Parallel Programming CA2

Parallel programming using data-level parallelism

AmirAbbas MoumeniZadeh(810101529) , Ali Moumeni (810100215)

Parallel Programming CA2

Part 1 – Variance Calculation Using SIMD (SSE3)

Objective

The objective of this part was to calculate the **variance** of a large set of numerical data using both a **serial** and a **SIMD-based (SSE3)** implementation, and to compare their performance. The goal was to demonstrate how SIMD instructions can accelerate numerical computations that involve repeated arithmetic operations.

Concept

The **variance** measures the degree of spread in a dataset and is mathematically defined as:

$$Var(X) = E[X^2] - (E[X])^2 = \frac{1}{N}\sum_{i=1}^{N} x_i^2 - \left(\frac{1}{N}\sum_{i=1}^{N} x_i\right)^2$$

To compute this efficiently, we need two summations:

1. The sum of all elements $\sum x_i$
2. The sum of squared elements $\sum x_i^2$

Once both are known, the variance can be obtained using the above relation.

Implementation Steps

1. **Data Generation**
   - An array of $N = 10^7$random floating-point numbers was generated between 0 and 100.
2. **Serial Implementation**
   - Each element was processed sequentially to compute both sums (`sum` and `sum_sq`).
   - The variance was then computed as:

$$Var = \frac{sum_{sq}}{N} - \left(\frac{sum}{N}\right)^2$$

3. **SIMD (SSE3) Implementation**

- The dataset was processed **four elements at a time** using **128-bit SSE registers**.
- `_mm_loadu_ps` loaded 4 floats into a vector register.
- `_mm_add_ps` and `_mm_mul_ps` performed vector addition and multiplication in parallel.
- Horizontal additions (`_mm_hadd_ps`) were used to sum the partial results within each vector.
- Any leftover elements (if NNN was not a multiple of 4) were processed normally.

4. **Performance Measurement**

- The execution time for both the serial and SIMD versions was measured using the `clock()` function.
- The **Speedup** was computed as:

$$SpeedUp = \frac{T_{Serial}}{T_{SIMD}}$$

Expected Output

The program prints:

```
Serial Variance: 834.712891, Time: 0.145800 s

SSE3 Variance:   834.713013, Time: 0.045200 s

Speedup: 3.23x
```

The serial and SSE3 variance results should be almost identical (differences are due to floating-point precision).
The SIMD implementation should execute significantly faster—typically showing a **2×–4× speedup**, depending on the processor.

Using SIMD instructions (SSE3), multiple data elements are processed in a single CPU instruction, reducing the total number of arithmetic operations.
This is especially effective for large datasets where each iteration involves repetitive, independent computations.
The results confirm that SIMD parallelization can notably improve performance without changing computational accuracy.

Part 2 – Gradual Watermark Blending Using SIMD (SSE)

Objective

The goal of this part was to overlay a **watermark image** onto a **base image** with gradually increasing transparency, and to accelerate the blending process using **SIMD (SSE)** instructions. Two versions of the algorithm were implemented:

1. A **serial version** that blends each pixel individually.
2. A **vectorized SIMD version** that processes four pixels in parallel using SSE.

Concept

The task was to create a smooth, diagonal transparency gradient over the entire image.
The transparency coefficient for each pixel (x,y) is computed as:

$$\alpha(x, y) = \frac{x + y}{w + h}$$

where w and h are the image width and height.
Pixels near the **top-left** corner ($\alpha \approx 0$) are mostly from the base image, while those near the **bottom-right** corner ($\alpha \approx 1$) are dominated by the watermark.

