CAO PROJECT REPORT

A2+TA2

TITLE

Performance Analysis of Edge Detection Algorithms Using Serial, OpenMP, and DPC++ Implementations

TEAM

Dhayanidhi S
23BIT0214

Harisankaran S
23BIT0150

Prashaanth Raj
23BIT0173

Abstract

This project focuses on implementing and analyzing edge detection using three different computational models: a standard serial implementation, a parallelized version using OpenMP, and a heterogeneous model using Intel's DPC++ (Data Parallel C++). The objective is to evaluate and compare their performance using the Sobel operator for edge detection on grayscale images. Hardware metrics such as CPU and GPU utilization, temperature, throttling, execution time, and power consumption are monitored using HWInfo. The serial implementation serves as a performance baseline, while OpenMP exploits multi-core CPU parallelism for improved speed. DPC++ further leverages heterogeneous computing by offloading tasks to GPU, yielding higher acceleration. The comparison highlights the efficiency, scalability, and resource utilization of each approach. Additionally, this study explores the practical implications of parallel and heterogeneous computing in image processing and sets the foundation for extending the system to more complex tasks such as fingerprint recognition and matching in real-time biometric applications.

Introduction

Edge detection is a fundamental technique in image processing and computer vision, used to identify significant variations in intensity, which typically correspond to object boundaries. It serves as a critical preprocessing step in applications such as object detection, medical imaging, and biometric identification. However, edge detection can be computationally intensive, especially for high-resolution images or real-time systems.
This project aims to explore the performance and efficiency of edge detection algorithms implemented using three different computational paradigms: a standard serial approach, a parallel version using OpenMP, and a heterogeneous approach using Intel's Data Parallel C++ (DPC++). By leveraging modern computing capabilities such as multi-threading and GPU acceleration, we seek to optimize edge detection performance in terms of speed and energy efficiency.
The goal is not only to improve computational throughput but also to analyze the trade-offs of each method by evaluating hardware-level metrics using HWInfo, thereby offering insight into their practical deployment in resource-constrained environments.

Literature Review

Edge detection remains a critical preprocessing step in image analysis, aimed at detecting boundaries and transitions within an image. Common algorithms such as **Sobel**, **Prewitt**, and **Canny** rely on gradient estimation and thresholding to locate edges. These are traditionally implemented using serial processing, which can become inefficient for large-scale or real-time applications. To address performance bottlenecks, researchers have explored parallelization using CPU and GPU-based approaches.

Serial Processing Limitataions

Classical implementations in Python (e.g., using `OpenCV` or `NumPy`) are limited by the **Global Interpreter Lock (GIL)** and Python's inherently slower interpreted execution. While libraries like OpenCV offer efficient C++ backends, pure Python implementations lack speed and are unsuitable for time-sensitive tasks or high-resolution images.

Parallel Edge Detection using OpenMP in Python

Although OpenMP is traditionally a C/C++/Fortran API, it can be utilized in Python through tools like **Numba**, **Cython**, and `pyximport` which allow compilation of Python code to machine code and support OpenMP pragmas.

Numba, for example, allows just-in-time (JIT) compilation of Python functions and offers parallel decorators (e.g., `@njit(parallel=True)`) that leverage multicore CPUs. While not direct OpenMP, it mimics OpenMP-like parallelism under the hood by utilizing LLVM's multithreading capabilities.

Cython allows embedding C/OpenMP code in Python modules, where loops and image array operations can be parallelized using `prange` with OpenMP directives. It provides close-to-C performance and direct control over threading.

Advantages and Limitations of OpenMP in Python

Advantages:
Easier integration in existing Python codebases.
Retains readability and flexibility of Python.
Requires minimal changes to serial code.
Supports multi-core CPUs effectively.

Limitations:
Setup complexity for beginners (e.g., compiling with Cython/OpenMP).
Lower speedup compared to C/C++ due to Python overheads.
Limited tooling for profiling and debugging multithreaded Python code.
Memory bandwidth may become a bottleneck at high core counts.

