DV1674 - Performance Optimization - Assignment 2

Daniele Gagni, Riccardo Storchi

October 31, 2024

Blekinge Tekniska Högskola

Teachers: Suejb Memeti, Marcus Kicklighter

1 Introduction

In this assignment, we were tasked with measuring and optimizing the performance of two algorithms: the first one computed the Pearson correlation coefficient on a given dataset, the second applied gaussian blur to a `.ppm` image. For the Pearson algorithm, we had 4 datasets of different sizes (`128.data, 256.data, 512.data, 1024.data`) available for testing. Same for the blur algorithm, with 4 different images (`im1.ppm, im2.ppm, im3.ppm, im4.ppm`).All code and collected metrics are available in this repository.

1.1 Methodology

- in order to gather the **execution time**, we employed a bash script that run the `time` command for a specific binary (`blur / pearson`) 5 times and then computed the average.
- to obtain the `%CPU` and `%MEM` metrics we used `pidstat`
- in order to generate the **flamegraphs**, we used `perf record` to gather data and the `hotspot` GUI to visualize them.
- the `speedup` is computed for every optimization according to the following formula:

$$\text{Speedup} = \frac{T_{\text{baseline}}}{T_{\text{currentOpt}}} \tag{1}$$

- all the performance data included in this report refers to the execution of `pearson` and `blur` with `1024.data` and `im4.ppm` as input file, respectively.

1.2 System Under Test (SUT)

CPU Information

- Number of cores: `4`
- Number of threads: `8`
- Clock speed: `100MHz`
- Cache size: `8 MB`

Memory (RAM)

- Type: `DDR4`
- Total available memory: `16 GB`
- Memory speed: `2400 MT/s`

SSD Information

- Device size: `512 GB`
- Interface: `SATA 3`
- Max throughput (theoretical): `600 MB/s`

OS and Kernel

- OS: `Arch Linux x86_64`
- Kernel: `6.11.1-arch1-1`

2 Pearson

2.1 Baseline performance

Average Execution Time	32.843s
Average %CPU	12.51%
Average %MEM	0.14%

2.2 Optimization 1: `-O3` g++ flag

The `-O3` compiler flag in C++ enables the highest level of optimization by applying techniques such as function inlining, loop unrolling, vectorization and others, with the goal to to maximize the execution speed. The detailed list of all the optimization flags enabled by `-O3` can be found at the following link.

2.2.1 Performance Measurement

Average Execution Time	6.448s
Average %CPU	12.48%
Average %MEM	0.14%
Speedup	**5.09x**

2.3 Optimization 2: additional loop unrolling in `vector.cpp` functions

By looking at the flamegraphs in figure 1b we noticed that the main bottlenecks could be found in the functions `Vector::operator/()`, `Vector::operator-()` and `Vector::dot()` contained in the `vector.cpp` class. For this reason, we decided to implement additional loop unrolling for these 3 functions. An example of the loop unrolling implementation is provided in the following snippet.

```
double Vector::dot(const Vector& rhs) const
{
    double result{0};
    unsigned i = 0;

    // Loop unrolling for dot product
    for (; i + 4 <= size; i += 4) {
        result += data[i] * rhs.data[i];
        result += data[i + 1] * rhs.data[i + 1];
        result += data[i + 2] * rhs.data[i + 2];
        result += data[i + 3] * rhs.data[i + 3];
    }
    // Handle remaining elements
    for (; i < size; ++i) {
        result += data[i] * rhs.data[i];
    }

    return result; // Return the computed dot product
}
```

Listing 1: LoopUnrolling

2.3.1 Performance Measurement

Average Execution Time	5.569s
Average %CPU	12.49%
Average %MEM	0.14%
Speedup	**5.88x**

2.4 Optimization 3: pass data by reference

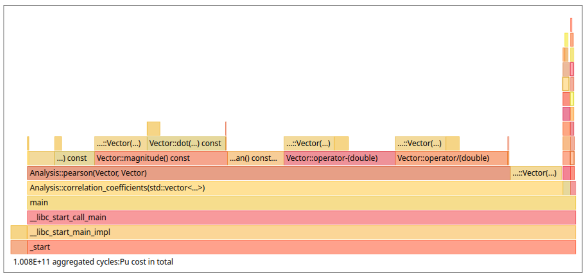
In C++, passing data by reference ensures that a function receives a reference to the original variable, rather than a copy of the variable's value. When computing the Pearson coefficient, the function `pearson()` needs 2 arrays as arguments; these 2 arrays were passed by copy, introducing a significant overhead as this function is called multiple times. By modifying the `correlation_coefficients()` and `pearson()` functions declaration as follows, we ensured that all the data is passed by reference, reducing the copying overhead.

