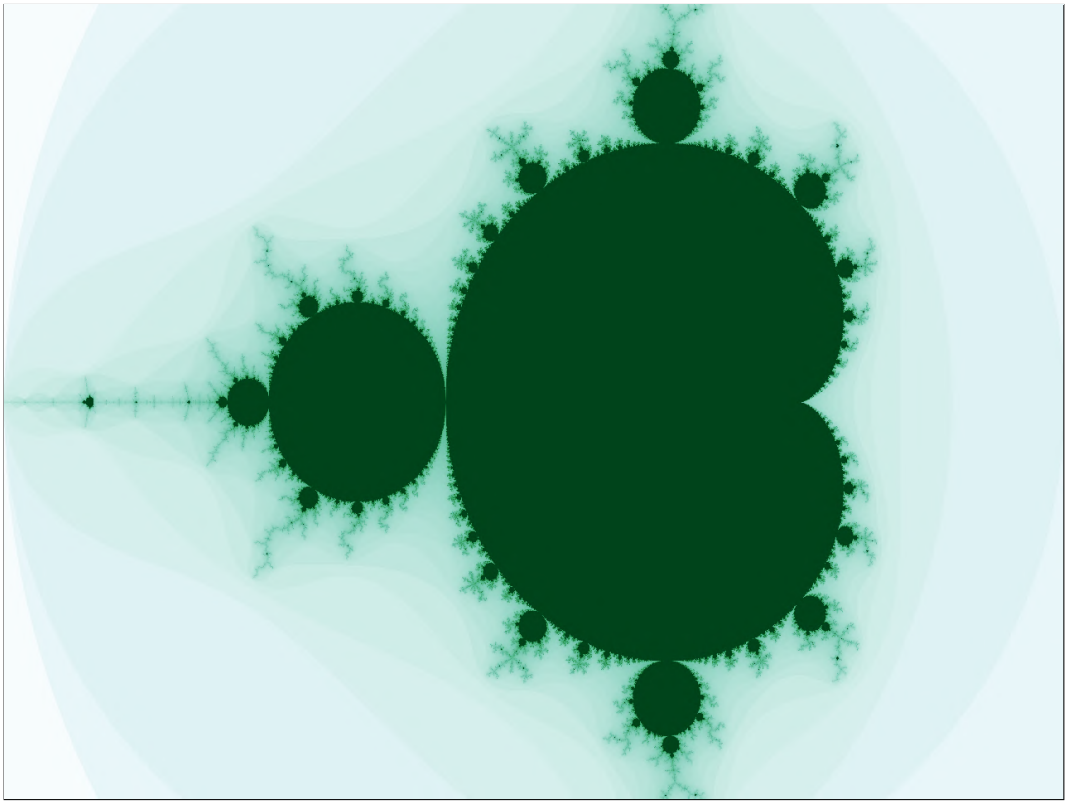
Final report
High Performance Computing

Georg Auge[1] and Alexander Sohn[2]

[1] Hasso Plattner Institute for Digital Engineering
georg.auge@student.hpi.de
[2] Hasso Plattner Institute for Digital Engineering
alexander.sohn@student.hpi.de



A visualization of the Mandelbrot set, illustrating its intricate fractal nature.

1. Introduction

In this report, we showcase our work on optimizing the computation of the Mandelbrot set. We start with a baseline implementation and gradually improve its performance using Intel Advisor and other tools useful for High Performance Computing tasks. By analyzing bottlenecks and applying optimization techniques such as vectorization, parallelization, and memory efficiency, we aim to accelerate computation while maintaining correctness. This report outlines our optimization process, the techniques we apply, and the performance gains we achieve through our refinements.

The Mandelbrot set is defined by the following recursive formula in the complex number plane:

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

with

$$z_0 = (0, 0)$$

For each c the mandelbrot set computes how fast the absolute value of z_n is bigger than a given threshold t. By mapping the coordinates of an image, to the real and imaginary part of c we get the characteristic picture of a mandelbrot set.

2. Project Structure

We decided to split the project into a main file, that handles the input and calls the function `mandelbrot_computation`. Each implementation implements that function, which measures the execution time in milliseconds and returns it to the main function which prints it on stdout. Everything related to debug output is printed on stderr, so it is possible to easily get the runtime for external scripts. The file output is only generated if a filename is supplied as an argument. The file output is also generated in the function `mandelbrot_computation`. This is because we were not sure in the beginning if we might need different representations of the mandelbrot set in memory, that need different code to write this representation to a file. Now we know, that a refactoring, that moves the file write code to the main function would be possible.

The CPU implementations are all in the `src/` folder. Everything related to GPUs is in the folder `gpu/`. Helper scripts are in their separate folders. We keep the results of our measurements as CSV data in our repository. For build instructions and a more detailed explanation of what the scripts do, take a look at the README of our repository.

3. Baseline implementation

For the baseline we are provided an unoptimized implementation of the Mandelbrot set. Listing 1 shows the main part of the code. Although it includes some other parts, including measuring execution time and printing the output, only the part relevant to this report is shown. In order to measure the difference between number representations, we also use a second version of the baseline, which uses double precision representation. The baseline uses the std::complex implementation, that builds an abstraction that keeps track of the real and imaginary part of each number in one entity.

Listing 1: Baseline implementation

```
1  for (int pos = 0; pos < HEIGHT * WIDTH; ++pos)
2  {
3
4      const int row = pos / WIDTH;
5      const int col = pos % WIDTH;
6      const complex<float> c((float)(col * STEP + MIN_X), (float)(row * STEP + MIN_Y));
7
8      // z = z^2 + c
9      complex<float> z(0, 0);
10     for (uint16_t i = 1; i <= ITERATIONS; ++i)
11     {
12         z = pow(z, 2) + c;
13
14         image[pos] = i;
15         // If it is convergent
16         if (abs(z) >= 2)
17         {
18             break;
19         }
20     }
21 }
```

4. Experimental setup

To ensure comparable results across all optimizations, we run all CPU benchmarks on the same server. The machine is a dual socket system equipped with AMD EPYC 7451 24-Core processors and 64 GB of DDR4 RAM. On this server we have a virtual machine that is assigned 32 threads of each socket and operates on Ubuntu 22.04 LTS with Linux kernel version 6.8.0-55. We compile all implementations using

ICPX 2025.0.4, ensuring support for AVX2 vector instructions to leverage hardware acceleration.

GPU benchmarks are run on an NVIDA RTX 2060, NVIDIA T4 and AMD Radeon 780M.

For all implementation regardless of hardware we used the following compiler options as base options:

- `-Wall`
- `-Wextra`
- `-Wshadow`
- `-Wfloat-equal`
- `-Wconversion`
- `-g`

Those options produce debug symbols and advise the compiler to issue warnings for different things, that help to discover bugs during compile time.

For benchmarking, we first perform five warm-up executions to account for caching effects and stabilize performance. We then execute ten benchmark runs, recording the median computation time to minimize the impact of outliers and ensure reliable performance measurements.

To analyze the scaling behavior of our implementations, we control CPU core usage using `taskset`, specifying the number of cores allocated for each run. We perform benchmarks using 1, 2, 4, 8, 16, and 32 cores, ensuring that the assigned cores are contiguous for each run (e.g., cores 0-7 for an 8-core execution). This approach helps maintain consistent CPU affinity and minimizes performance variability due to thread migration across non-adjacent cores. Also this ensures that every run uses only threads from one socket.

5. Correctness check

Combining the different optimization levels, floating-point models, core configurations, and compiler flags results in a large number of benchmarking runs. To efficiently manage this workload, we developed a custom scheduler that automates execution, ensures proper CPU core allocation, and records the results for analysis.

To verify that all the implementations are correct, we developed a script that compares the output files generated by all binaries. For that, we compute a sha256sum for each output file and aggregate the files with the same hash. Especially with higher optimization levels and less strict floating point models, the output between the binaries differs.

We generate a report based on the results of the different implementations. The report can be found in listing 11 in the appendix. In the report we see that

optimization level and floating point model heavily affect the result. One thing to note is, that the number of groups of single precision results is bigger, than the number of groups of double precision results. This is the case, because the precision loss when using the constants like WIDTH, STEP etc. and casting them from double to float, affects the results. To lower the number of single precision groups, one would have to revise the casting so it is consistent among all implementations or use a different header and define all constants as floats. We did not dig deeper into it, because it is not performance relevant and we can see from the double precision results, that all our implementations are correct. Depending on the desired numerical accuracy, an appropriate optimization level and floating point model should be chosen. In this report, we do not focus on this aspect. We only compare the performance of the different levels and models and use this tool to verify that our implementations compute correct values.

