Silesian University of Technology

Assembly languages

Gaussian Blur cpp asm

Emanuel Jureczko

Agenda

Gaussian blur explanation and visualization

Cpp implementation

Multithreading issue

Desktop application

Cpp & Asm performance

Observations and conclusions

Gaussian blur explanation

Image-blurring filter that uses a Gaussian function for calculating the transformation applied to each pixel

Using 2 dimensional or 1 dimensional kernel

	2 dimensional	1 dimensional
formula	$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$	$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$
Time complexity	$O(h \cdot w \cdot k^2)$	$O(h \cdot w \cdot 2 \cdot k)$

h - image height
w - image width
k - kernel size

Separable Convolution Method

Using a 1 dimensional kernel going through all pixels of the image two times. Ones horizontally, ones vertically (kernel flipped 90°) on the result of the previous transformation.

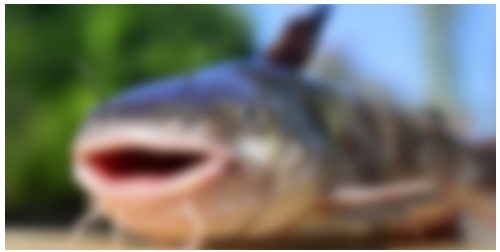
Just a visualization. The actual pipeline is on the next slide.

only horizontal

+

only vertical

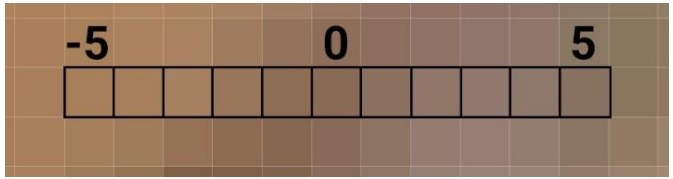
=



outcome

Catfish image - https://rybyswiata.pl/ryby-slodkowodne/sum-smazony-czy-pieczony/

Process explanation



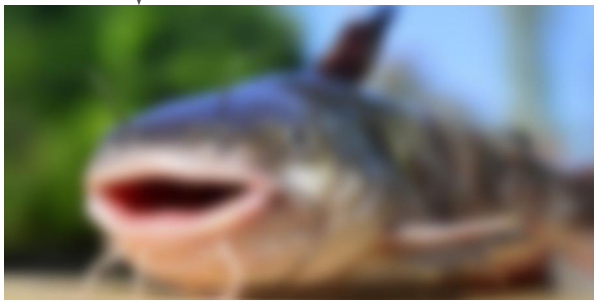
Kernel of size = 11

Kernel is an array of uneven elements.
Each cell contains a floating-point number calculated from the Gauss formula, where
- sigma must be given as a parameter
- x is the "index" of the cell (e.g. -5 ... 5)
and later normalised.

$k = 2\lceil 3\sigma \rceil + 1$ - Standard deviation (sigma) to kernel size relation

horizontal blur

vertical blur



Why does 1 dimensional kernel work?

Simple but sufficient answer

```
2D Gaussian Kernel:
[[0.07511361 0.1238414  0.07511361]
 [0.1238414  0.20417996 0.1238414 ]
 [0.07511361 0.1238414  0.07511361]]

1D Gaussian Kernel:
[0.27406862 0.45186276 0.27406862]

1Dx1D^T:
[[0.07511361 0.1238414  0.07511361]
 [0.1238414  0.20417996 0.1238414 ]
 [0.07511361 0.1238414  0.07511361]]
```

Mathematically

$$G(x, y) = G_x(x) \cdot G_y(y)$$

Additional explanation

$$\alpha XY = (\alpha X)\ Y = X\ (\alpha Y)$$

$$A = A'\ A''$$

$$\alpha A = (\alpha A')\ A'' = A'\ (\alpha A'')$$

α - pixel value

A - 2 dimensional kernel

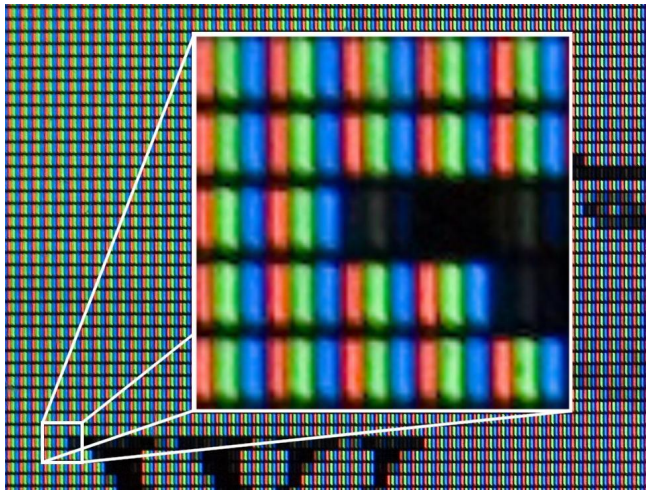
$A'\ A''$ - 1 dimensional kernels

Transformation

Is performed on each pixel element (RGB values)

$$p_{transformed} = \sum_{i=0}^{k-1} p_i \cdot kernel_i$$

k	-	kernel size
p_i	-	pixel in kernel at index i
$kernel_i$	-	normalized value in kernel at index i



Code 1/2

This is an implementation of a Gaussian blur in cpp, adapted for use with a single thread.
The implementation of multithreading introduces minor alterations to the boundary-checking process.