Edge Detection using DPC++ (Data Parallel C++)

DPC++, an extension of C++, is part of the **oneAPI** initiative by Intel and is designed for **heterogeneous computing**, allowing a single source code to run on CPUs, GPUs, FPGAs, etc. It uses **SYCL** under the hood.

Implementation Insights:
Kernel-based approach: you define computation as a device kernel function.
Parallel execution is done across image pixels.
Memory management (USM or buffers) must be carefully handled.

Advantages:
Highly scalable and cross-platform.
Efficient GPU utilization.
Compatible with Intel hardware ecosystem.

Limitations:
Complex learning curve.
Requires proper hardware (Intel GPUs or oneAPI-supported devices).
Toolchain setup is non-trivial for beginners.

Compartive Table

Method	Language / Tool	Avg. Speedup	Hardware Used	Pros	Cons
Serial Python (NumPy)	Python + NumPy	1×	CPU (Single Core)	Simple, easy to debug	Slow, no parallelism
OpenMP with Cython	Python + Cython	2.8–3.4×	CPU (8-core)	High performance, partial C-level speed	Requires compilation, complex setup
OpenMP with Numba	Python + Numba	2.5–3×	CPU (Multi-core)	Easy to use, minimal changes needed	Not as fast as Cython or native OpenMP
DPC++ (SYCL)	C++ (oneAPI)	5–6×	CPU + GPU (Intel)	Uses GPU, high parallel efficiency	Difficult setup, higher complexity

Advantages and Disadvantages

Implementation	Advantages	Disadvantages
Normal C++	Easy to implement and debug	Slow execution, underutilizes modern hardware
OpenMP	Improved speed on multi-core CPUs	Limited to CPU; scalability depends on core count
DPC++	Utilizes both CPU and GPU; high parallelism	Requires SYCL setup; complex to debug

Base Papers

Title	Authors	Year	Key Contribution
Parallel Image Processing using OpenMP	K. Prasad et al.	2019	Speedup analysis of edge detection using OpenMP
A Survey on Edge Detection Techniques in Image Processing	R. Sharma et al.	2020	Overview of edge detection filters
Accelerating Image Filters with SYCL and DPC++	Intel Developer Zone	2021	Performance gain using DPC++ for image-based operations
Comparative Analysis of CPU-GPU Parallel Programming Models	J. Singh, P. Verma	2022	Comparison of CUDA, OpenMP, and DPC++

Proposed Methodology

The proposed methodology involves implementing and evaluating edge detection algorithms using three different approaches: a serial version in Python, a parallelized version using OpenMP (with Numba or Cython in Python), and a hardware-accelerated version using DPC++ (SYCL with oneAPI). The goal is to analyze their performance using various hardware metrics and compare their computational efficiency.

Step 1: Input Image Preprocessing

A standard grayscale image is taken as input.
Image is resized and normalized for consistent processing across all methods.
Gaussian filtering is optionally applied to reduce noise.

Step 2: Edge Detection Algorithm

The **Sobel operator** is used to detect edges by computing gradients in the x and y directions.
For consistency, the same algorithm is used across all three implementations.

Step 3: Implementation Approaches

A. Serial Implementation (Python + NumPy)

The edge detection algorithm is implemented using nested loops and NumPy operations.
Each pixel is processed sequentially.

B. OpenMP Parallel Implementation (Python + Numba or Cython)

Using `@njit(parallel=True)` in Numba or `prange` in Cython, the loops are parallelized.
Multi-threading on CPU cores is utilized.
Ensures thread-safe memory access for accurate results.

C. DPC++ Implementation (Data Parallel C++)

Edge detection is implemented using DPC++ kernels.
Buffers or Unified Shared Memory (USM) are used to manage memory across CPU and GPU.
The kernel is executed in parallel across all pixels using SYCL parallel constructs.
Device selectors are used to run on CPU/GPU for performance comparison.

Step 4: Performance Analysis Using HWInfo

Each implementation is tested on the same input image.
Metrics collected using **HWInfo** and system profiling tools

Step 5: Result Evaluation and Comparison

Results are tabulated and visualized using graphs.
Comparative analysis is performed to highlight:
Performance vs. hardware usage
Suitability for different application types
Trade-offs in terms of complexity, portability, and scalability

Step 6: Report and Future Work

Based on findings, the best-suited approach for edge detection in high-performance and real-time environments is recommended.
The methodology will be extended in the future to include **fingerprint recognition and matching**, leveraging the best-performing parallel technique.