6. Optimizations and their performance

A first analysis shows that the computation of the mandelbrot set is structured into two for loops. The outer loop starts the computation for each pixel. The inner loop iterates the recursive computation of the series. The outer loop is a so-called embarrassingly parallel problem. The computation for each pixel can happen independently because there is no need to share any data between computations. The inner loop however is not easily parallelizable because there are dependencies. Each iteration depends on all previous iterations. For this reason we focused on parallelizing the outer loop of the computations. We used mainly two parallelizing techniques. Thread based parallelism with openmp and vectorization using SIMD instructions. SIMD is short for Single Instruction Multiple Data and is implemented on CPUs using the SSE and AVX instruction set.

From taking a look at the mathematical definition, we can see that the mandelbrot set is symmetrical to the real line. This means that an implementation can save computations by only computing either the values below or above the real line. An implementation can have a speed up of two if the window where to compute is symmetric to the real line. In the worst case it has a speed up of one if the window is completely above or below the real line. We chose not to use the symmetry for our implementations, because it makes the implementations harder to read and we expect a similar speed up for all implementations. So to compare the performance of different approaches, it is not necessary and adapting the window can later be done for the most promissing approach.

6.1. OpenMP-based parallelization

To leverage multiple CPU cores, we parallelize the outer loop of our Mandelbrot set implementation using OpenMP. Distributing the computation across multiple threads can significantly improve performance.

6.1.1. Static multithreading

Our first parallelization approach uses static scheduling, where loop iterations are evenly divided among available threads at the beginning of execution:

Listing 2: OpenMP-based parallelization using static multithreading

```
1 #pragma omp parallel for
2 for (int pos = 0; pos < HEIGHT * WIDTH; ++pos)
3 {
4                                       ⋮
5 }
```

With static scheduling, each thread receives a fixed chunk of iterations, ensuring minimal scheduling overhead. This works well when all iterations have similar execution times, as the threads remain balanced throughout execution.

6.1.2. Dynamic multithreading

In contrast, dynamic scheduling assigns iterations to threads on demand, allowing for better load balancing when iteration costs vary:

Listing 3: OpenMP-based parallelization using dynamic multithreading

```
1 #pragma omp parallel for schedule(dynamic, 1)
2 for (int pos = 0; pos < HEIGHT * WIDTH; ++pos)
3 {
4                                       ⋮
5 }
```

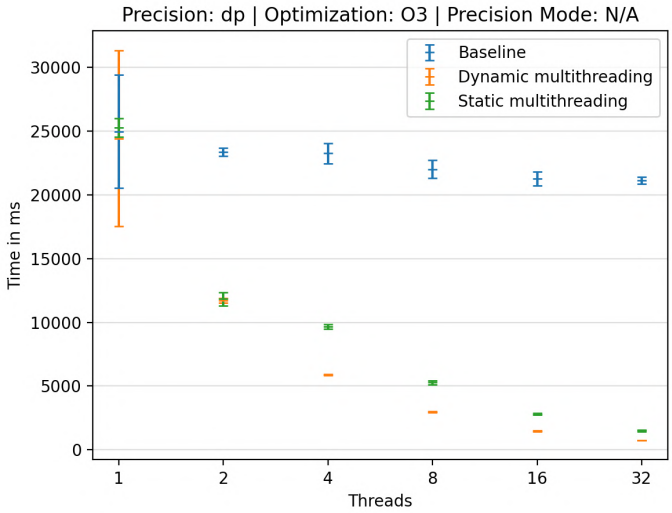
Here, `schedule(dynamic, 1)` assigns one iteration at a time to the next available thread. This helps when iterations have uneven workloads, preventing some threads from finishing early while others remain busy.

6.1.3. Performance Comparison

We benchmarked both scheduling methods as well as the baseline implementation while keeping all other parameters fixed. Figure 2 below shows the execution times (in milliseconds) for different numbers of threads.

Our results show that dynamic scheduling consistently outperforms static scheduling, especially as the number of threads increases. This makes intuitive sense in the context of the Mandelbrot set: some coordinates escape quickly (few iterations),

Figure 2: Performance of static and dynamic parallelization



while others remain in the loop for longer. With static scheduling, some threads may get many "heavy" iterations while others get "light" ones, leading to load imbalance. Dynamic scheduling prevents this by redistributing work in real time, ensuring all threads remain active. This can also be seen in the Intel Advisor trip count data. The inner loop has a minimal trip count of 1 and a maximum trip count of the number of iterations.

Since dynamic scheduling significantly outperforms static scheduling in our previous experiments, we now base further optimizations on this approach.

6.2. Leveraging Vectorization

After implementing parallelism, we thought about vectorizing our code. Vectorizing the inner loop would be difficult, because the iterations depend on each other. So we have to wait, until the previous iterations finish. However the outer loop together with the inner loop is vectorizable. Because between pixels there are no dependencies, we can compute multiple pixels in parallel. Each entry in our vectors corresponds to one pixel. Modern compilers can automatically vectorize code. Another way of implementing vectorization is using intrinsics that produce vector instructions. Both approaches have advantages and disadvantages. We implemented both and compared their performance.

6.2.1. Automatic Vectorization

We look at the compiler reports to find out, if the compiler might already be using vectorization. This shows, that the compiler does not vectorize the inner loop, because it can not resolve the flow dependencies and it does not consider the outer loop, as a vectorization candidate, so there is no vectorization applied. As discussed

earlier, the outer loop is vectorizable. By using the openMP `simd` pragma, we can force the compiler to consider the outer loop for vectorization. How this pragma can be used together with the parallelism is shown in listing 4. From analyzing the Intel icpx compiler reports, we found that the compiler now vectorizes the code but struggles to do so effectively. The optimization report in listing 5 for the dynamic scheduling with SIMD enabled in the pragma shows, that no vectorized version of the `abs` method is available. This means that the compiler produces vectorized code, that has a sequential part. This limits the performance and we expect to see almost no speed up.

Listing 4: Leveraging SIMD with Collapsed Loops

```
1 #pragma omp parallel for simd collapse(2) schedule(dynamic, 100)
2 for (int pos = 0; pos < HEIGHT * WIDTH; ++pos)
3 {
4                                       ⋮
5 }
```

Listing 5: Optimization report from dynamic_multithreading_dp_simd_O3.txt

```
LOOP BEGIN at parallel-mandelbrot/src/dynamic_multithreading_dp_simd.cpp (21, 21)
    remark #25530: Stmt at line 0 sinked after loop using last value computation
    remark #25438: Loop unrolled without remainder by 2
    remark #15475: --- begin vector loop cost summary ---
    remark #15485: serialized function calls: 2
    remark #15558: Call to function 'cabs' was serialized
        due to no suitable vector variants were found.
    remark #15558: Call to function 'cabs' was serialized
        due to no suitable vector variants were found.
    remark #15488: --- end vector loop cost summary ---
    remark #15447: --- begin vector loop memory reference summary ---
    remark #15474: --- end vector loop memory reference summary ---
LOOP END
```

Also in the Intel Advisor report (Figure 3) it is shown, that still most time is spent computing the absolute value of the complex number. To address this, there are different options. One is to look for a complex type method that can replace `abs` and is vectorizable. The other is to implement complex arithmetic on our own and to keep track of the real and imaginary parts in two separate variables.

Our hypothesis is that the `abs` call can not be vectorized because it uses the square root. However, the square root is not strictly necessary for our use case. We use the `abs` value only in the inequality to check the break condition. Instead, we can square both sides of the inequality to get rid of the square root. The complex

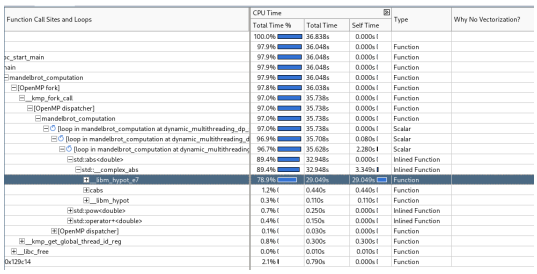


Figure 3: Most time is spent for computing the absolute value

type has a function called `norm`, to compute the squared magnitude of a complex number. This method is vectorizable by the compiler as can be seen in the compiler report listing 6. In our code we call the implementation with `norm` instead of `abs` optimized version.