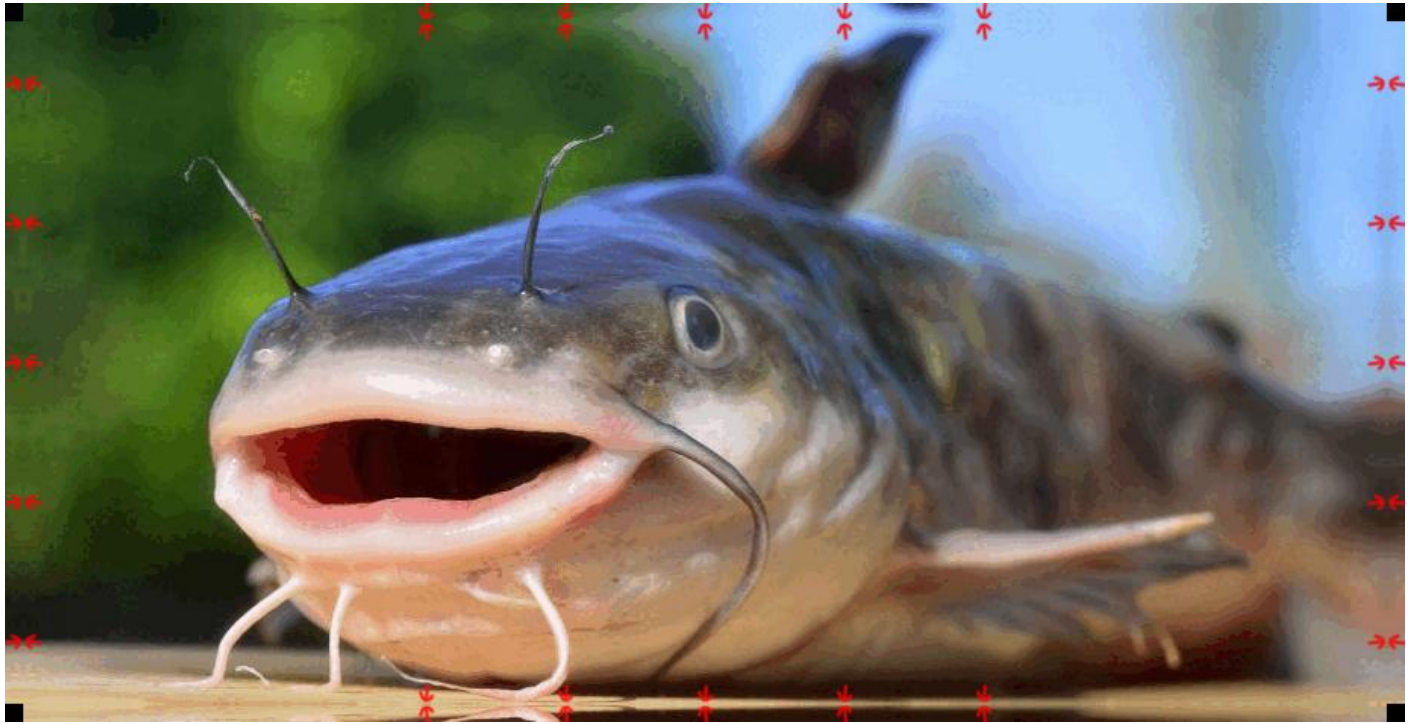
```cpp
extern "C" __declspec(dllexport) void gaussBlur(unsigned char* bitmapData, unsigned char* tempData, float* kernel,
    int width, int height, int stride, int kernelSize) {

    int offset = kernelSize / 2; // we consider the middle index as 0 e.g. [-2, -1, 0, 1, 2]

    // Horizontal blur
    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            int pixelIndex = y * stride + x * 3; // BMP uses 3 bytes per pixel (BGR format)

            float blurredPixelR = 0, blurredPixelG = 0, blurredPixelB = 0;

            for (int i = 0; i < kernelSize; ++i) {
                int selectedX = x + (i - offset);
                int selectedIndex = pixelIndex;

                //if out of border we want to mirror the edge
                //-2 -1 [ 0 1 ... n-1 n ] n+1 n+2
                // 1  0 [ 0 1 ... n-1 n ] n   n-1
                if (selectedX < 0) {
                    selectedIndex += (1 + selectedX) * -3; //(1 + selectedX) * -1 * 3;
                }
                else if (selectedX >= width) {
                    selectedIndex += (width - 1 - selectedX) * 3;
                }
                else {
                    selectedIndex += ((i - offset) * 3);
                }

                blurredPixelB += bitmapData[selectedIndex] * kernel[i];
                blurredPixelG += bitmapData[selectedIndex + 1] * kernel[i];
                blurredPixelR += bitmapData[selectedIndex + 2] * kernel[i];
            }

            tempData[pixelIndex]     = static_cast<unsigned char>(std::round(blurredPixelB));
            tempData[pixelIndex + 1] = static_cast<unsigned char>(std::round(blurredPixelG));
            tempData[pixelIndex + 2] = static_cast<unsigned char>(std::round(blurredPixelR));
        }
    }
```