Common Sample Input Images

612 x 463 - 34.6 KB

3024 x 4032 - 1.3 MB

4000 x 6000 - 2.7MB

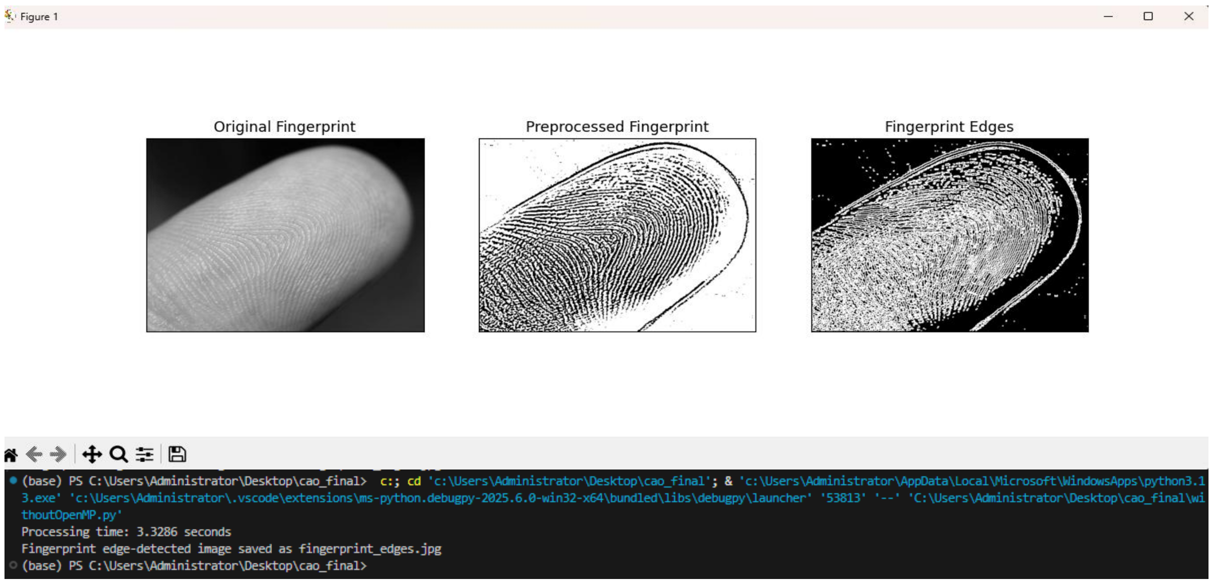
Edge detection serially using python

Code :