Listing 6: Optimization report from dynamic_multithreading_dp_simd_optimized_O3.txt

```
LOOP BEGIN at parallel-mandelbrot/
    src/dynamic_multithreading_simd_optimized_dp.cpp (8, 5)

    remark #30000: OpenMP: Outlined parallel loop
    remark #15301: SIMD LOOP WAS VECTORIZED
    remark #15305: vectorization support: vector length 8
    remark #15389: vectorization support: unmasked unaligned unit stride
        store: [ parallel-mandelbrot/
        src/dynamic_multithreading_simd_optimized_dp.cpp (22, 9) ]
    remark #15475: --- begin vector loop cost summary ---
    remark #15476: scalar cost: 46.000000
    remark #15477: vector cost: 20.640625
    remark #15478: estimated potential speedup: 2.218750
    remark #15309: vectorization support: normalized vectorization overhead 0.046875
    remark #15488: --- end vector loop cost summary ---
    remark #15447: --- begin vector loop memory reference summary ---
    remark #15451: unmasked unaligned unit stride stores: 1
    remark #15474: --- end vector loop memory reference summary ---

    LOOP BEGIN at parallel-mandelbrot/
        src/dynamic_multithreading_simd_optimized_dp.cpp (17, 17)

        remark #25530: Stmt at line 0 sinked after loop using last value computation
        remark #25439: Loop unrolled with remainder by 2
        remark #15475: --- begin vector loop cost summary ---
        remark #15488: --- end vector loop cost summary ---
        remark #15447: --- begin vector loop memory reference summary ---
        remark #15474: --- end vector loop memory reference summary ---
    LOOP END

    LOOP BEGIN at parallel-mandelbrot/
```

```
            src/dynamic_multithreading_simd_optimized_dp.cpp (15, 9)
        <Remainder loop>
            remark #25436: Loop completely unrolled by 1
        LOOP END
    LOOP END
```

Now we want to investigate how we can replace the complex type, to allow for effective vectorization. To be able to compute the real and imaginary parts separately, we must understand how complex arithmetic works. Let us look at the mathematical definition and the arithmetic operations when using the squared magnitude:

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

with

$$z_0 = 0$$

The series is assumed to diverge if

$$|z_n| > 2\sqrt{z_n^2} \geq 2z_n^2 \geq 4$$

For a complex number $z_n = a_n + i\,b_n$, the separate calculations for the real and imaginary part look like this:

$$a_{n+1} = a_n^2 - b_n^2 + c_{\text{re}},$$

$$b_{n+1} = 2\,a_n\,b_n + c_{\text{im}}.$$

Divergence is checked using the condition:

$$a_n^2 + b_n^2 \geq 4.$$

Note how the squares computed for the divergence check can be reused at the start of the next iteration. So we can not only get rid of the square root calculation, so we get vectorizable code, but we are able to save instructions by reusing the intermediate results of the squared magnitude calculation. We see that the compiler is able to vectorize this code as well, by looking at the optimization report in listing 7.

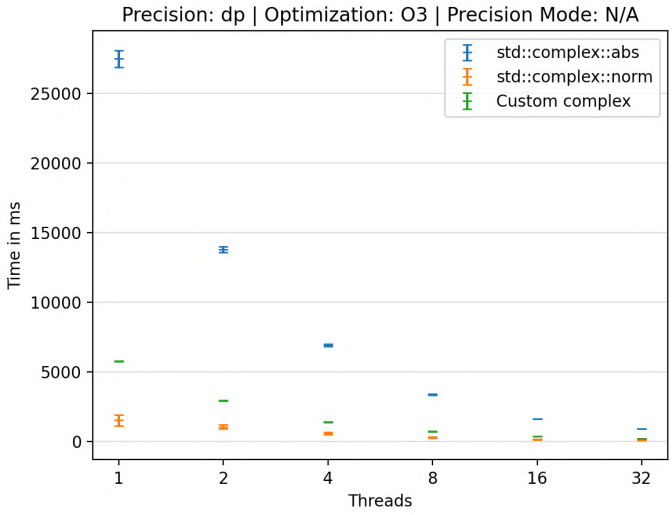
As shown in figure 4 below, both approaches result in a notable performance improvement, demonstrating that vectorization leads to notable performance gains. This also correlates with the speedup calculations done by the compiler.

Listing 7: Optimization report from no_complex_dp_simd_O3.txt

```
remark #30000: OpenMP: Outlined parallel loop
remark #15301: SIMD LOOP WAS VECTORIZED
remark #15305: vectorization support: vector length 4
remark #15389: vectorization support: unmasked unaligned unit stride
    store: [ parallel-mandelbrot/src/no_complex_dp_simd.cpp (33, 9) ]
remark #15475: --- begin vector loop cost summary ---
```

```
remark #15476: scalar cost: 46.000000
remark #15477: vector cost: 23.765625
remark #15478: estimated potential speedup: 1.921875
remark #15309: vectorization support: normalized vectorization overhead 0.031250
remark #15488: --- end vector loop cost summary ---
remark #15447: --- begin vector loop memory reference summary ---
remark #15451: unmasked unaligned unit stride stores: 1
remark #15474: --- end vector loop memory reference summary ---
```

Figure 4: Performance of custom complex numbers vs. `std::complex`



6.2.2. Manual Vectorization

In addition to using OpenMP's automatic SIMD vectorization, we manually implemented vectorization using AVX2 intrinsics to further optimize the Mandelbrot computation. By explicitly loading and processing multiple values in parallel using AVX2 registers, we aim to achieve better utilization of the CPU's vector processing capabilities. We used the no complex implementation as a starting point, because for the manual vectorization, we need to keep track of real and imaginary part in separate variables as well. Computing the values of z_n is pretty straight forward. The hardest part is to keep track of the iterations for each part of the vector separately. We do that by using the mask we get from the compare operation. We increment the counter vector only in those places where the mask is true. We leave the iterations, when all values in the mask are false.

6.2.3. Optimized manual vectorization

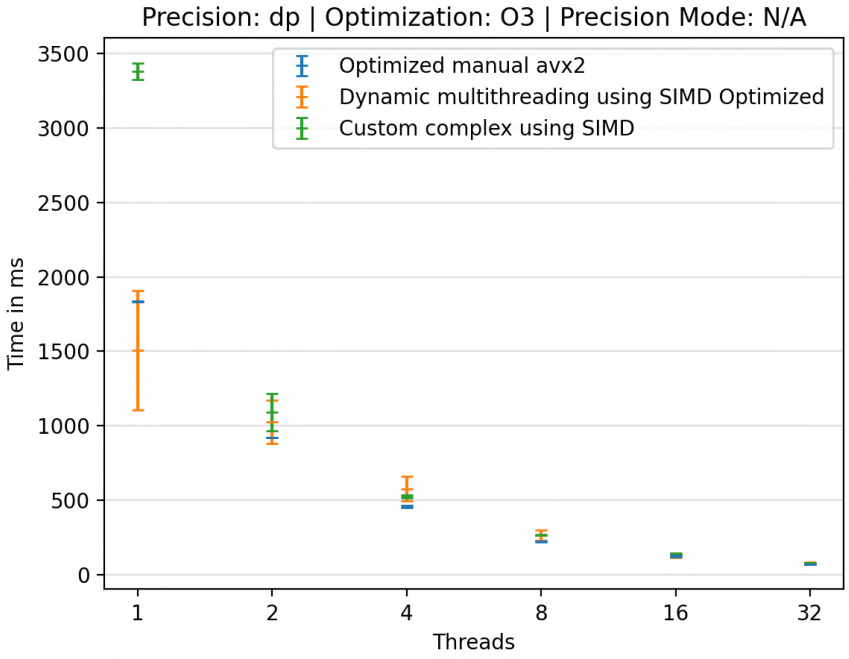
For the optimized implementation we try to get rid of as many instructions as possible in the inner loop or to replace them by less costly instructions. We achieve

this by reusing the square of the real part that we compute for the magnitude as described in section 6.2. This results in less frequent square operations for `zi` and `zr`. Further, the counter is converted to an Integer, which requires one less conversion step before saving it to memory.

6.2.4. Performance Results of Vectorization

We see the results of our performance measurements for the different SIMD implementations in figure 5. We left out the version which use `abs` because it is too slow and makes the other results hard to read. We see that all three implementations are pretty close together in this graph which means that the gains by using manual vectorization are not too big and the compiler can do a decent job at vectorization.

Figure 5: Performance of automatic and manual SIMD vectorization



7. Using Accelerators

In high-performance computing, a variety of hardware accelerators are available to optimize different types of workloads. One prominent class of accelerators is graphics processing units (GPUs), which are particularly well-suited for highly parallelizable computations. To facilitate general-purpose computing on GPUs, most vendors provide specialized toolkits. NVIDIA offers the CUDA framework,

while AMD provides ROCm, enabling developers to harness the computational power of their respective hardware.

Given the diversity of available accelerators, developing code that supports multiple architectures presents a significant challenge. To address this, high-level abstractions have been introduced, allowing developers to write code that is portable across different hardware platforms. One such framework is SYCL, developed by Intel, which enables the definition of hardware-agnostic compute kernels. SYCL allows for compilation to multiple backends, enabling a single binary to support different accelerators and dynamically select the appropriate hardware at runtime, improving both portability and flexibility in heterogeneous computing environments.