Code 2/2

```
// Vertical blur
for (int y = 0; y < height; ++y) {
    for (int x = 0; x < width; ++x) {
        int pixelIndex = y * stride + x * 3;


        float blurredPixelR = 0, blurredPixelG = 0, blurredPixelB = 0;

        for (int i = 0; i < kernelSize; ++i) {
            int selectedY = y + (i - offset);
            int selectedIndex = pixelIndex;

            //if out of border we want to mirror the edge
            //-2 -1 [ 0 1 ... n-1 n ] n+1 n+2
            // 1  0 [ 0 1 ... n-1 n ] n   n-1
            if (selectedY < 0) {
                selectedIndex += (1 + selectedY) * -1 * stride; //(1 + selectedX) * -1 * 3;
            }
            else if (selectedY >= height) {
                selectedIndex += (height - 1 - selectedY) * stride;
            }
            else {
                selectedIndex += ((i - offset) * stride);
            }


            blurredPixelB += tempData[selectedIndex] * kernel[i];
            blurredPixelG += tempData[selectedIndex + 1] * kernel[i];
            blurredPixelR += tempData[selectedIndex + 2] * kernel[i];
        }

        bitmapData[pixelIndex]     = static_cast<unsigned char>(std::round(blurredPixelB));
        bitmapData[pixelIndex + 1] = static_cast<unsigned char>(std::round(blurredPixelG));
        bitmapData[pixelIndex + 2] = static_cast<unsigned char>(std::round(blurredPixelR));
    }
}
```

Out of boundary handling visualization

Once over the image edge, pixels are being mirrored.



Multithreading issue

While each thread writes data in distinct area, they also read data from neighbouring areas. It must be ensured that no thread while performing the 2nd stage (vertical blur) reads a neighbouring area pixel that has not undergone transformation.

The image is split into sub-images. White lines represent borders. More saturated areas represent pixels accessed by neighbouring threads

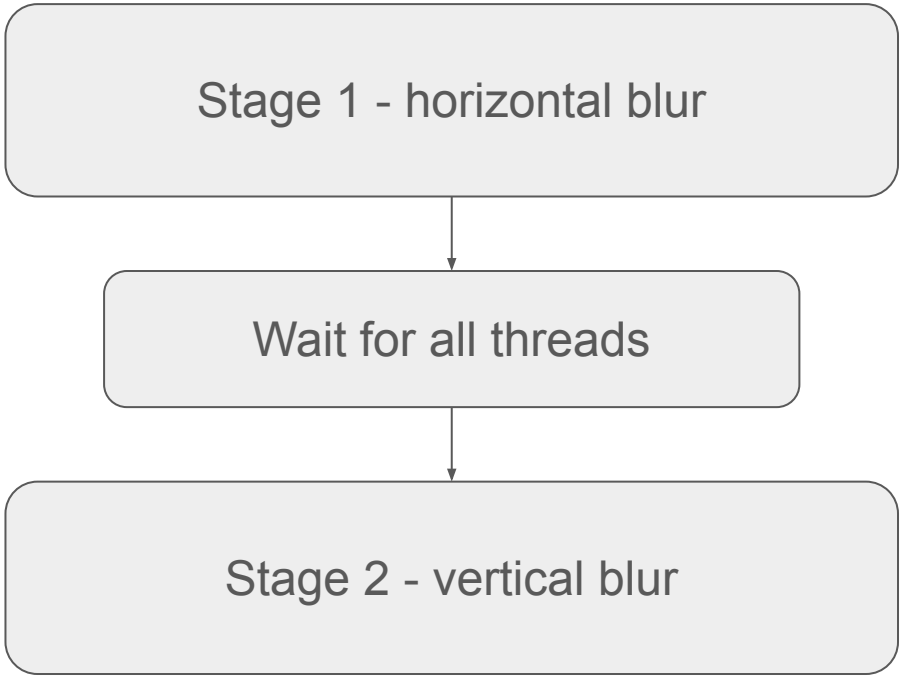
Segmentation

```
int[] startHeights = new int[numOfThreads];
int[] endHeights = new int[numOfThreads];
int segmentHeight = height / numOfThreads;

// Segment the image
for (int i = 0; i < numOfThreads; ++i) {
    startHeights[i] = i * segmentHeight;
    endHeights[i] = (i + 1) * segmentHeight;

    if (i == numOfThreads - 1) { endHeights[i] = height; }
}
```