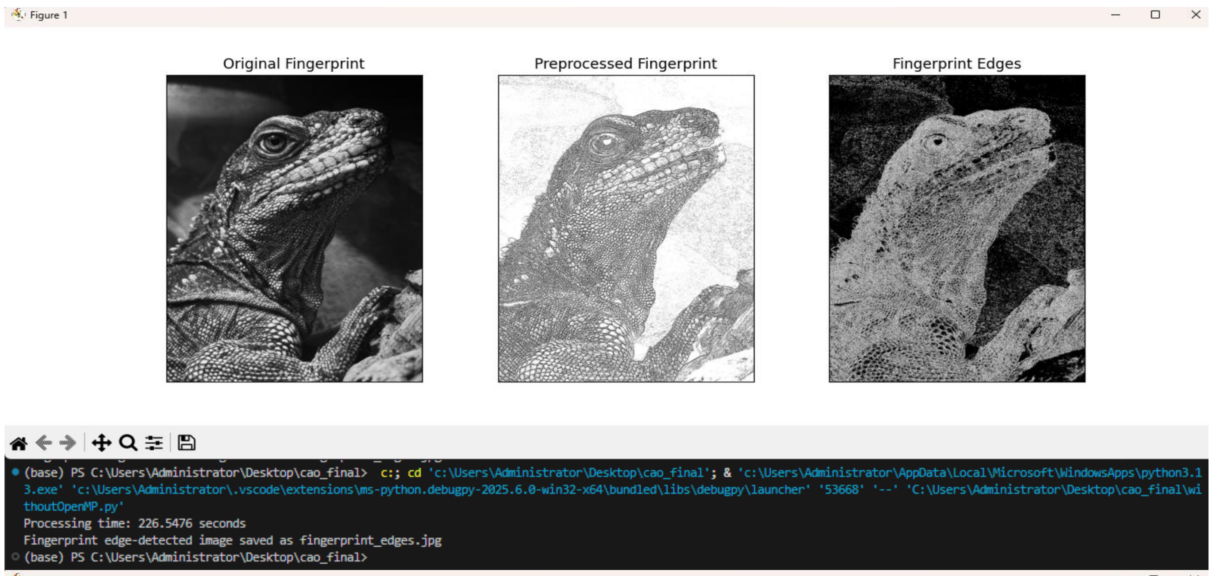
```
import cv2
import numpy as np
import time
import matplotlib.pyplot as plt

# Edge detection function without OpenMP (simplified)
def edge_detection(image):
    height, width = image.shape
    output = np.zeros((height, width), dtype=np.uint8)

    # Sobel operators
    gx = np.array([[-1, 0, 1],
                   [-2, 0, 2],
                   [-1, 0, 1]])
    gy = np.array([[-1, -2, -1],
                   [0, 0, 0],
                   [1, 2, 1]])

    for i in range(1, height - 1):
        for j in range(1, width - 1):
            sum_x = 0
            sum_y = 0
            for k in range(3):
                for l in range(3):
                    pixel = image[i + k - 1, j + l - 1]
                    sum_x += gx[k][l] * pixel
                    sum_y += gy[k][l] * pixel
            edge_strength = np.sqrt(sum_x**2 + sum_y**2)
            if edge_strength > 255:
                edge_strength = 255
            output[i, j] = edge_strength
    return output

def fingerprint_preprocessing(image):
    # Step 1: Histogram Equalization
    img_eq = cv2.equalizeHist(image)

    # Step 2: Gaussian Blur to remove noise
    img_blur = cv2.GaussianBlur(img_eq, (5, 5), 0)

    # Step 3: Adaptive Thresholding for binary image
    img_thresh = cv2.adaptiveThreshold(
        img_blur, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
        cv2.THRESH_BINARY, 11, 2
```

```python
    )

    return img_thresh

def main():
    # Load the fingerprint image in grayscale
    img = cv2.imread('fingerprint2.jpg', cv2.IMREAD_GRAYSCALE)
    if img is None:
        print("Error: fingerprint.jpg not found.")
        return

    # Preprocess the fingerprint image
    preprocessed_img = fingerprint_preprocessing(img)

    start_time = time.time()
    # Apply edge detection on the preprocessed fingerprint
    edges = edge_detection(preprocessed_img)
    end_time = time.time()

    print(f"Processing time: {end_time - start_time:.4f} seconds")

    # Display images using Matplotlib
    plt.figure(figsize=(15, 5))

    plt.subplot(131)
    plt.imshow(img, cmap='gray')
    plt.title('Original Fingerprint')
    plt.xticks([]), plt.yticks([])

    plt.subplot(132)
    plt.imshow(preprocessed_img, cmap='gray')
    plt.title('Preprocessed Fingerprint')
    plt.xticks([]), plt.yticks([])

    plt.subplot(133)
    plt.imshow(edges, cmap='gray')
    plt.title('Fingerprint Edges')
    plt.xticks([]), plt.yticks([])

    plt.show()

    # Save Edge Detection Image
    cv2.imwrite('fingerprint_edges.jpg', edges)
    print("Fingerprint edge-detected image saved as fingerprint_edges.jpg")

if __name__ == "__main__":
    main()
```