7.1. CUDA

CUDA is the toolkit that NVIDIA provides for their GPUs. It is well known and probably the most used. For CUDA we define a kernel, that is similar to the CPU implementation which uses a custom implementation of `std::complex`. The kernel is shown in listing 8. The kernel gets launched on a 2D grid and the column and row get calculated from the position on the grid.

Listing 8: CUDA double precision kernel

```
__global__ void mandelbrot_kernel_2D(uint16_t* __restrict__ image) {
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    if (row < HEIGHT && col < WIDTH) {
        double cr = col * STEP + MIN_X;
        double ci = row * STEP + MIN_Y;
        uint16_t iter = 1;
        double zr = 0.0f, zi = 0.0f;
        // Compute Mandelbrot iterations: z = z^2 + c.
        for (; iter <= ITERATIONS; iter++) {
            double new_zr = zr * zr - zi * zi + cr;
            double new_zi = 2.0f * zr * zi + ci;
            zr = new_zr;
            zi = new_zi;
            if (zr * zr + zi * zi >= 4.0f)
                break;
        }
        image[row * WIDTH + col] = iter;
    }
}
```

In the driver code, it is necessary to allocate the memory on the GPU and to copy the result from the GPU memory to the host buffer in the end. This causes some considerable overhead, as can be seen in the runtime analysis of the GPU approaches.

7.2. ROCm

ROCm is AMD's GPU programming framework, designed to enable general-purpose computing on AMD GPUs. A key distinction from NVIDIA's CUDA is that ROCm is open source, providing greater transparency and flexibility for developers. However, ROCm adoption remains limited, as AMD officially supports only a subset of its high-end GPUs, making it less accessible for consumer-grade hardware.

Despite these limitations, we experimented with ROCm on our own hardware. As shown in Listing 9, the ROCm kernel closely resembles the CUDA implementation. This similarity is intentional, as AMD aims to simplify the transition for developers familiar with CUDA, lowering the barrier to entry for programming AMD GPUs.

Listing 9: ROCm double precision kernel

```
__global__ void mandelbrot_kernel_2D(uint16_t* __restrict__ image) {
    // Compute 2D thread indices.
    uint col = blockIdx.x * blockDim.x + threadIdx.x;
    uint row = blockIdx.y * blockDim.y + threadIdx.y;

    if (row < HEIGHT && col < WIDTH) {
        double cr = col * STEP + MIN_X;
        double ci = row * STEP + MIN_Y;
        uint16_t iter = 1;
        double zr = 0.0f, zi = 0.0f;

        // Compute Mandelbrot iterations: z = z^2 + c.
        for (; iter <= ITERATIONS; ++iter) {
            double new_zr = zr * zr - zi * zi + cr;
            double new_zi = 2.0f * zr * zi + ci;
            zr = new_zr;
            zi = new_zi;
            if (zr * zr + zi * zi >= 4.0f)
                break;
        }
        // Write the iteration count for the current pixel.
        image[row * WIDTH + col] = iter;
    }
}
```

The kernel launch syntax differs from that of CUDA, but the steps involved in the driver code remain the same. One issue we encountered was that for larger workloads, launching the kernel caused the display manager to crash. This occurs because the display manager imposes a timeout on scheduled jobs, terminating any process that exceeds this limit.

To address this, we implemented a tiled version of the ROCm code, where multiple smaller kernel launches are responsible for processing individual tiles. This approach allows the display manager to schedule other tasks between kernel

executions, preventing timeouts. However, to enable execution of large workloads without tiling, we conducted all ROCm performance tests with the display manager disabled.

7.3. SYCL

SYCL is a high-level, standardized programming model designed to enable cross-platform execution on various accelerators, including CPUs, GPUs, and FPGAs. Its primary goal is to provide a single codebase that can run efficiently across different hardware architectures. To evaluate its performance, we compiled SYCL binaries targeting CPUs, CUDA, and ROCm, allowing for a direct comparison with their respective low-level implementations. The corresponding SYCL-based Mandelbrot implementation is shown in Listing 10.

SYCL utilizes work queues for task scheduling. Execution begins by creating a queue with a device selector, followed by submitting compute tasks—such as the Mandelbrot kernel—to the queue. SYCL automatically manages data transfers between host and device memory, eliminating the need for explicit driver code.

Listing 10: SYCL double precision kernel

```
queue q(default_selector_v);
{
    // Create a buffer over the image array.
    buffer<uint16_t, 1> image_buf(image, range<1>(HEIGHT * WIDTH));
    // Submit the kernel to the queue.
    q.submit([&](handler &h) {
        // Get write access to the buffer.
        auto image_acc = image_buf.get_access<access::mode::write>(h);
        // Launch a 2D kernel for each pixel.
        h.parallel_for(range<2>(HEIGHT, WIDTH), [=](id<2> idx) {
            size_t row = idx[0];
            size_t col = idx[1];
            // Map pixel coordinate to a point in the complex plane.
            // Using macros from mandelbrot.hpp for MIN_X, STEP, etc.
            double cr = (double) col * STEP + MIN_X;
            double ci = (double) row * STEP + MIN_Y;
            uint16_t iter = 0;
            double zr = 0.0f, zi = 0.0f;
            // Compute Mandelbrot iterations: z = z^2 + c.
            for (; iter <= ITERATIONS; ++iter) {
                double new_zr = zr * zr - zi * zi + cr;
                double new_zi = 2.0f * zr * zi + ci;
                zr = new_zr;
                zi = new_zi;
                if (zr * zr + zi * zi >= 4.0f)
                    break;
            }
            // Write the iteration count for the current pixel.
```

```
                image_acc[row * WIDTH + col] = iter;
            });
        });
    }
```

A key advantage of SYCL is its ability to compile a single codebase for multiple GPU architectures, enabling flexible deployment across different hardware backends. Additionally, its programming model is intuitive for developers already experienced in GPU programming, making it a practical choice for heterogeneous computing environments.

8. Results & Analysis

In this section, we analyze the performance of our implementations under various configurations. We begin by evaluating the baseline performance of the raw implementations before systematically examining the effects of optimization levels, floating-point models, core scaling, and AVX2 vectorization. Additionally, we analyze the performance of Mandelbrot implementations on GPU accelerators, comparing their efficiency to CPU-based approaches.

8.1. CPU

We discussed global compiler options in section 4. In addition, we use the following options for our CPU implementations:

- `-qopenmp`
- `-qopt-report=3`
- `-fsave-optimization-record=yaml`
- `-qopt-report-file=TARGET_NAME.txt`
- `-foptimization-record-file=TARGET_NAME.yml`

These options enable automatically generated compiler reports that aid in finding performance bottlenecks.

Each CPU implementation is executed with optimization levels O0 (no compiler optimization) through O3. For each optimization level, we perform three variations: plain compilation, compilation with `-fp-model=precise`, and compilation with `-fp-model=strict`, allowing us to evaluate the impact of floating point precision settings on performance.

8.1.1. Performance of unoptimized baseline implementation

To assess the impact of our optimizations, we first establish a baseline performance measurement using the unoptimized single-threaded implementation. Running

with single precision (`float`), the Mandelbrot set computation completes in approximately 22.2 seconds.

Since the baseline implementation lacks any form of parallelization, performance remains essentially unchanged when executed across multiple threads. While additional threads are available, they are not utilized, as the implementation does not include explicit multithreading or vectorization. This highlights the need for parallelization strategies to fully leverage modern multi-core processors.

This baseline measurement serves as a reference point for evaluating the effectiveness of subsequent optimizations, including multithreading, SIMD vectorization and custom complex number implementation.