- `std::vector<double> correlation_coefficients(const std::vector<Vector>& datasets)`
- `double pearson(const Vector& vec1, const Vector& vec2)`

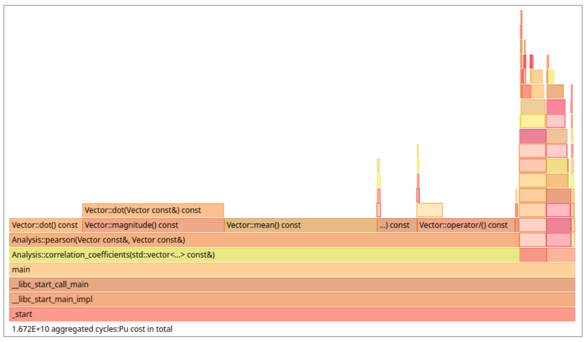
2.4.1 Performance Measurement

Average Execution Time	5.316s
Average %CPU	12.48%
Average %MEM	0.1%
Speedup	**6.18x**

2.5 Flamegraphs comparison



(a) Pearson flamegraph - baseline version



(b) Pearson flamegraph - optimized version

Figure 1: We can see the benefits of our optimizations directly from the flamegraph. In the optimized version, the yellow squares above `Vector::operator/()`, `Vector::mean()` and `Vector::dot()`, representing the overhead introduced by copying, disappeared.

2.6 Parallel Implementation using `pthreads`

Our approach in parallelizing the blur algorithm is structured as follows:

- the dataset is divided into chunks; the chunk size is computed according to the following formula:
  `int chunk_size = dataset_size / num_threads;`
- each chunk is assigned to a thread, which computes the Pearson coefficient for all datasets pairs within its chunk
- then, each thread writes the results to a specific portion of a shared vector
- A struct named `ThreadData`, containing all the relevant information, is passed to each thread

```
        struct ThreadData {
            const std::vector<Vector>* datasets; // dataset vector
            std::vector<double>* result;         // shared result vector
            int start_idx;                    // data starting index for the current thread
            int end_idx;                      // data ending index for the current thread
            int result_start_idx;             // index where the thread should start writing
        };
```

Listing 2: ThreadData

Nr. of threads	Avg. exec time	Speedup
1	5.316s	6.18x
2	4.980s	6.59x
4	3.852s	8.53x
8	3.137s	10.47x
16	**2.989s**	**10.99x**
32	4.066s	8.08x