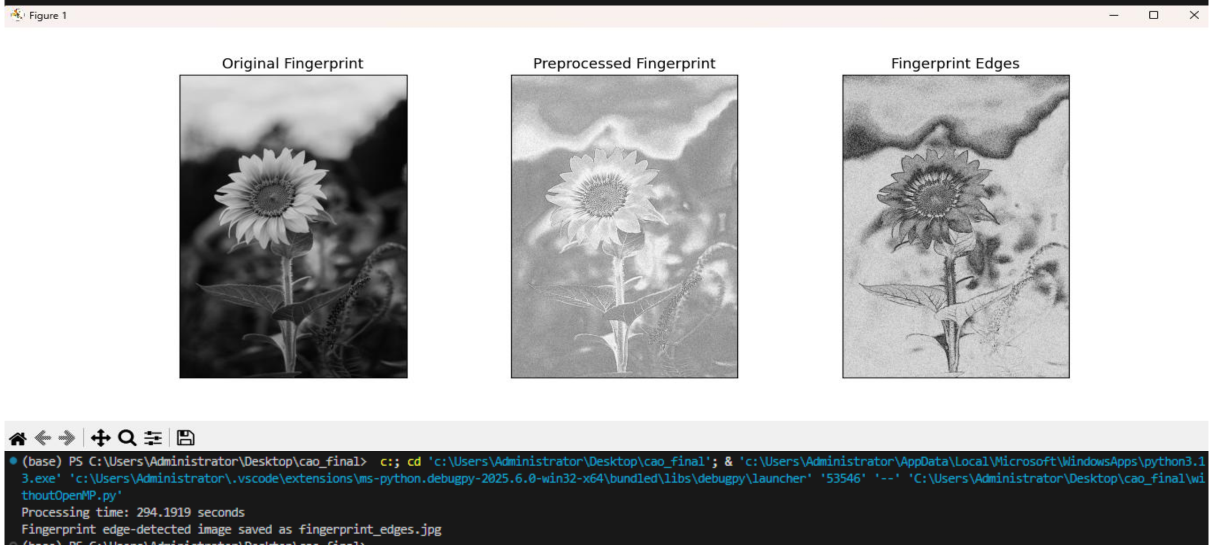
Output along with their Execution Times : (3.3286 s ,226.5476 s ,294.1919 s respectively)



```
(base) PS C:\Users\Administrator\Desktop\cao_final>  c:; cd 'c:\Users\Administrator\Desktop\cao_final'; & 'c:\Users\Administrator\AppData\Local\Microsoft\WindowsApps\python3.1
3.exe' 'c:\Users\Administrator\.vscode\extensions\ms-python.debugpy-2025.6.0-win32-x64\bundled\libs\debugpy\launcher' '53813' '--' 'C:\Users\Administrator\Desktop\cao_final\wi
thoutOpenMP.py'
Processing time: 3.3286 seconds
Fingerprint edge-detected image saved as fingerprint_edges.jpg
(base) PS C:\Users\Administrator\Desktop\cao_final>
```



```
(base) PS C:\Users\Administrator\Desktop\cao_final>  c:; cd 'c:\Users\Administrator\Desktop\cao_final'; & 'c:\Users\Administrator\AppData\Local\Microsoft\WindowsApps\python3.1
3.exe' 'c:\Users\Administrator\.vscode\extensions\ms-python.debugpy-2025.6.0-win32-x64\bundled\libs\debugpy\launcher' '53668' '--' 'C:\Users\Administrator\Desktop\cao_final\wi
thoutOpenMP.py'
Processing time: 226.5476 seconds
Fingerprint edge-detected image saved as fingerprint_edges.jpg
(base) PS C:\Users\Administrator\Desktop\cao_final>
```