8.1.2. Speedup and Efficiency

Figure 6 shows the speedup of the different CPU implementations. The first and last columns show the single-threaded and 32-threaded performance of each implementation compared to the baseline, respectively. All other columns show their multi-threaded performance in comparison to their own single-threaded performance, which should ideally increase linearly. In a similar way, figure 7 shows efficiency, calculated as

$$efficiency = \frac{speedup}{num\_cores}.$$

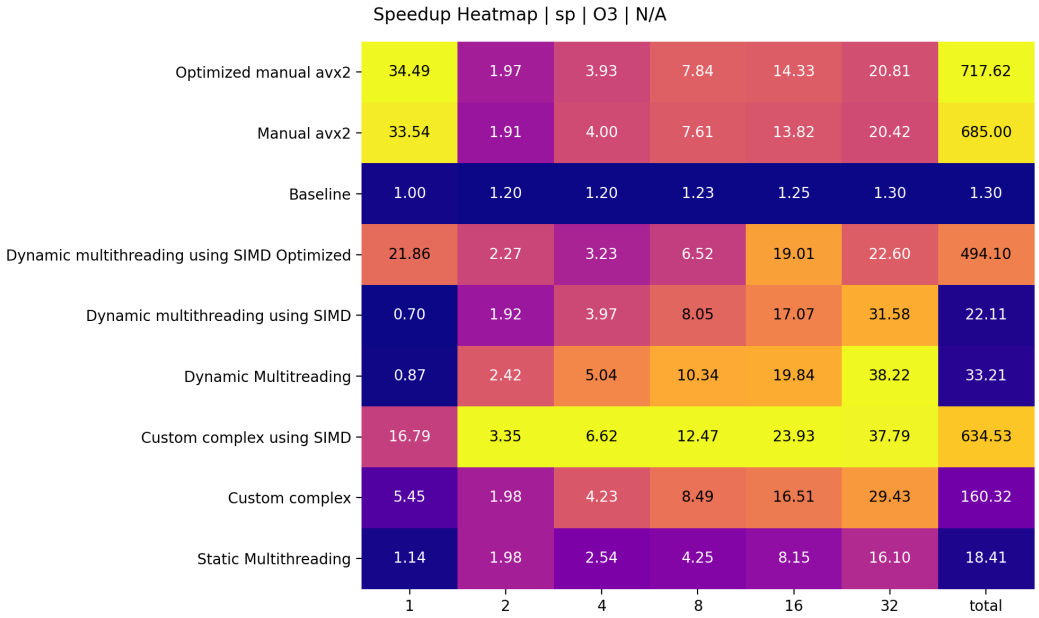


Figure 6: Performance speedup of different implementations

As highlighted in previous sections, manual `avx2` and custom complex number implementations using SIMD drastically outperform all other implementations. Interestingly, the latter shows a speedup greater than the increase in the num-

ber of threads, which suggests comparatively poor single-threaded performance but strong scaling capabilities with little multi-threading overhead. This becomes especially evident, when looking at the CPU efficiency of this implementation.

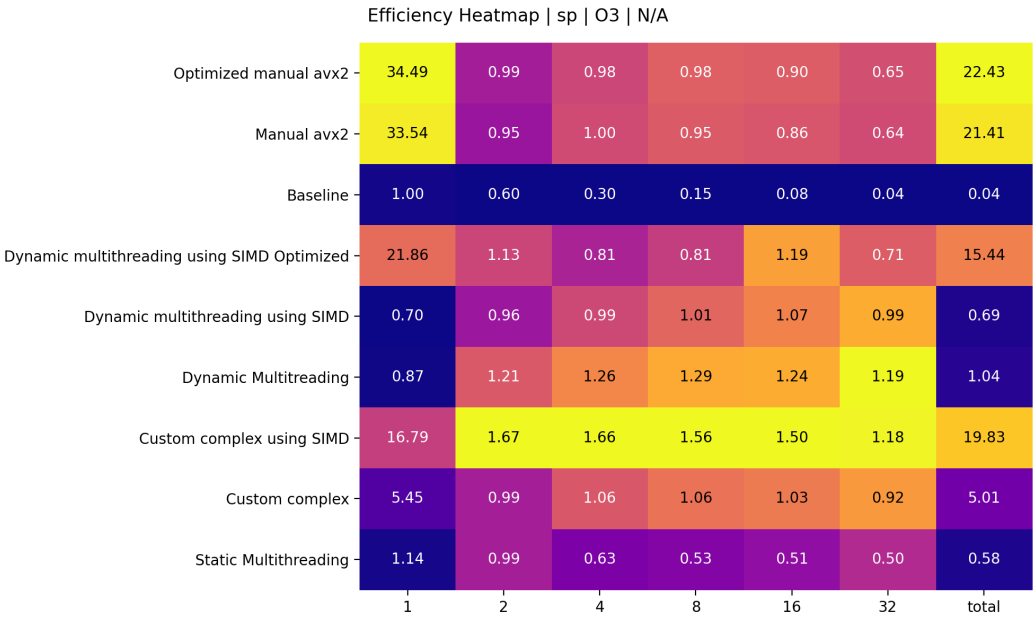


Figure 7: CPU efficiency of different implementations

8.1.3. Impact of Optimization Levels

To analyze the impact of compiler optimizations on performance, we ran our implementations with the different optimization levels `-O0` through `-O3` while keeping all other parameters fixed. The results are shown in figure 8.

Our analysis reveals two key observations:

1. `-O1` and `-O2` outperform `-O3` in almost all cases. This may be due to `-O3` enabling aggressive loop unrolling and inlining, which can sometimes lead to poorer cache utilization or increased register pressure, offsetting potential gains.

2. The optimized version of the dynamic multithreading implementation using SIMD benefits the most from compiler optimizations, achieving up to a 10× speedup compared to all other implementations. This suggests that vectorization and multi-threading provide the largest performance gains, making efficient compiler optimization crucial for high-performance computations.

8.1.4. Effect of Floating-Point Models

We evaluated the impact of floating-point precision and different floating-point models on performance. Each implementation was tested using single precision (SP,

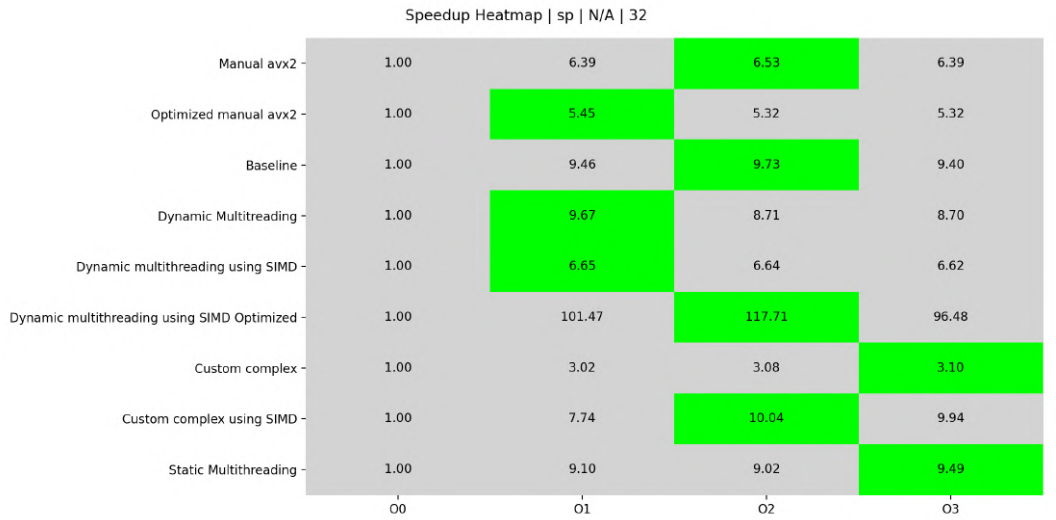


Figure 8: Speedup from compiler optimization

float) and double precision (DP, double), with the speedup from DP to SP shown in Figure 9. Additionally, we tested three floating-point models:

- strict – Enforces IEEE compliance and prevents optimizations that could alter precision.
- precise – Allows some optimizations while maintaining reasonable precision.
- No model specified – Leaves optimization choices to the compiler.

The performance impact of different floating-point models is shown in Figure 10. Our key observations are:

1. Manual AVX2, optimized AVX2, and custom SIMD-based complex number implementations show significant speedups (2×) when using SP, indicating that these implementations fully utilize SIMD efficiency and reduced memory bandwidth in lower precision.
2. Dynamic multithreading, and especially dynamic multithreading with SIMD, perform worse in SP than in DP. This suggests that threading overhead and memory access patterns outweigh computational savings in lower precision.
3. The remaining implementations show no significant differences between SP and DP, likely due to memory-bound behavior or insufficient vectorization.
4. Floating-point model selection mainly impacts manual AVX2 vectorization:
   - `precise` is much faster than `strict` for manual AVX2.
   - Optimized AVX2 suffers a significant slowdown when using `precise`, suggesting that certain auto-vectorization strategies are more sensitive to precision constraints.

These results highlight that floating-point settings can have a significant effect on performance, particularly in manually optimized implementations.