Figure 2: Pearson - Parallel Performance

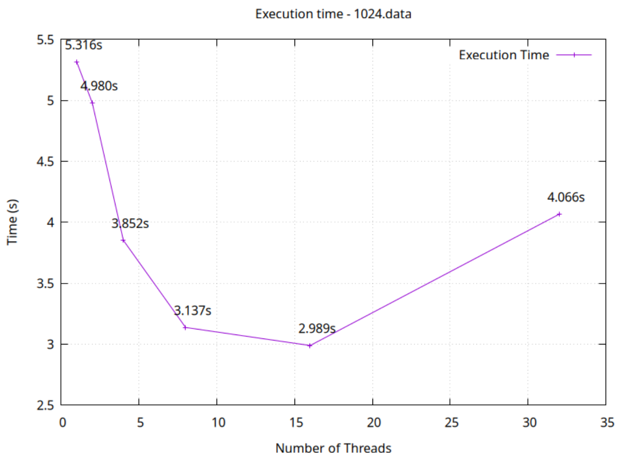


Figure 3: Pearson - Execution Time

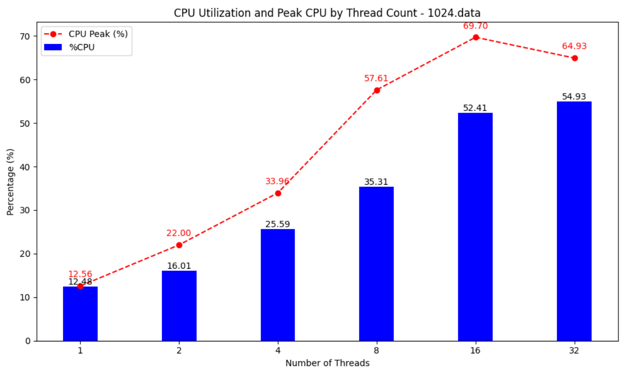


Figure 4: CPU usage

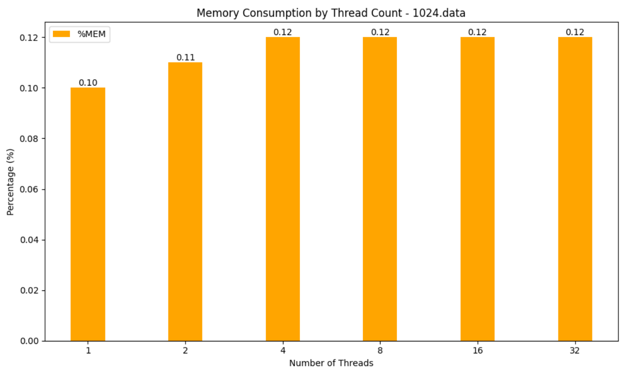


Figure 5: Pearson - Memory usage

2.6.1 Performance Measurements

As we can see, the best speedup is achieved with **16 threads**. This is an unusual result since the SUT has a maximum of 8 logical threads. A possible explanation for this could be that since the task to perform may be very long (computing the pearson coefficient for two datasets of 1024 elements), the time spent actually performing the tasks outweighs the synchronization overhead.

3 Blur

3.1 Baseline Performance

Average Execution Time	18.673s
Average %CPU	12.49%
Average %MEM	0.84%

3.2 Optimization 1: `-O3` g++ flag

As we did for the Pearson code, we added `-O3` g++ flag, enabling the highest level of optimization for the compiler.

3.2.1 Performance Measurement

Average Execution Time	11.922s
Average %CPU	12.48%
Average %MEM	0.84%
Speedup	**1.57x**

3.3 Optimization 2: additional loop unrolling in `Filter::blur()`

We implemented additional loop unrolling on the blur() function so to process 4 pixel in one iteration, both in the horizontal and vertical pass. This way the the outer loop's setup, increment, and termination checks occur only a quarter as frequently. Note that this is note pure loop unrolling, as we added an inner loop to process each of the four pixels. However, its control overhead is minimal because it has a fixed, small number of iterations (up to 4) and benefits from spatial locality, accessing nearby pixel data efficiently. The reduction in outer loop overhead outweighs the small added cost of the inner loop.

```
    /* Horizontal pass */
    for (auto x{0}; x < x_size; x += 4)
    {
        for (auto y{0}; y < y_size; y++)
        {
            double w[Gauss::max_radius]{};
            Gauss::get_weights(radius, w);

            // Process 4 pixels in one iteration if available
            for (int offset = 0; offset < 4 && (x + offset) < x_size; ++offset) {
                auto r = w[0] * dst.r(x + offset, y);
                auto g = w[0] * dst.g(x + offset, y);
                auto b = w[0] * dst.b(x + offset, y);
                auto n = w[0];

                for (auto wi{1}; wi <= radius; wi++)
                {
                    auto wc = w[wi];
                    auto x2 = x + offset - wi;
                    if (x2 >= 0) // left neighbour
                    {
                        r += wc * dst.r(x2, y);
                        g += wc * dst.g(x2, y);
                        b += wc * dst.b(x2, y);
                        n += wc;
                    }
                    x2 = x + offset + wi;
                    if (x2 < x_size) // right neighbour
                    {
                        r += wc * dst.r(x2, y);
                        g += wc * dst.g(x2, y);
                        b += wc * dst.b(x2, y);
                        n += wc;
                    }
                }
                scratch.r(x + offset, y) = r / n;
                scratch.g(x + offset, y) = g / n;
                scratch.b(x + offset, y) = b / n;
            }
        }
    }
```

Listing 3: Partial Loop Unrolling

3.3.1 Performance Measurement

Average Execution Time	7.240s
Average %CPU	12.49%
Average %MEM	0.67%
Speedup	**2.58x**

3.4 Optimization 3: loop peeling in `Filter::blur()`

The blurring algorithm relies on Gaussian weights, which are constant because they are independent of specific pixel values. Therefore, we moved the `get_weights()` function outside of the horizontal and vertical pass loops, allowing us to compute the weights only once instead of recalculating them for every pixel.

```
Matrix blur(Matrix& m, const int radius)
{
    Matrix scratch{PPM::max_dimension};
    auto dst{m}; // dst = destination

    double w[Gauss::max_radius]{};
    // function called outside the loop
    Gauss::get_weights(radius, w);

    auto x_size = dst.get_x_size();
    auto y_size = dst.get_y_size();

    /* Horizontal pass */
    for (auto x{0}; x < x_size; x += 4)
    {
        for (auto y{0}; y < y_size; y++)
        {
            // removed get_weights call from here
        }
    }

    /* Vertical pass */
    for (auto x{0}; x < x_size; x += 4)
    {
        for (auto y{0}; y < y_size; y++)
        {
            // removed get_weights call from here
        }
    }
    return dst;
}
```