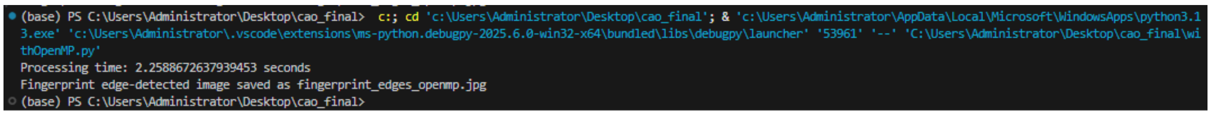
```
(base) PS C:\Users\Administrator\Desktop\cao_final>  c:; cd 'c:\Users\Administrator\Desktop\cao_final'; & 'c:\Users\Administrator\AppData\Local\Microsoft\WindowsApps\python3.1
3.exe' 'c:\Users\Administrator\.vscode\extensions\ms-python.debugpy-2025.6.0-win32-x64\bundled\libs\debugpy\launcher' '53546' '--' 'C:\Users\Administrator\Desktop\cao_final\wi
thoutOpenMP.py'
Processing time: 294.1919 seconds
Fingerprint edge-detected image saved as fingerprint_edges.jpg
```

Parallel Using OpenMP in python

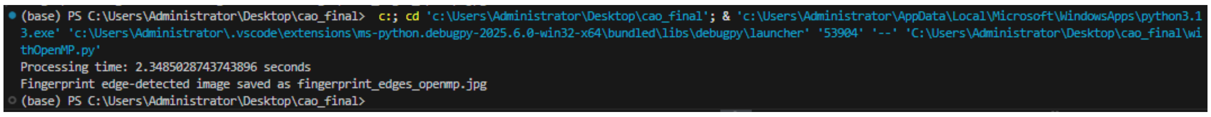
Sample code where OpenMP is implemented

```python
import cv2
import numpy as np
from numba import njit, prange
import time
import matplotlib.pyplot as plt
# Edge detection function with OpenMP parallelism using Numba
@njit(parallel=True)
def edge_detection(image):
    height, width = image.shape
    output = np.zeros((height, width), dtype=np.uint8)
    gx = np.array([[-1, 0, 1],
                   [-2, 0, 2],
                   [-1, 0, 1]])
    gy = np.array([[-1, -2, -1], [0, 0, 0],  [1, 2, 1]])
    # Using parallelism via prange
    for i in prange(1, height - 1):  # Parallelized outer loop
        for j in range(1, width - 1):  # Regular inner loop
            sum_x = 0
            sum_y = 0
            for k in range(3):
                for l in range(3):
                    pixel = image[i + k - 1, j + l - 1]
                    sum_x += gx[k][l] * pixel
                    sum_y += gy[k][l] * pixel
            edge_strength = np.sqrt(sum_x**2 + sum_y**2)
            if edge_strength > 255:
                edge_strength = 255
            output[i, j] = edge_strength
    return output
```

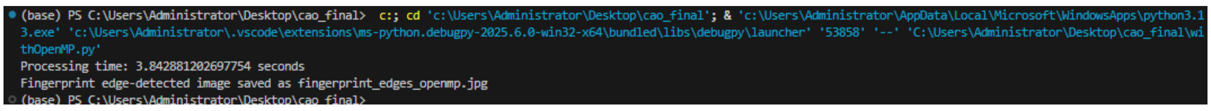
Execution time of same using openMP (2.258867 s ,2.34850 s , 3.84288 s respectively)



```
(base) PS C:\Users\Administrator\Desktop\cao_final>  c:; cd 'c:\Users\Administrator\Desktop\cao_final'; & 'c:\Users\Administrator\AppData\Local\Microsoft\WindowsApps\python3.1
3.exe' 'c:\Users\Administrator\.vscode\extensions\ms-python.debugpy-2025.6.0-win32-x64\bundled\libs\debugpy\launcher' '53961' '--' 'C:\Users\Administrator\Desktop\cao_final\wi
thOpenMP.py'
Processing time: 2.2588672637939453 seconds
Fingerprint edge-detected image saved as fingerprint_edges_openmp.jpg
(base) PS C:\Users\Administrator\Desktop\cao_final>
```



```
(base) PS C:\Users\Administrator\Desktop\cao_final>  c:; cd 'c:\Users\Administrator\Desktop\cao_final'; & 'c:\Users\Administrator\AppData\Local\Microsoft\WindowsApps\python3.1
3.exe' 'c:\Users\Administrator\.vscode\extensions\ms-python.debugpy-2025.6.0-win32-x64\bundled\libs\debugpy\launcher' '53904' '--' 'C:\Users\Administrator\Desktop\cao_final\wi
thOpenMP.py'
Processing time: 2.3485028743743896 seconds
Fingerprint edge-detected image saved as fingerprint_edges_openmp.jpg
(base) PS C:\Users\Administrator\Desktop\cao_final>
```