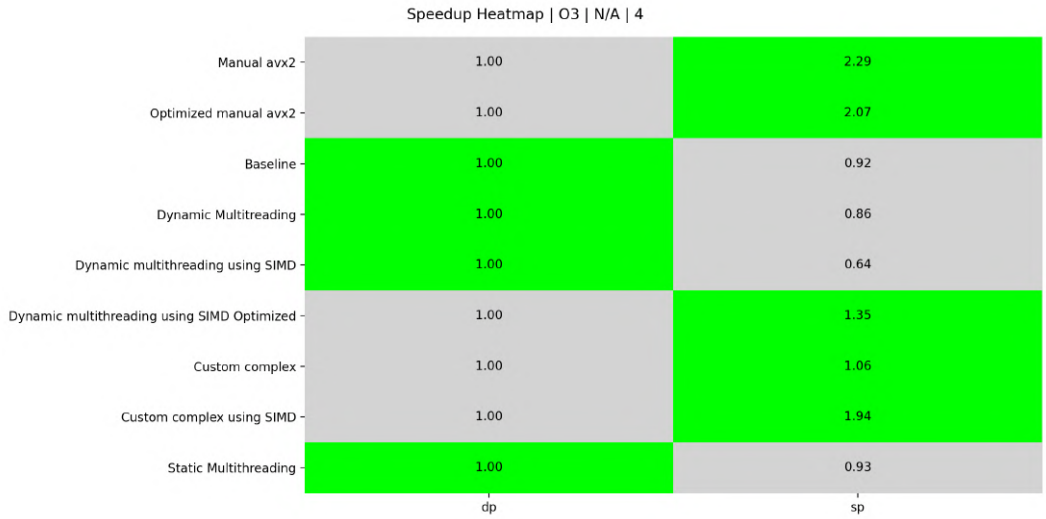


Figure 9: Speedup double precision vs single precision

8.2. GPU

We discussed our available GPU hardware in section 4. According to performance data found on the website Tech Powerup, they have the following compute capabilities shown in table 1. Both NVIDIA cards are from the Turing architecture and have a double precision performance that is $\frac{1}{32}$ of the single precision performance. For the AMD card the double precision performance is $\frac{1}{16}$ of the single precision performance. The big performance penalty for double precision is because all cards are consumer cards. Accelerators specialized for high performance computing have a higher double precision performance. But for typical consumer applications, double precision performance is less important.

	FP32	FP64
RTX 2060	6.451 TFLOPS	201.6 GFLOPS
Tesla T4	8.141 TFLOPS	254.4 GFLOPS
AMD 780m	8.294 TFLOPS	518.4 GFLOPS

Table 1: Performance data for GPU hardware

For the GPUs we ran benchmarks with different work sizes. Specifically, we ran the same parameters as for the CPU benchmarks as well as a run with parameters

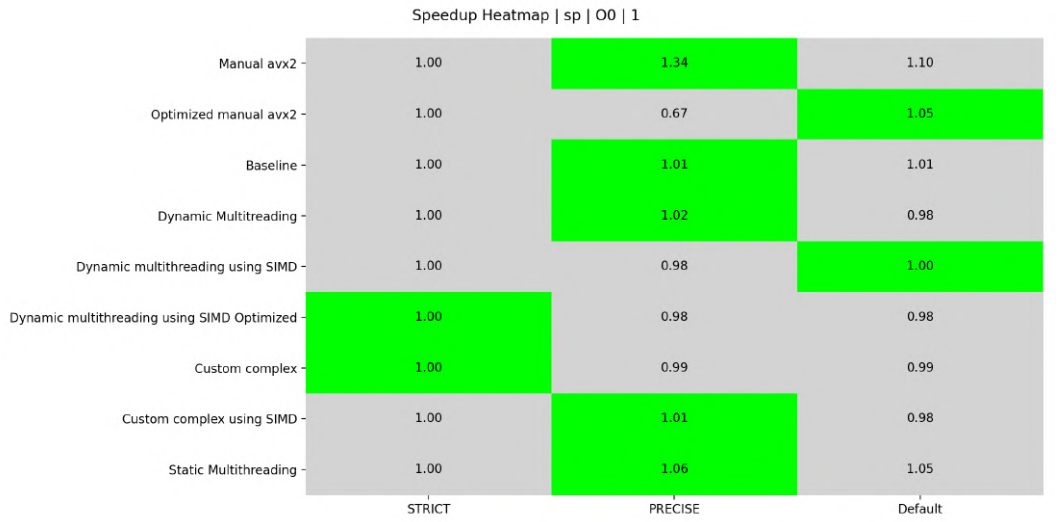


Figure 10: Speed up different floating point models

drastically increased. In figure 11 we see all the results we collected for the GPUs. For reference, results of the avx2 optimized implementation using 32 threads and optimization level O3 are included.

The graphics show, that for the resolution 1000 and Iterations 1000 case the avx2 implementation with 32 threads outperforms all other solutions. This is because the initialization overhead for the GPUs is large, regardless of the work size. For the bigger workload and single precision most implementations are faster than the avx2 version. In fact, only SYCL CPU and SYCL CUDA are slower. For the double precision the avx2 variant once again outperforms all other implementations.

Although the AMD 780m offers in theory a higher compute performance, it performs worse than the CUDA hardware. This matches with real world benchmarks that were done with the AMD 780m. The theoretical performance limits can not be reached because the GPU is limited by the memory speed. Compared to the other cards, this card does not have dedicated graphics memory. Instead it uses part of the normal system memory as graphics memory.

The double precision vs. single precision performance of the NVIDIA hardware approximately matches the expected 32x slowdown. For the ROCm hardware, the slowdown is only 9.48 instead of 16 as expected.

9. Conclusion

Our experiments demonstrate that both modern CPUs and GPUs offer powerful computational capabilities, each excelling in different scenarios. Modern compilers are highly effective at automatically vectorizing code, enabling maintainable and readable implementations without significant performance trade-offs. However, achieving optimal performance requires adhering to best practices and analyzing

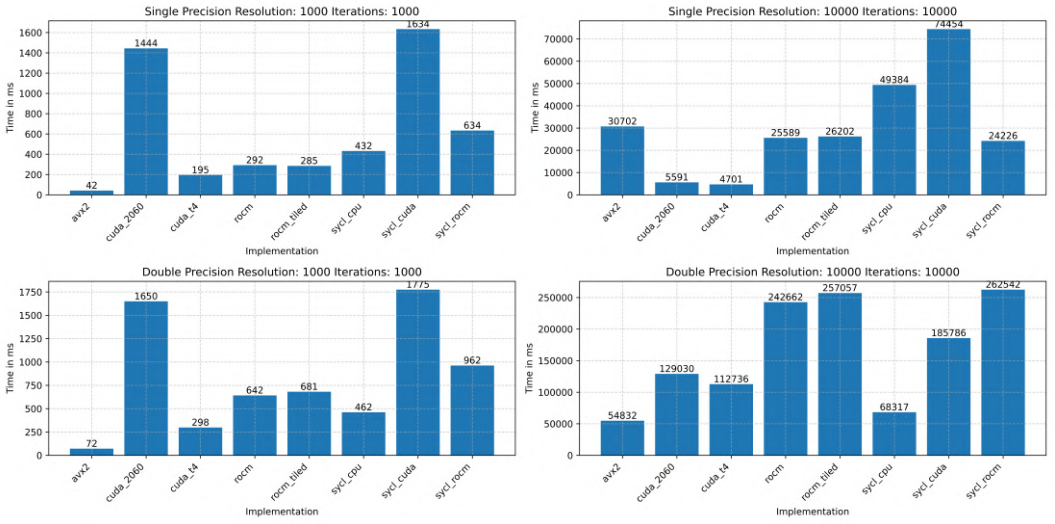


Figure 11: Performance of GPU implementations in comparison to fastest CPU implementation

compiler-generated reports to identify potential obstacles to vectorization. When maintainability is not the primary concern, manual vectorization using intrinsics can further improve performance in certain cases.

On the other hand, GPUs are highly specialized hardware, delivering exceptional speedups for workloads that benefit from massively parallel execution. Their effectiveness is particularly pronounced for large-scale computations, where the initialization overhead is outweighed by the significant performance gains. However, our experiments show that consumer-grade GPUs are primarily suited for single-precision workloads, as double-precision computations experience a substantial performance degradation.

By leveraging both efficient CPU vectorization and GPU acceleration, computational performance can be maximized, depending on the specific workload characteristics and hardware constraints.

