ xor     %eax,%eax
              │ xorpd   %xmm4,%xmm4
              │ xorpd   %xmm1,%xmm1
              │ nop
5.98    200:  │ subsd   %xmm3,%xmm2
6.27          │ addsd   %xmm4,%xmm4
7.03          │ mulsd   %xmm4,%xmm1
7.71          │ movapd  %xmm2,%xmm4
7.36          │ addsd   %xmm6,%xmm4
8.27          │ addsd   %xmm7,%xmm1
6.01          │ movapd  %xmm1,%xmm3
5.85          │ mulsd   %xmm1,%xmm3
4.75          │ movapd  %xmm4,%xmm2
4.67          │ mulsd   %xmm4,%xmm2
4.10          │ lea     0x1(%rax),%rcx
4.27          │ movapd  %xmm2,%xmm5
3.98          │ addsd   %xmm3,%xmm5
8.02          │ ucomisd %xmm5,%xmm0
3.69          │ ↑ jb    1e0
3.49          │ cmp     $0x270f,%rax
4.09          │ mov     %rcx,%rax
4.42          │ ↑ jb    200
0.03          └─jmp     1e0
        247:    mov     %rbx,%rdi
```

Computation Reuse

Assembly Comparison

Key Observations

- Manual reuse can still be a beneficial optimization
  - No abstraction: 4x moves, 5x multiplications, 6x add/subs
  - Computation reuse: 4x moves, 3x multiplications, 6x add/subs
- 2 fewer multiplications per iteration with reuse
- Manual optimization beneficial despite compiler optimizations

Modern x86_64 Microarchitecture Levels

Our Current Code: Using x86_64-v1 Instructions

- Using `mulsd` (Multiply Scalar Double)
- Part of SSE2 (Streaming SIMD Extensions 2)
- Operates on just 2 registers at a time

x86_64 Microarchitecture Levels

x86_64-v2 POPCNT, CMPXCHG16B

- Common since 2011 (Nehalem/Bulldozer)

x86_64-v3 AVX, AVX2, BMI1/2, FMA, LZCNT

- Common since 2015 (Haswell/Excavator)

Modern Compiler Targeting

CMake Configuration

```
target_compile_options(
    bench PRIVATE -march=x86-64-v3 -mtune=native
)
```

Why Target x86_64-v3?

- Realistic for today's deployments
- Enables vectorized operations (AVX/AVX2)
- Modern math instructions (FMA)
- Bit manipulation (BMI1/2)
- Better bit operations (LZCNT)

Assembly Comparison x86_64-v3

```
              data16      data16 cs nopw 0x0(%rax
       1e0:   mov         %rcx,0x80(%rsp)
              dec         %r14
            ↓ je          23a
       1ed:   vxorpd      %xmm1,%xmm1,%xmm1
              vxorpd      %xmm2,%xmm2,%xmm2
              xor         %eax,%eax
              nop
 5.64  200:   vmulsd      %xmm2,%xmm2,%xmm4
 5.47         vmovapd     %xmm2,%xmm3
 5.26         vaddsd      %xmm1,%xmm1,%xmm2
 5.54         lea         0x1(%rax),%rcx
 5.14         vfmsub231sd %xmm1,%xmm1,%xmm4
 5.45         vfmadd213sd %xmm6,%xmm3,%xmm2
 9.16         vaddsd      %xmm4,%xmm5,%xmm1
 9.30         vmulsd      %xmm2,%xmm2,%xmm3
 9.64         vfmadd231sd %xmm1,%xmm1,%xmm3
19.08         vucomisd    %xmm3,%xmm0
 9.05       ↑ jb          1e0
 3.60         cmp         $0x270f,%rax
 3.75         mov         %rcx,%rax
 3.91       ↑ jb          200
 0.03         jmp         1e0
       23a:   mov         %rbx,%rdi
```

No Abstraction