Listing 4: Loop Peeling in `Filter::blur()`

3.4.1 Performance Measurement

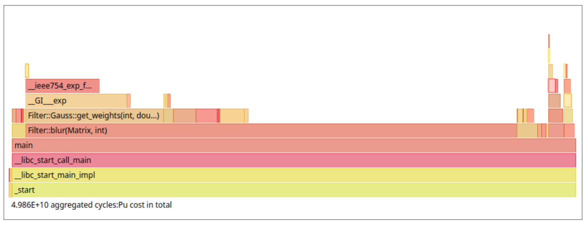
Average Execution Time	4.363s
Average %CPU	12.44%
Average %MEM	0.68%
Speedup	**4.28x**

3.5 Extra: SIMD instructions using `_m256d` instriniscs

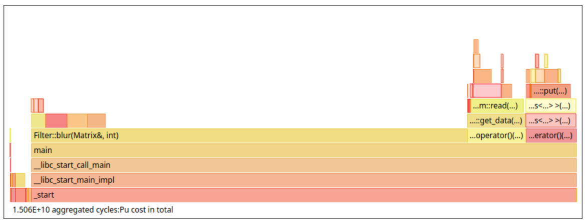
By adding the `-mavx2` flag to `gcc` and using the `_m256d` data type, we enabled AVX registers and exploited SIMD instructions to process four pixels simultaneously (a `_m256d` variable can hold four double values). Although we successfully implemented this change, we ultimately had to discard it because the optimized code failed the `verify.sh` test script. While the blurred image was visually similar to the one generated by the non-SIMD code, slight approximation errors from vectorization led to minor RGB value deviations.

```
    /* Horizontal pass */

    for (auto x{0}; x < dst.get_x_size(); x++)
    {
        int y = 0;
        for (; y <= dst.get_y_size() - 4; y += 4)
        {
            __m256d r = _mm256_setzero_pd();
            __m256d g = _mm256_setzero_pd();
            __m256d b = _mm256_setzero_pd();
            __m256d n = _mm256_setzero_pd();

            __m256d w0 = _mm256_set1_pd(w[0]);
            __m256d r_center = _mm256_set_pd(dst.r(x, y + 3), dst.r(x, y + 2), dst.r(x, y + 1), dst.r(x, y));
            __m256d g_center = _mm256_set_pd(dst.g(x, y + 3), dst.g(x, y + 2), dst.g(x, y + 1), dst.g(x, y));
            __m256d b_center = _mm256_set_pd(dst.b(x, y + 3), dst.b(x, y + 2), dst.b(x, y + 1), dst.b(x, y));

            r = _mm256_add_pd(_mm256_mul_pd(w0, r_center), r);  // r = (w0 * r_center) + r
            g = _mm256_add_pd(_mm256_mul_pd(w0, g_center), g);
            b = _mm256_add_pd(_mm256_mul_pd(w0, b_center), b);
            n = _mm256_add_pd(n, w0);

            // ...
```

3.6 Flamegraphs comparison



(a) Blur flamegraph - baseline version



(b) Blur flame graph - optimized version

Figure 6: Our optimizations are evident in the flamegraph: in the optimized version, the box for the `get_weights` function has disappeared. This shows that the function is no longer a bottleneck, as the number of calls has been dramatically reduced.

3.7 Parallel Implementation

Our approach in parallelizing the blur algorithm is structured as follows:

- The image is divided into chunks; each chunk is processed by a different thread
  - the chunks size is computed accordong to the following formula:
    `int chunk_size = (x_size + num_threads - 1)/ num_threads;`
- **Horizontal Pass**: Each thread applies the horizontal blur to its chunk of the image, and the intermediate results are stored in the scratch matrix
- **Vertical Pass**: After the horizontal pass, the threads apply the vertical blur to the scratch matrix, and the final result is written to the dst matrix
- A struct named `ThreadData`, containing all the relevant information, is passed to each thread

```
        struct ThreadData {
            const Matrix* src; // source matrix
            Matrix* dst;       // destination matrix
            Matrix* scratch;   // intermediate matrix
            int start_x;       // start x coord
            int end_x;         // end x coord
            int y_size;        // image height
            int radius;        // blurring radius
            int x_size;        // image witdh
        };
```

Listing 5: ThreadData

It is important to note that our parallel implementation does not employ the additional loop unrolling of 4 pixels (see section 3.3): we decided to discard this implementation because it made difficult to avoid overlappings among the different chunks of the images that each thread would modify independently.