A. Appendix

Listing 11: Report that compares results of all variants

```
Difference baseline_dp baseline_sp: 1749999 0.2916665
Group 8c0c113b1b4cfd29ddc60ed7001236ed1f0cd56b7888577ec10c12a3f5066999 0:
        avx2_dp_O0
        avx2_dp_O0-PRECISE
        avx2_dp_O0-STRICT
        avx2_dp_O1-PRECISE
        avx2_dp_O1-STRICT
        avx2_dp_O2-PRECISE
        avx2_dp_O2-STRICT
        avx2_dp_O3-PRECISE
        avx2_dp_O3-STRICT
        avx2_dp_optimized_O0
        avx2_dp_optimized_O0-PRECISE
        avx2_dp_optimized_O0-STRICT
        avx2_dp_optimized_O1-PRECISE
        avx2_dp_optimized_O1-STRICT
        avx2_dp_optimized_O2-PRECISE
        avx2_dp_optimized_O2-STRICT
        avx2_dp_optimized_O3-PRECISE
        avx2_dp_optimized_O3-STRICT
        baseline_dp_O0
        baseline_dp_O0-STRICT
        baseline_dp_O1-STRICT
        baseline_dp_O2-STRICT
        baseline_dp_O3-STRICT
        dynamic_multithreading_dp_O0
        dynamic_multithreading_dp_O0-STRICT
        dynamic_multithreading_dp_O1-STRICT
        dynamic_multithreading_dp_O2-STRICT
        dynamic_multithreading_dp_O3-STRICT
        dynamic_multithreading_dp_simd_O1-STRICT
        dynamic_multithreading_dp_simd_O2-STRICT
        dynamic_multithreading_dp_simd_O3-STRICT
        dynamic_multithreading_simd_optimized_dp_O1-STRICT
        dynamic_multithreading_simd_optimized_dp_O2-STRICT
        dynamic_multithreading_simd_optimized_dp_O3-STRICT
        no_complex_dp_O0
        no_complex_dp_O0-STRICT
        no_complex_dp_O1-STRICT
        no_complex_dp_O2-STRICT
        no_complex_dp_O3-STRICT
        no_complex_dp_simd_O0-STRICT
        no_complex_dp_simd_O1-STRICT
        no_complex_dp_simd_O2-STRICT
        no_complex_dp_simd_O3-STRICT
        static_multithreading_dp_O0
```

```
        static_multithreading_dp_O0-STRICT
        static_multithreading_dp_O1-STRICT
        static_multithreading_dp_O2-STRICT
        static_multithreading_dp_O3-STRICT
        Difference Sum: 0 0.0
Group c119c47712478c5bd413616f6370dafcbde13e7be878a4124385b0acf27fda69 1:
        avx2_sp_O0
        avx2_sp_O0-PRECISE
        avx2_sp_O0-STRICT
        avx2_sp_O1-PRECISE
        avx2_sp_O1-STRICT
        avx2_sp_O2-PRECISE
        avx2_sp_O2-STRICT
        avx2_sp_O3-PRECISE
        avx2_sp_O3-STRICT
        avx2_sp_optimized_O0
        avx2_sp_optimized_O0-PRECISE
        avx2_sp_optimized_O0-STRICT
        avx2_sp_optimized_O1-PRECISE
        avx2_sp_optimized_O1-STRICT
        avx2_sp_optimized_O2-PRECISE
        avx2_sp_optimized_O2-STRICT
        avx2_sp_optimized_O3-PRECISE
        avx2_sp_optimized_O3-STRICT
        dynamic_multithreading_simd_optimized_sp_O1-STRICT
        dynamic_multithreading_simd_optimized_sp_O2-STRICT
        dynamic_multithreading_simd_optimized_sp_O3-STRICT
        no_complex_sp_simd_O0-STRICT
        no_complex_sp_simd_O1-STRICT
        no_complex_sp_simd_O2-STRICT
        no_complex_sp_simd_O3-STRICT
        Difference Sum: 2256806 0.37613433333333335
Group f1bbe963f246678b0b2aee285314b5a0ed322ddf1a312c5131ae19771f1d21dd 2:
        baseline_sp_O0
        baseline_sp_O0-STRICT
        baseline_sp_O1-STRICT
        baseline_sp_O2-STRICT
        baseline_sp_O3-STRICT
        dynamic_multithreading_sp_O0
        dynamic_multithreading_sp_O0-STRICT
        dynamic_multithreading_sp_O1-STRICT
        dynamic_multithreading_sp_O2-STRICT
        dynamic_multithreading_sp_O3-STRICT
        dynamic_multithreading_sp_simd_O1-STRICT
        dynamic_multithreading_sp_simd_O2-STRICT
        dynamic_multithreading_sp_simd_O3-STRICT
        static_multithreading_sp_O0
        static_multithreading_sp_O0-STRICT
        static_multithreading_sp_O1-STRICT
        static_multithreading_sp_O2-STRICT
        static_multithreading_sp_O3-STRICT
```

```
        Difference Sum: 0 0.0
Group 4ac6e5664794ae3b6a86614a85f64f027e333b7d62d4041849698f5f59fb01c7 3:
        baseline_sp_O0-PRECISE
        baseline_sp_O1-PRECISE
        baseline_sp_O2-PRECISE
        baseline_sp_O3-PRECISE
        dynamic_multithreading_sp_O0-PRECISE
        dynamic_multithreading_sp_O1-PRECISE
        dynamic_multithreading_sp_O2-PRECISE
        dynamic_multithreading_sp_O3-PRECISE
        dynamic_multithreading_sp_simd_O1
        dynamic_multithreading_sp_simd_O1-PRECISE
        dynamic_multithreading_sp_simd_O2
        dynamic_multithreading_sp_simd_O2-PRECISE
        dynamic_multithreading_sp_simd_O3
        dynamic_multithreading_sp_simd_O3-PRECISE
        static_multithreading_sp_O0-PRECISE
        static_multithreading_sp_O1-PRECISE
        static_multithreading_sp_O2-PRECISE
        static_multithreading_sp_O3-PRECISE
        Difference Sum: 468352 0.07805866666666667
Group 34497dc52b1c56de53b3305a9cc20d56a9ebcc73e2eafbdce61441203f71371a 4:
        baseline_dp_O0-PRECISE
        baseline_dp_O1-PRECISE
        baseline_dp_O2-PRECISE
        baseline_dp_O3-PRECISE
        dynamic_multithreading_dp_O0-PRECISE
        dynamic_multithreading_dp_O1-PRECISE
        dynamic_multithreading_dp_O2-PRECISE
        dynamic_multithreading_dp_O3-PRECISE
        dynamic_multithreading_dp_simd_O1-PRECISE
        dynamic_multithreading_dp_simd_O2-PRECISE
        dynamic_multithreading_dp_simd_O3-PRECISE
        static_multithreading_dp_O0-PRECISE
        static_multithreading_dp_O1-PRECISE
        static_multithreading_dp_O2-PRECISE
        static_multithreading_dp_O3-PRECISE
        Difference Sum: 502401 0.0837335
Group ece713ff7f6a7fd1d1a03c0d31fa895ec2b59f008be58954a2596fbf74384a6f 5:
        baseline_dp_O1
        baseline_dp_O2
        baseline_dp_O3
        dynamic_multithreading_dp_O1
        dynamic_multithreading_dp_O2
        dynamic_multithreading_dp_O3
        dynamic_multithreading_dp_simd_O1
        dynamic_multithreading_dp_simd_O2
        dynamic_multithreading_dp_simd_O3
        static_multithreading_dp_O1
        static_multithreading_dp_O2
        static_multithreading_dp_O3
```

```
        Difference Sum: 506706 0.084451
Group acf760cd225aa7ba77772e4a62512b16d8022020cf49f428e52055f16fb40249 6:
        avx2_dp_optimized_O1
        avx2_dp_optimized_O2
        avx2_dp_optimized_O3
        no_complex_dp_O0-PRECISE
        no_complex_dp_O1-PRECISE
        no_complex_dp_O2-PRECISE
        no_complex_dp_O3-PRECISE
        no_complex_dp_simd_O0
        Difference Sum: 502400 0.08373333333333334
Group d140232fe2b52157e2ea9358125f254a748c49e338b7f886a6a192db3c2155c5 7:
        dynamic_multithreading_simd_optimized_dp_O1
        dynamic_multithreading_simd_optimized_dp_O2
        dynamic_multithreading_simd_optimized_dp_O3
        no_complex_dp_simd_O1
        no_complex_dp_simd_O2
        no_complex_dp_simd_O3
        Difference Sum: 502979 0.08382983333333334
Group e69309e2067daa588674c9095244b5e61d951b7ce85450161c3c937b68c1ec4b 8:
        dynamic_multithreading_simd_optimized_sp_O1
        dynamic_multithreading_simd_optimized_sp_O2
        no_complex_sp_simd_O0-PRECISE
        no_complex_sp_simd_O1-PRECISE
        no_complex_sp_simd_O2-PRECISE
        no_complex_sp_simd_O3-PRECISE