```
              data16      data16 cs nopw 0x0(%rax
       1e0:   mov         %rcx,0x80(%rsp)
              dec         %r14
            ↓ je          238
       1ed:   vxorpd      %xmm1,%xmm1,%xmm1
              vxorpd      %xmm2,%xmm2,%xmm2
              vxorpd      %xmm3,%xmm3,%xmm3
              vxorpd      %xmm4,%xmm4,%xmm4
              xor         %eax,%eax
              nop
 3.67  200:   vmovapd     %xmm4,%xmm5
 4.70         vaddsd      %xmm3,%xmm3,%xmm4
 6.82         vsubsd      %xmm1,%xmm2,%xmm1
 6.46         lea         0x1(%rax),%rcx
 7.78         vfmadd213sd %xmm7,%xmm5,%xmm4
10.05         vaddsd      %xmm1,%xmm6,%xmm3
 9.78         vmulsd      %xmm3,%xmm3,%xmm2
 9.87         vmulsd      %xmm4,%xmm4,%xmm1
 7.94         vaddsd      %xmm1,%xmm2,%xmm5
16.23         vucomisd    %xmm5,%xmm0
 4.30       ↑ jb          1e0
 4.66         cmp         $0x270f,%rax
 3.39         mov         %rcx,%rax
 4.36       ↑ jb          200
              jmp         1e0
       238:   mov         %rbx,%rdi
```

Computation Reuse

Assembly Comparison x86_64-v3

Key Observations

- Now the simpler code is faster, our computation reuse now hinders codegen
  - No abstraction: 1x move, 2x multiplications, 3x add/subs 3x fma
  - Computation reuse: 1x move, 2x multiplications, 5x add/subs, 1x fma
- Not obvious why the simpler code is faster
- Sometimes KISS (keep it simple & straightforward) is better

Pipeline Analysis with LLVM-MCA

What is LLVM-MCA?

- A performance analysis tool that simulates CPU pipeline behavior
- Models out-of-order execution, register renaming, and instruction scheduling
- Helps identify pipeline stalls, port pressure, and bottlenecks

Why Use It?

- Evidence driven approach to understand non-obvious performance differences
- Identifies resource contention and dependencies
- Helps understand the impact of instruction scheduling
- Particularly useful for analyzing microarchitecture-specific behavior

LLVM-MCA Analysis

Commands Used

```
clang++-20 mandelbrot.cpp -std=c++23 -O3 -S -o - |
    llvm-mca-20 -mcpu=haswell
```

LLVM-MCA Analysis

Results Comparison

Metric	No Abstraction	Computation Reuse
Iterations	100	100
Instructions	1,900	2,100
Total Cycles	572	621
Total uOps	2,100	2,300
uOps/Cycle	3.67	3.70
IPC	3.32	3.38
Block RThroughput	**5.3**	**5.8**

Key Insight

The no-abstraction version has better (lower) Block RThroughput (5.3 vs 5.8), meaning it has better port scheduling and hence hides its latency better.

What about SIMD?

Single Instruction, Multiple Data

- A parallel computing technique where one instruction operates on multiple data elements simultaneously
- Modern CPUs have dedicated SIMD instruction sets (SSE, AVX, AVX2, AVX-512)
- Perfect for operations on arrays, vectors, and mathematical computations

Benefits for Mandelbrot Generation

- Calculate multiple results simultaneously
- Better utilization of CPU resources
- Mandelbrot is embarrassingly parallel

Scalar Addition (Traditional)

A:	1.0	2.0	3.0	4.0
B:	5.0	6.0	7.0	8.0
	+	+	+	+
C:	6.0	8.0	10.0	12.0

Scalar Processing

- 4 separate CPU instructions required
- Sequential execution
- One operation per clock cycle

SIMD Addition (AVX2)

A:	**1.0**	**2.0**	**3.0**	**4.0**
B:	**5.0**	**6.0**	**7.0**	**8.0**

+ **Single SIMD instruction**

C:	**6.0**	**8.0**	**10.0**	**12.0**

SIMD Processing

- Single CPU instruction (e.g., `vaddpd`)
- Parallel execution within one instruction
- 4x theoretical speedup for this example

SIMD Mandelbrot Implementation

xsimd-based Implementation