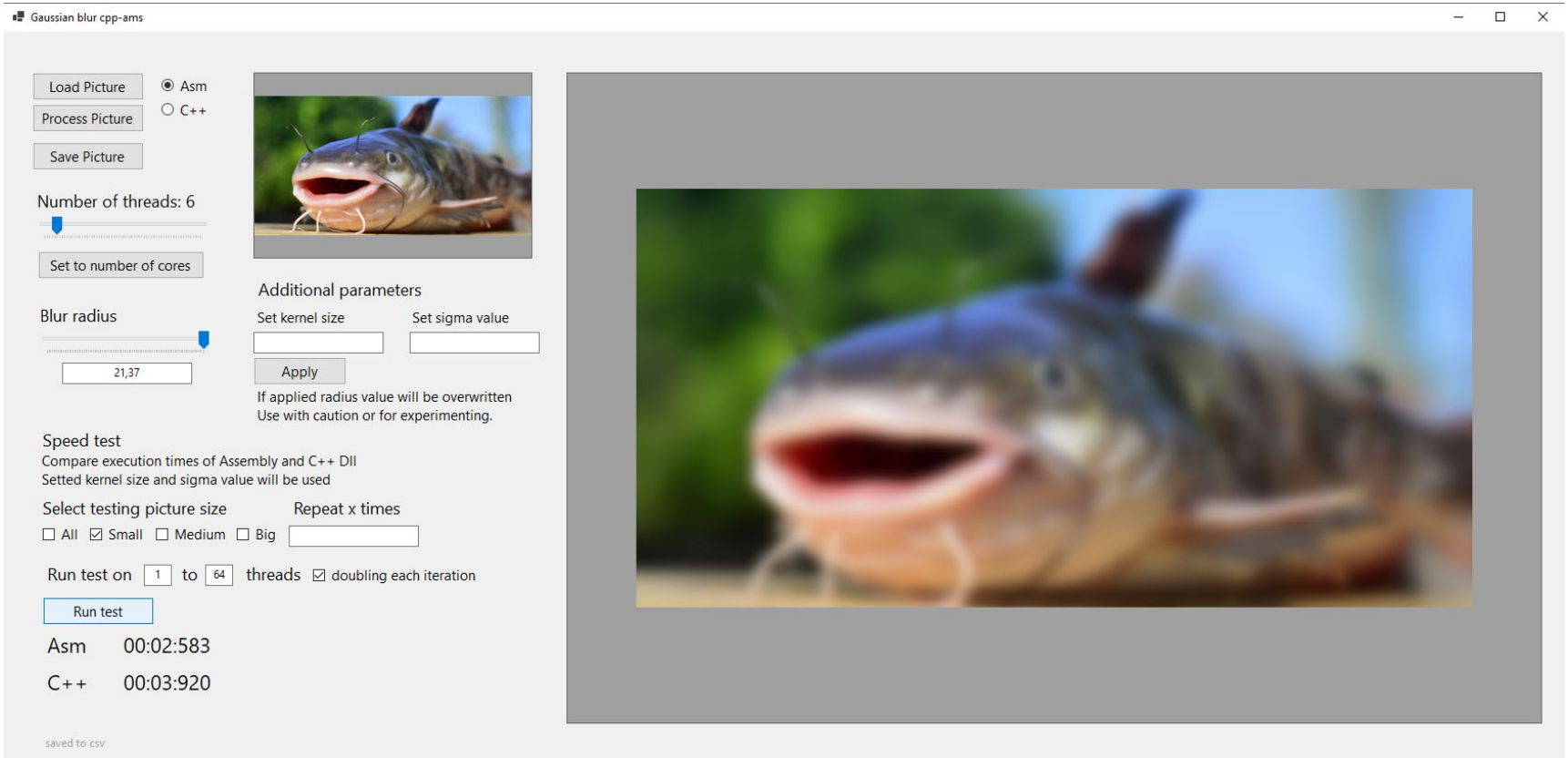
Solution

Before starting stage 2 every thread must complete stage 1.

While losing time by waiting for the slowest thread, time is saved by not needing to supervise neighbour areas data access.