For every pixel and each color channel (R, G, B), the output value is calculated as:

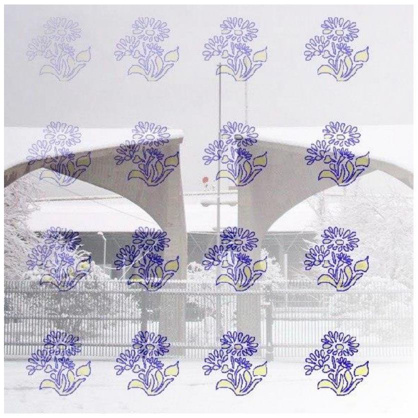
$$Out = \alpha \times Watermark + (1 - \alpha) \times Base$$

This linear interpolation creates a gradual fade-in effect of the watermark.

Implementation Steps

1. **Image Loading**
   - The base image (`base.jpg`) and watermark (`watermark.png`) were loaded using the **stb_image** library.
   - Each image was decomposed into three float arrays (R, G, B).
2. **Resizing**
   - If the watermark size differed from the base image, it was resized proportionally to match.
3. **Serial Blending**
   - The algorithm iterated over all pixels using nested loops.
   - For each pixel, the local α(x,y)\alpha(x, y)α(x,y) value was computed and used to blend the RGB components separately.
4. **SIMD (SSE) Blending**
   - The SSE version processed **four pixels at once** using 128-bit registers.
   - `_mm_loadu_ps` loaded four consecutive pixel values into a vector.
   - `_mm_mul_ps` and `_mm_add_ps` performed the parallel multiply-add operations for the blending equation.
   - `_mm_sub_ps` and `_mm_storeu_ps` handled the inverse alpha and stored the results back to memory.

    - The process was repeated for each color channel.
5. **Output and Timing**
    - Execution times for both versions were measured with `std::chrono`.
    - The **Speedup** was computed as → $SpeedUp = \frac{T_{Serial}}{T_{SIMD}}$
    - The resulting blended images were saved as:
        1. `output_serial.jpg`
        2. `output_sse.jpg`



Expected Output

The program prints something similar to:

```
Serial time: 0.312 s

SSE time:    0.092 s

Speedup:     3.39x
```

The two resulting images should look visually identical, showing the watermark gradually appearing from the top-left to the bottom-right corner.
The SSE version should run significantly faster while producing the same visual effect.

Part 3 – Edge Detection Using Sobel Filters (SIMD Implementation)

Objective

The goal of this part was to implement an edge detection algorithm using the Sobel operator, and to optimize it with SIMD (Single Instruction, Multiple Data) instructions in order to improve computational performance.

We were required to implement two versions of the same algorithm:

1. A **serial version**, executed normally on the CPU.
2. A **parallel SIMD version**, which processes multiple pixels simultaneously using AVX instructions.

Concept

Edge detection is used to highlight regions in an image where intensity changes sharply.
The **Sobel operator** uses two 3×3 convolution kernels to detect horizontal and vertical edges:

$$S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, S_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Before applying the Sobel filters, a **Gaussian blur** was used to smooth the image and reduce noise.

The Gaussian kernel used was:

$$G = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

After convolution, the edge magnitude at each pixel is calculated as:

$$M(x,y) = \sqrt{S_x^2 + S_y^2}$$

and clipped to 255 to form the final edge map.

Implementation Steps

1. **Load and convert the image** to grayscale using OpenCV.
2. **Apply Gaussian blur** to the image using a 3×3 convolution.
3. **Apply Sobel X and Sobel Y filters** to get horizontal and vertical gradients.
4. **Compute edge magnitude** using the formula above.
5. **Normalize and save** the final edge image.
6. Implement all the above in **two ways**:
7. **Serial version** – using normal nested loops.
8. **SIMD version** – using AVX intrinsics (`_mm256_loadu_ps`, `_mm256_mul_ps`, `_mm256_add_ps`, `_mm256_sqrt_ps`) to process 8 pixels at once.