```cpp
auto mandelbrot(xsimd::batch<double> a, xsimd::batch<double> b)
    -> xsimd::batch<std::size_t> {
  using batch = xsimd::batch<double>;
  using bsize = xsimd::batch<std::size_t>;

  auto const four = batch(4.0);
  auto const two = batch(2.0);
  auto const one = bsize(1);

  auto x = batch(0.0);
  auto y = batch(0.0);
  auto iter = bsize(0);

  for (std::size_t i = 0; i < MAX_ITER; ++i) {
    auto const x2 = x * x;
    auto const y2 = y * y;

    auto const mask = (x2 + y2) <= four;
    if (none(mask)) {
      break;
    }

    auto const xy = x * y;
    auto const mask_i = batch_bool_cast<std::size_t>(mask);

    x = x2 - y2 + a;
    y = fma(two, xy, b);
    // Only update where still running
    iter = select(mask_i, iter + one, iter);
  }

  return iter;
}
```

SIMD Mandelbrot Key Concepts

Processing Multiple Complex Numbers

- `xsimd::batch<double>` gives largest size registers for your architecture, AVX2 would pack 4 doubles
- `mask = (x2 + y2) <= four` creates boolean mask for non-diverged points
- `select(mask, new_value, old_value)` conditionally updates only active points
- `if (none(mask))` early exit when all points have diverged

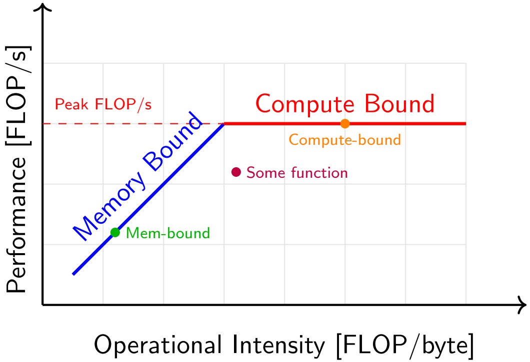
Performance Benefits

- 4x theoretical speedup from processing 4 complex numbers simultaneously
- Efficient handling of divergence without branching
- Modern CPU features: FMA, efficient mask operations

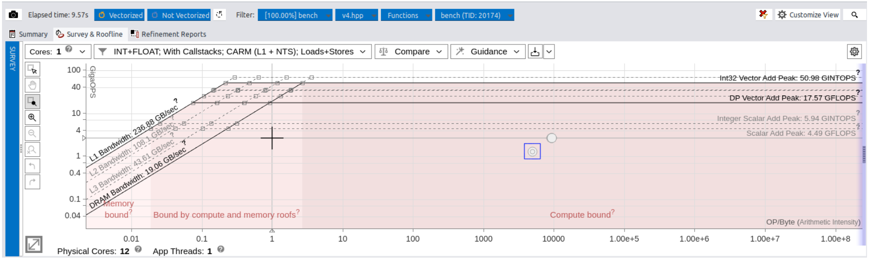
Intel Advisor for Performance Analysis

What is Intel Advisor?

- Performance profiling and optimization tool from Intel
- Provides vectorization analysis and roofline modeling
- Identifies optimization opportunities in CPU-bound applications



Compare scalar to SIMD



V4: No Abstraction (Scalar)

V6: SIMD Vectorization

Possible extra SIMD Optimizations choices

Loop Unrolling Strategies

- **Reduced escape checking**: Check every 4, 8, or 16 iterations
- **Compiler hints**: `#pragma clang loop unroll_count(16)`
- **Benefit**: Reduces branch prediction penalties

Is it worth it?

What is the likelihood of all members escaping? Are we ok to delay early exit by upto N-1 iterations?

SIMD with Loop Unrolling Implementation

unrolled xsimd-based Implementation

```
auto mandelbrot(xsimd::batch<double> a, xsimd::batch<double> b)
    -> xsimd::batch<std::size_t> {
  using batch = xsimd::batch<double>;
  using bsize = xsimd::batch<std::size_t>;

  auto const four = batch(4.0);
  auto const two = batch(2.0);
  auto const one = bsize(1);

  auto x = batch(0.0);
  auto y = batch(0.0);
  auto iter = bsize(0);