Desktop application



Functionality

Applying Gaussian blur

Adjusting blur settings

Saving blurred image

Selecting number of threads to be used

Choosing implementation between cpp and asm

Performing a speed test between implementations

Saving test result in csv files

Cpp & Asm performance

Cpu used for measurements

Name	AMD FX(tm)-6300 Six-Core Processor
Number of cores	3
Number of logical processors	6

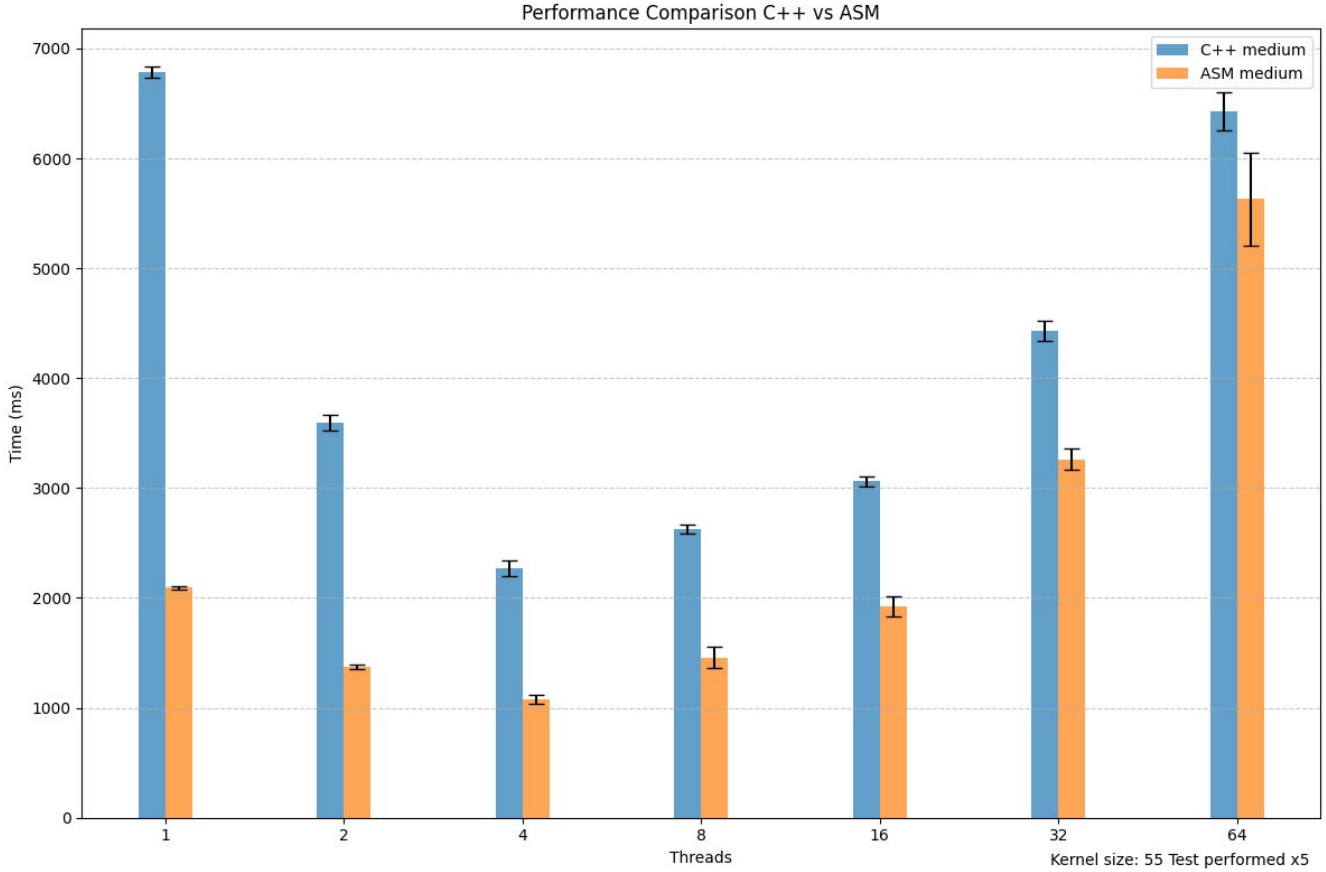
Test data size

small	900x450
medium	2600x1462
big	4624x3472

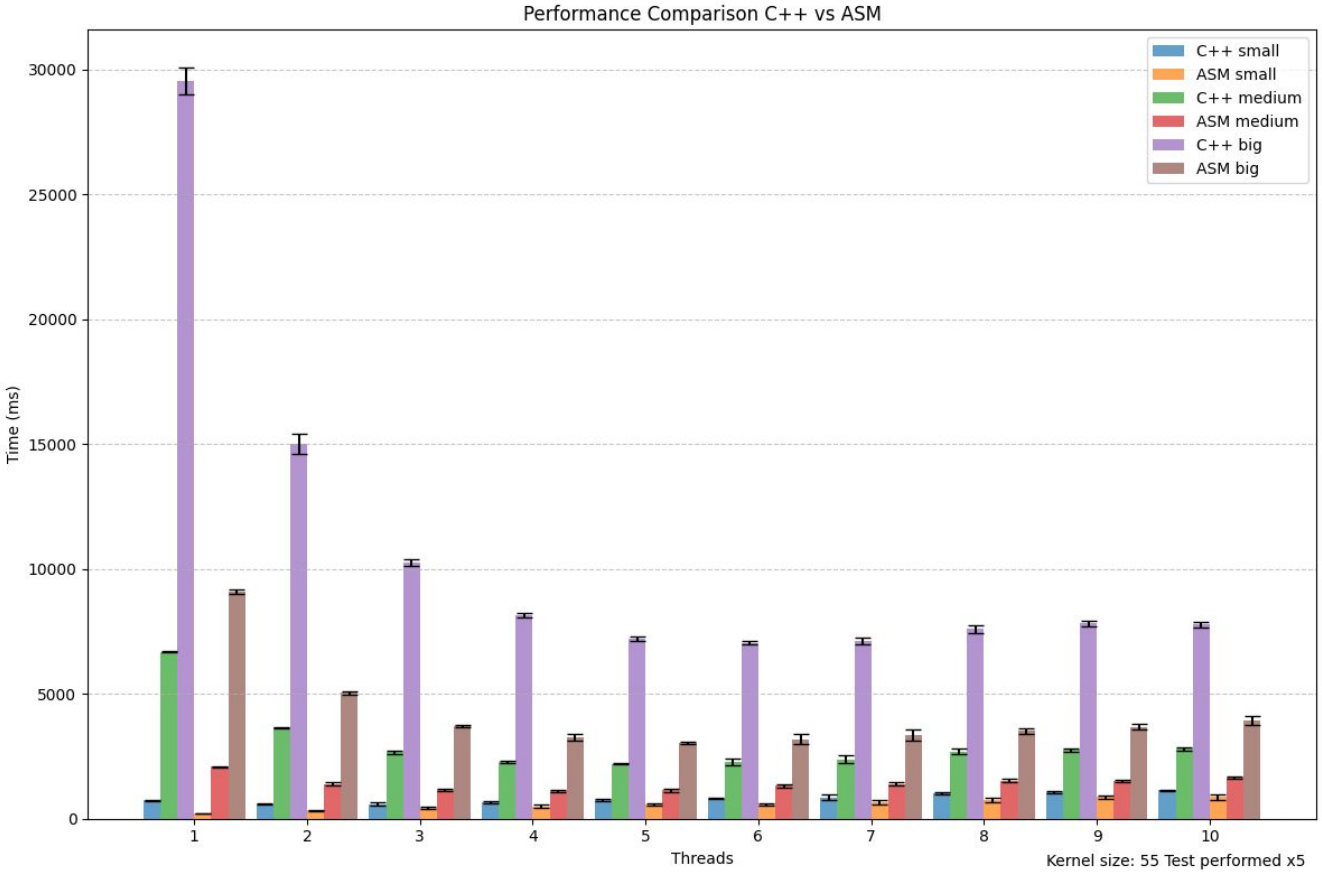
Cpp optimization

Maximize Speed (/O2)