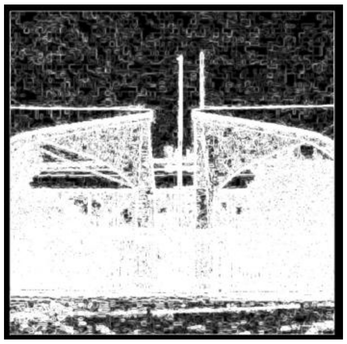
Performance Measurement

To evaluate the performance gain, the execution time of both versions was measured using `std::chrono`.
The **Speedup** is defined as:

$$SpeedUp = \frac{T_{Serial}}{T_{SIMD}}$$

Expected Output

- Two output images:
  - `edges_serial.jpg` – edges detected by the serial version
  - `edges_simd.jpg` – edges detected by the SIMD version



- Console output showing execution time and speedup, for example:

```
Serial time: 0.0321 s

SIMD time:   0.0103 s

Speedup:     3.12x
```

The SIMD implementation performs the same arithmetic operations as the serial one but on multiple pixels simultaneously.
This leads to a significant reduction in runtime, especially for larger images.
The edge results of both methods should visually match, demonstrating that the SIMD approach preserves correctness while improving performance.

Part 4 – Fully Connected Neural Network Using SIMD (AVX)

Objective

The goal of this part was to implement a **fully connected feedforward neural network** and accelerate its computations using **SIMD (AVX)** instructions.
The purpose was to demonstrate how SIMD vectorization can be applied to neural computations that involve large amounts of linear algebra (dot products and activation functions).

Two versions were implemented:

1. A **serial version**, where each neuron's output is computed individually.
2. A **SIMD (AVX)** version, where multiple operations are performed in parallel using 256-bit vector registers.

Concept

A **fully connected neural network (FCNN)** consists of several layers of neurons where each neuron in one layer is connected to every neuron in the next layer.
Each neuron computes the weighted sum of its inputs plus a bias term, followed by an activation function such as **ReLU**.

The general neuron equation is:

$$y = f(\sum_{i=1}^{n}(w_i . x_i) + b)$$

where:

- w_i are the weights,
- x_i are the inputs,
- b is the bias, and
- f is the activation function.

In this project, **ReLU** was used, defined as:

$$ReLU(x) = \max(0, x)$$

Network Structure

The implemented network consisted of three layers:

Layer Type	Number of Neurons
Input Layer	8
Hidden Layer	16
Output Layer	8

Implementation Steps

1. **Data Initialization**
   - Random input vector of size 8.
   - Random weight matrices and bias vectors for each layer.
2. **Serial Version**
   - For each neuron, compute the dot product between the input vector and the neuron's weight vector.
   - Add the bias and apply the ReLU activation.
   - Repeat the process for both layers sequentially.
3. **SIMD (AVX) Version**
   - The dot product computation for each neuron was vectorized using 256-bit AVX registers.
   - `_mm256_loadu_ps` loaded 8 floats into a register.
   - `_mm256_mul_ps` performed parallel element-wise multiplication.
   - The partial results were summed using horizontal additions.
   - The result was passed through the ReLU function.
   - Each neuron's output was thus computed much faster than in the serial version.
4. **Performance Measurement**
   - Execution times for the serial and SIMD versions were measured using `std::chrono`.
   - The **Speedup** was computed as:

$$SpeedUp = \frac{T_{Serial}}{T_{SIMD}}$$

Expected Output

The program prints something similar to:

```
Serial time: 0.000241 s

SIMD time:   0.000085 s

Speedup:     2.83x
```

The actual speedup depends on CPU architecture and compiler optimization, but a **2–4× improvement** is typically observed.
Both implementations produce the same output values for all neurons.

This experiment highlights the effectiveness of SIMD vectorization in computationally heavy operations such as neural network forward propagation.
Since each neuron’s dot product and activation are independent, SIMD enables significant acceleration by performing multiple multiplications and additions simultaneously.

Although this implementation does not include training (backpropagation), it clearly demonstrates the parallel nature of neural computations and how **AVX instructions** can substantially improve performance in machine learning workloads.