#pragma clang loop unroll_count(16)
  for (std::size_t i = 0; i < MAX_ITER; ++i) {
    auto const x2 = x * x;
    auto const y2 = y * y;

    auto const mask = (x2 + y2) <= four;
    if (i % 16 == 0 and none(mask)) {
      break;
    }

    auto const xy = x * y;
    auto const mask_i = batch_bool_cast<std::size_t>(mask);

    x = x2 - y2 + a;
    y = fma(two, xy, b);
    // Only update where still running
    iter = select(mask_i, iter + one, iter);
  }

  return iter;
}
```

Modern CPUs: Beyond Single Core

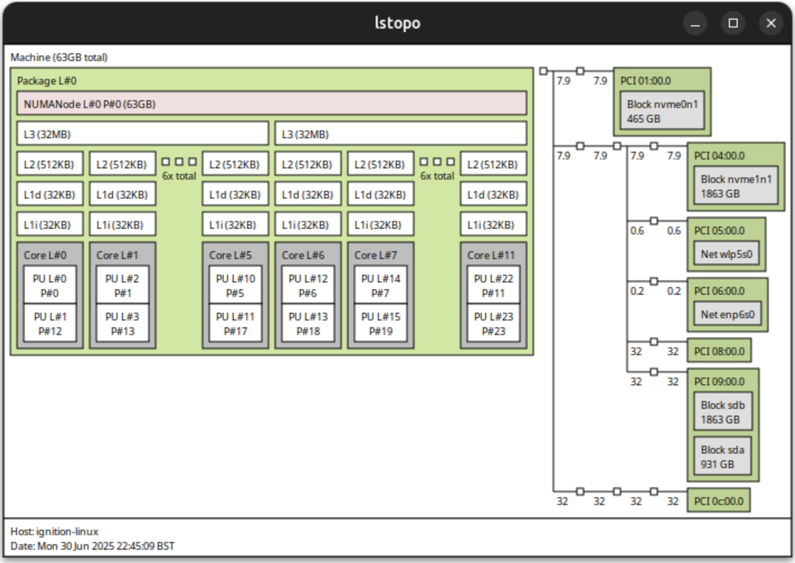
The Multi-Core Reality

- CPUs are no longer typically single core
- Even budget processors now have 4+ cores
- High-end consumer CPUs: 8-24+ cores
- Server CPUs: 64+ cores becoming common

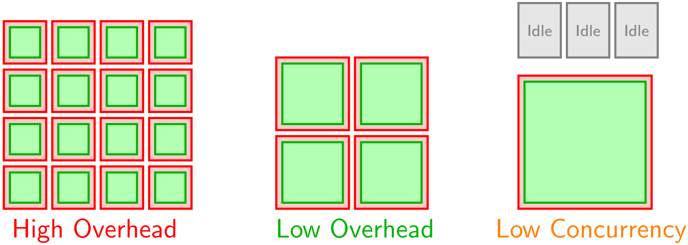
Why Single-Threaded Limits Performance

- Only utilizing 1/N of available CPU resources
- Mandelbrot calculation is embarrassingly parallel
- Each pixel calculation is independent
- Perfect candidate for multithreading

Hardware Topology



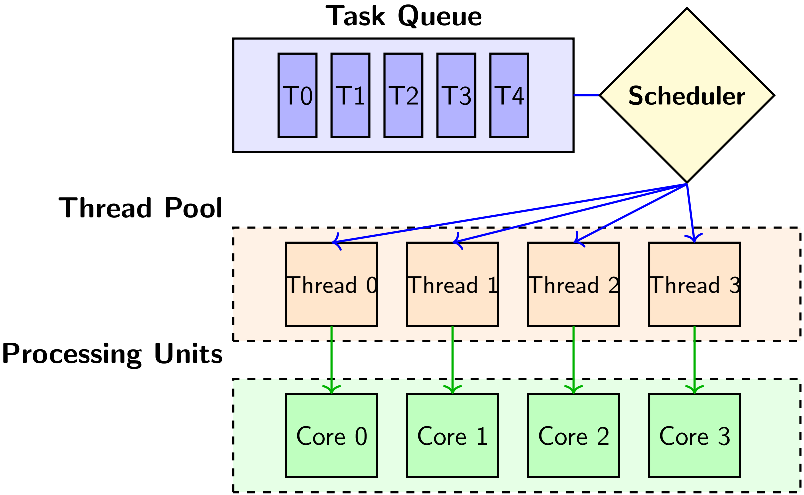
Concurrent Overhead Visualization



Overhead vs Useful Work

- Red border: Scheduling/synchronization/concurrency mechanism overhead
- Green fill: Useful computation work

Thread Pool Task Scheduling



Multithreaded SIMD Implementation

stdexec-based Implementation