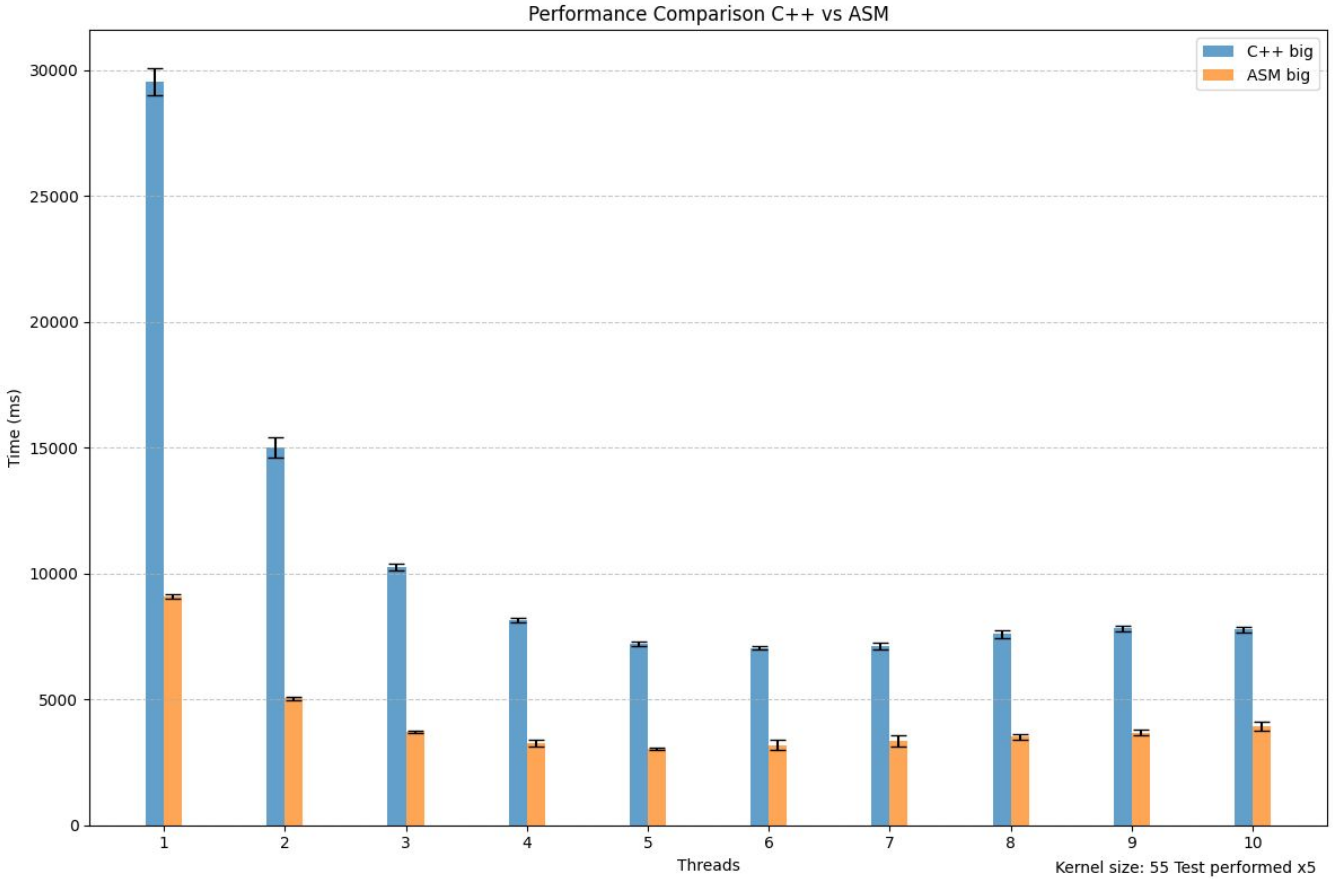
Medium data 1 - 64 threads



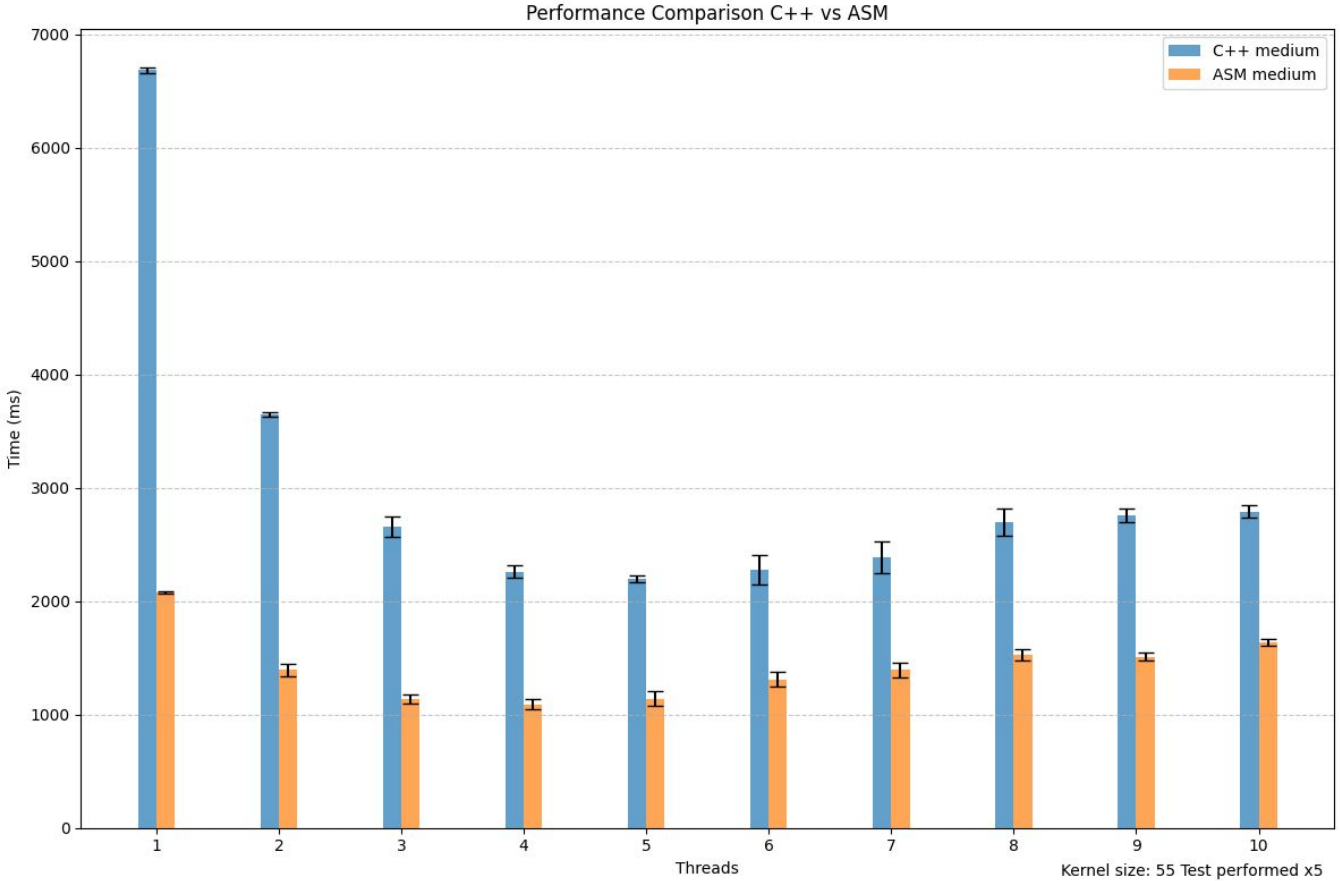
Big, medium, small data 1 - 10 threads



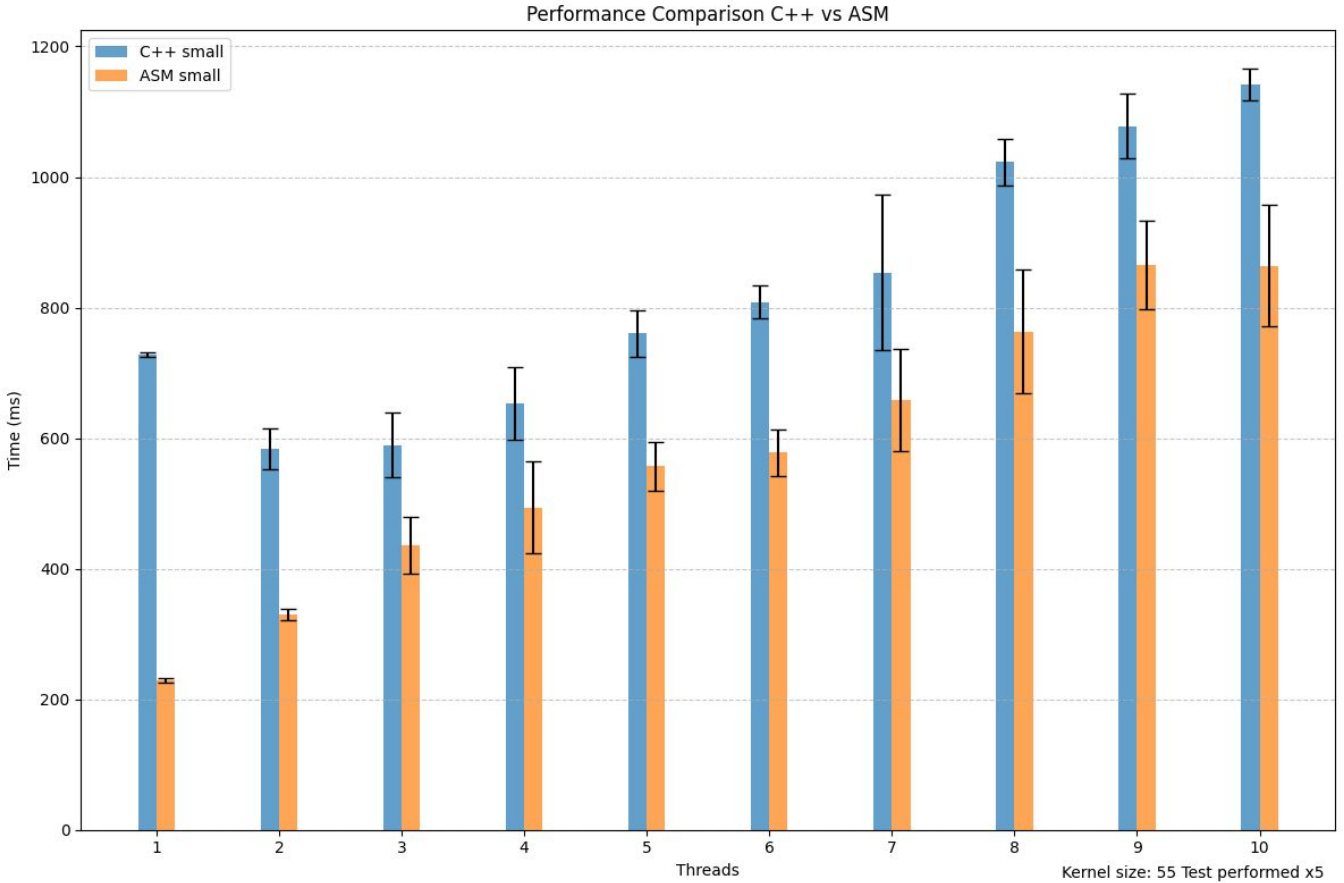
Big data 1 - 10 threads



Medium data 1 - 10 threads



Small data 1 - 10 threads



Observations

Only larger data should be segmented.

If the number of logical processors is exceeded, performance will suffer.

More time-consuming processes may benefit more from multithreading.

Conclusions

Although Asm is faster, the time spent writing it may not be worth it. It may be more beneficial to spend extra time optimising the Cpp code rather than writing in Asm.

The fact that Cpp is that much slower may be a sign of poor code quality. Compilers usually do a good job.