        Difference Sum: 2846435 0.47440583333333336
Group 34218b654a494d257f3e08888b10c3647291bd8692a37035ac8df7890245299c 9:
        no_complex_sp_O0
        no_complex_sp_O0-STRICT
        no_complex_sp_O1-STRICT
        no_complex_sp_O2-STRICT
        no_complex_sp_O3-STRICT
        Difference Sum: 2 3.3333333333333335e-07
Group f9ef676f04ec45b5af960bdbde10f564b5db640519d5a70c54f88909344ec89a 10:
        avx2_sp_optimized_O1
        avx2_sp_optimized_O2
        avx2_sp_optimized_O3
        no_complex_sp_simd_O0
        Difference Sum: 2840438 0.4734063333333333
Group 7ef262ace1be309d1da577bd4aac787d8cd70ccdb987cbfb39b1e54081164444 11:
        baseline_sp_O2
        baseline_sp_O3
        static_multithreading_sp_O2
        static_multithreading_sp_O3
        Difference Sum: 2121155 0.3535258333333333
Group cf407dd227cf307d59402c48ff3f5ff9008c20a4c439daa76324d6074a481ef2 12:
        dynamic_multithreading_simd_optimized_sp_O3
        no_complex_sp_simd_O1
        no_complex_sp_simd_O2
        no_complex_sp_simd_O3
```

```
        Difference Sum: 2841239 0.4735398333333333
Group c0d14b2403d7eb311bab2321c9b8759b27af2282f006b7b22e62a7bbaa20f845 13:
        no_complex_dp_simd_O0-PRECISE
        no_complex_dp_simd_O1-PRECISE
        no_complex_dp_simd_O2-PRECISE
        no_complex_dp_simd_O3-PRECISE
        Difference Sum: 504270 0.084045
Group 91f127a5b58259843e1325d381bb98c0acbbc154a52fac8e81a57b25fbf20df4 14:
        no_complex_sp_O0-PRECISE
        no_complex_sp_O1-PRECISE
        no_complex_sp_O2-PRECISE
        no_complex_sp_O3-PRECISE
        Difference Sum: 468354 0.078059
Group 86fc000af48f5ad8b9d7a0e6219aeedcb6aa95c4a5b0174bb05347724f56cc27 15:
        baseline_sp_O1
        dynamic_multithreading_sp_O1
        static_multithreading_sp_O1
        Difference Sum: 2046648 0.341108
Group d67696fce1dbfe57ace7acc42a1697270ee976affa6065d44183556b95991d07 16:
        dynamic_multithreading_simd_optimized_dp_O1-PRECISE
        dynamic_multithreading_simd_optimized_dp_O2-PRECISE
        dynamic_multithreading_simd_optimized_dp_O3-PRECISE
        Difference Sum: 502402 0.08373366666666666
Group e22ff07127aa1774ad82ca7c452f3d7d6d495fe27bb7f94e0729d50d8ca7cf7e 17:
        dynamic_multithreading_simd_optimized_sp_O1-PRECISE
        dynamic_multithreading_simd_optimized_sp_O2-PRECISE
        dynamic_multithreading_simd_optimized_sp_O3-PRECISE
        Difference Sum: 2840438 0.4734063333333333
Group 11aded665b15855a2f572f298c099c3076b39ac0d737d4a3b9dd0f187ca65bd9 18:
        avx2_dp_O2
        avx2_dp_O3
        Difference Sum: 505756 0.08429266666666667
Group 0aa22c50fd51e571cb97b12b5fdbdcbb9e2213eb965102758acd05dddb7cb798 19:
        avx2_sp_O2
        avx2_sp_O3
        Difference Sum: 2846742 0.474457
Group 804a3416bf8dbc7b03f8b8dde15cf709a6459b1f5066e8ff7752c0afa8c95b6f 20:
        dynamic_multithreading_dp_simd_O0
        dynamic_multithreading_dp_simd_O0-PRECISE
        Difference Sum: 9706901 1.6178168333333334
Group ebe206e7f6adab3ffcdbe80f833b26513bb683ec1ff05e6896b594493670e4ff 21:
        dynamic_multithreading_sp_O2
        dynamic_multithreading_sp_O3
        Difference Sum: 1821436 0.30357266666666666
Group d58071d0d67678d7ca365a886b4d014f847b86fb902ea0973d8a0ecdc6922b13 22:
        dynamic_multithreading_sp_simd_O0
        dynamic_multithreading_sp_simd_O0-PRECISE
        Difference Sum: 9668744 1.6114573333333333
Group 864045e241faa3dbdfbc61ee2ceaae5f20f3c8eda2290dbdb23985bcce2f7d97 23:
        no_complex_dp_O2
        no_complex_dp_O3
```

```
        Difference Sum: 505252 0.08420866666666667
Group 01a3ac91d6647d319bd1fe41fe4360bf0e72abecd7225edf89f5c3bec513b096 24:
        no_complex_sp_O2
        no_complex_sp_O3
        Difference Sum: 1998801 0.3331335
Group 9b173967b3109b975df40af467bc9e468043b88e1d1a09113e17c46f34ed2e30 25:
        rocm_sp
        sycl_sp_rocm
        Difference Sum: 4359699 0.7266165
Group 5ae542507755e6125d66403f0ccf5ec3bbb4271213f7c7a0a392f531734e6a26 26:
        avx2_dp_O1
        Difference Sum: 506707 0.08445116666666666
Group a1adeed90ea23576eff0142bf0ccb76d76f775dfe2fb9ab1e5227e84c0c035ed 27:
        avx2_sp_O1
        Difference Sum: 2849178 0.474863
Group af9f8de0af809e7f34105bbc570a002bc664f38628e5ce982ce56d111302aad5 28:
        cuda_dp
        Difference Sum: 2014263 0.3357105
Group 9d948a4ead055ec7a07dc0ea1792de083d7435aa24f1f644843ea6a45c5464d2 29:
        cuda_sp
        Difference Sum: 3478355 0.5797258333333334
Group 7559e5cae13cc651ccf12218b7a596f583620fb4148edbd7124bc78c64664648 30:
        dynamic_multithreading_dp_simd_O0-STRICT
        Difference Sum: 9219421 1.5365701666666667
Group 53a624e6225e7aae8db9fef9251362c70e05435bf46e05a8db2e5cd7251d0881 31:
        dynamic_multithreading_simd_optimized_dp_O0
        Difference Sum: 9705610 1.6176016666666666
Group 14820c52555072a7e4e60039d651fcf28638ad03390d41b00dbb460f65396ca5 32:
        dynamic_multithreading_simd_optimized_dp_O0-PRECISE
        Difference Sum: 9705612 1.617602
Group b7742a0215ce9588dc3de71a98df2200e434f1cca7854662f46afc1e0ca37aff 33:
        dynamic_multithreading_simd_optimized_dp_O0-STRICT
        Difference Sum: 9217186 1.5361976666666666
Group 05390e7bb12fbc02cf98ba64d8713963651e9e68923f4d855aff7f33edbd0dc9 34:
        dynamic_multithreading_simd_optimized_sp_O0
        Difference Sum: 10877664 1.812944
Group 2ef518184e16a374f11ec7414663cfd9e693bea03ce5c34a87f0f4cbbb8fed02 35:
        dynamic_multithreading_simd_optimized_sp_O0-PRECISE
        Difference Sum: 10877664 1.812944
Group 91357817c129dd4024f06d50078d72d78f45fe6946b4552bb8aeb0fbca5325cb 36:
        dynamic_multithreading_simd_optimized_sp_O0-STRICT
        Difference Sum: 10349446 1.7249076666666667
Group a3cbe1761c8ea1094f6368d9e42d9e1ec8d2bbea902442f551c02f30446adcf2 37:
        dynamic_multithreading_sp_simd_O0-STRICT
        Difference Sum: 9200582 1.5334303333333332
Group 13bbd3b50c5cdfa5793350cc0a6406cc789a110bcd0df5697104242a44544d2f 38:
        no_complex_dp_O1
        Difference Sum: 505001 0.08416683333333333
Group 0b4f5e19d566717680bed8c220cdc37c2aaef957072f96b5ef35c5c7f604d8c7 39:
        no_complex_sp_O1
        Difference Sum: 2146447 0.35774116666666667
```

```
Group 29a568125edfe32af406a9128feda568251c1add643765a38d4235502b6b10ea 40:
        rocm_dp
        Difference Sum: 2014265 0.33571083333333335
Group b5dd5ded21ace5aa5da077015581629f4224cffa09c9e471dd0c3744f2e77f4e 41:
        rocm_dp_tiled
        Difference Sum: 505654 0.08427566666666667
Group b7863216c3fccfa106e5ecd2fec0dd939677500f0887f0559b40359fcf78fd58 42:
        rocm_sp_tiled
        Difference Sum: 2842545 0.4737575
Group 8aca4b035573f26051bf9529d652fbe9c21d871ab1de6afa63e8fcfc1cff5947 43:
        sycl_dp_cuda
        Difference Sum: 2015087 0.33584783333333335
Group 8b104f7cab6286fbc53484d918bace6b7fb1038537155a7641fd98ca039b4b93 44:
        sycl_dp_rocm
        Difference Sum: 6505990 1.0843316666666667
Group 3d0a7ed53e2829d4f200ac35747ba5f592a2e0e14ea31e5448f164c9071bcf5e 45:
        sycl_sp_cuda
        Difference Sum: 4313558 0.7189263333333333

Total groups: 46 sp: 25 dp: 21
```