```
void mandelbrot(std::vector<xsimd::batch<std::size_t>> &vec,
                auto &&gen, auto scheduler) {
  auto sender = stdexec::bulk(
    stdexec::schedule(scheduler),
    stdexec::par,
    vec.size(),
    [&](std::size_t i) {
      vec[i] = std::apply(mandelbrot_simd, gen(i));
    });

  stdexec::sync_wait(sender);
}
```

Key Components

- **stdexec::bulk**: Splits the work to be dones as multiple chunked tasks
- **stdexec::par**: Parallel execution policy
- **scheduler**: Dispatch + coordinate via a thread pool
- **gen(i)**: Generates coordinates for batch `i`
- **mandelbrot_simd**: SIMD version from earlier
- **sync_wait**: Waits for all tasks to complete

Performance Results: Speedup vs Naive

Individual Optimizations

- Scalar optimizations: 1.8-2.2x improvement
- SIMD vectorization: 8-8.5x improvement
- Multithreading: 41x improvement (on 24 processing units)
- Combined MT + SIMD: **174x improvement**

The Journey: From Maths to Machines

Mathematical Foundation

- Started with elegant mathematical formula: $z_{n+1} = z_n^2 + c$
- Explored the complex plane and fractal geometry
- Translated mathematical concepts into computational algorithms

Tools Gave Us Insight

- **Profiling tools**: Hotspot, perf, Intel Advisor revealed bottlenecks
- **Assembly analysis**: Understanding what the compiler actually generates
- **LLVM-MCA**: Pipeline analysis for microarchitecture optimization
- **Benchmarking**: Evidence-driven development with Google Benchmark

Understanding Your Platform

Hardware Awareness Unlocks Performance

- **CPU microarchitecture**: x86_64-v3 enabled FMA and other better instructions
- **SIMD capabilities**: AVX2 provided 4x theoretical speedup, achieved 8.0x
- **Multi-core reality**: 24 processing units → 174x final speedup

Task Granularity is Critical

- Too fine: Overhead dominates useful work
- Too coarse: Poor load balancing and resource waste
- Sweet spot: Balance between parallelism and scheduling costs

The Learning Never Stops

Technology Evolves Continuously

- **Hardware changes**: New instruction sets, architectures, and capabilities
- **Languages evolve**: C++26 brings new abstractions (sender/receivers) [libstdexec]
- **Libraries advance**: xsimd provides portable SIMD, new algorithms emerge

Fundamental Principles Remain

- Measure first, optimize second
- Understand your problem domain and hardware platform
- Balance abstraction with performance requirements
- Use tools to guide decisions, not assumptions

Final Thoughts

174x Speedup: The Journey Matters

- Mathematical elegance → Clean initial implementation
- Profiling insights → Targeted optimizations (sqrt → norm)
- Assembly understanding → Manual improvements
- Hardware awareness → SIMD vectorization
- Platform knowledge → Effective multithreading

Keep Exploring

- Try other fractal sets (Julia, Burning Ship, Newton)
- Experiment with GPU computing (CUDA, OpenCL, compute shaders)
- Explore distributed computing and cloud scaling
- Different coloring techniques
- Getting past IEEE double precision limits