3.7.1 Performance Measurement

Nr. of threads	Avg. exec time	Speedup
1	5.653s	3.30x
2	3.984s	4.79x
4	3.088s	6.21x
8	**2.980s**	**6.27x**
16	3.320s	5.62x
32	4.296s	4.35x

Figure 7: Blur - Parallel Performance

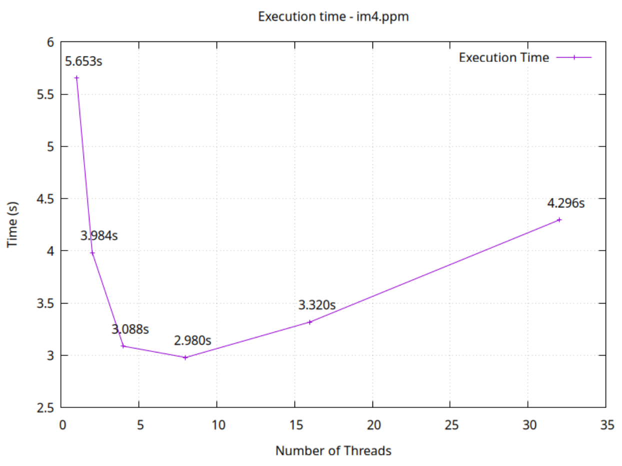


Figure 8: Blur - Execution Time

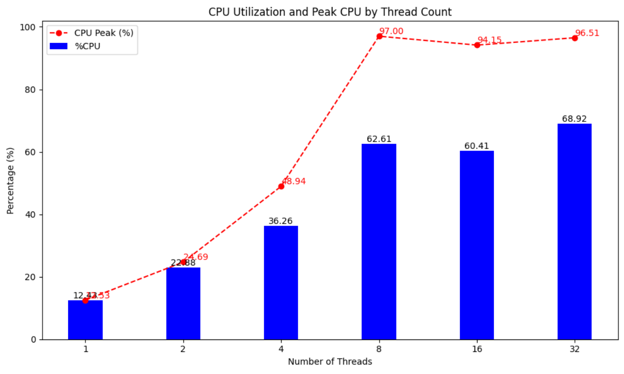


Figure 9: Blur - CPU usage

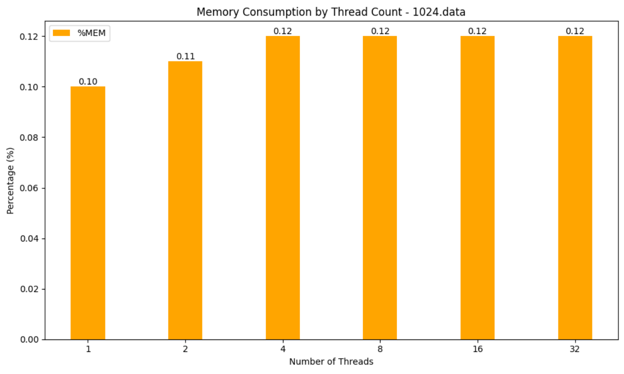


Figure 10: Pearson - Memory usage

The maximum speedup is achieved with 8 threads, which is consistent with the maximum number of logical threads of the SUT.

4 Summary

In this assignment, we measured the performance of the Blur and Pearson algorithms, starting with baseline metrics: Pearson had an average execution time of 32.843s, CPU usage of 12.51%, and memory usage of 0.14%, while Blur had an average execution time of 18.673s, CPU usage of 12.49%, and memory usage of 0.84%. We applied several optimizations to improve performance, beginning with the -O3 compiler flag, establishing new baselines for each algorithm.
For **Pearson**, we implemented additional loop unrolling in vector.cpp and passed data by reference, achieving a speedup to an average execution time of 5.316s, with CPU and memory usage reduced to 12.48% and 0.1%, respectively, resulting in a **6.18x speedup**. We then introduced multithreading using pthreads, yielding further speedups: the best performance was obtained with with **16 threads** (2.989s execution time, 10.99x speedup).
For **Blur**, optimizations included partial loop unrolling and loop peeling within the filter function, reaching a **4.28x speedup**. A SIMD version was also tested, but it produced noisy data that failed validation tests. Finally, multithreading was applied: the best execution time was achieved using **8 threads** (2.980s execution time, 6.27x speedup).