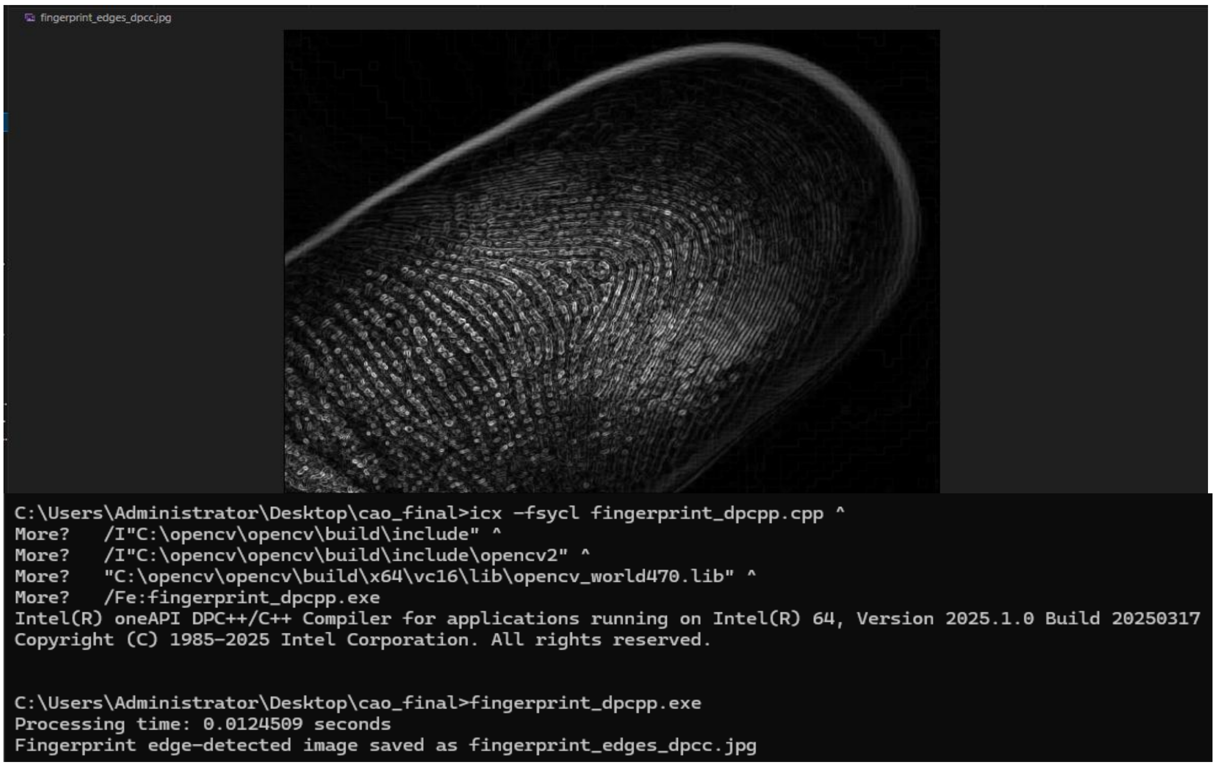
```
(base) PS C:\Users\Administrator\Desktop\cao_final>  c:; cd 'c:\Users\Administrator\Desktop\cao_final'; & 'c:\Users\Administrator\AppData\Local\Microsoft\WindowsApps\python3.1
3.exe' 'c:\Users\Administrator\.vscode\extensions\ms-python.debugpy-2025.6.0-win32-x64\bundled\libs\debugpy\launcher' '53858' '--' 'C:\Users\Administrator\Desktop\cao_final\wi
thOpenMP.py'
Processing time: 3.842881202697754 seconds
Fingerprint edge-detected image saved as fingerprint_edges_openmp.jpg
(base) PS C:\Users\Administrator\Desktop\cao_final>
```

Heterogenous Approach using DPC++

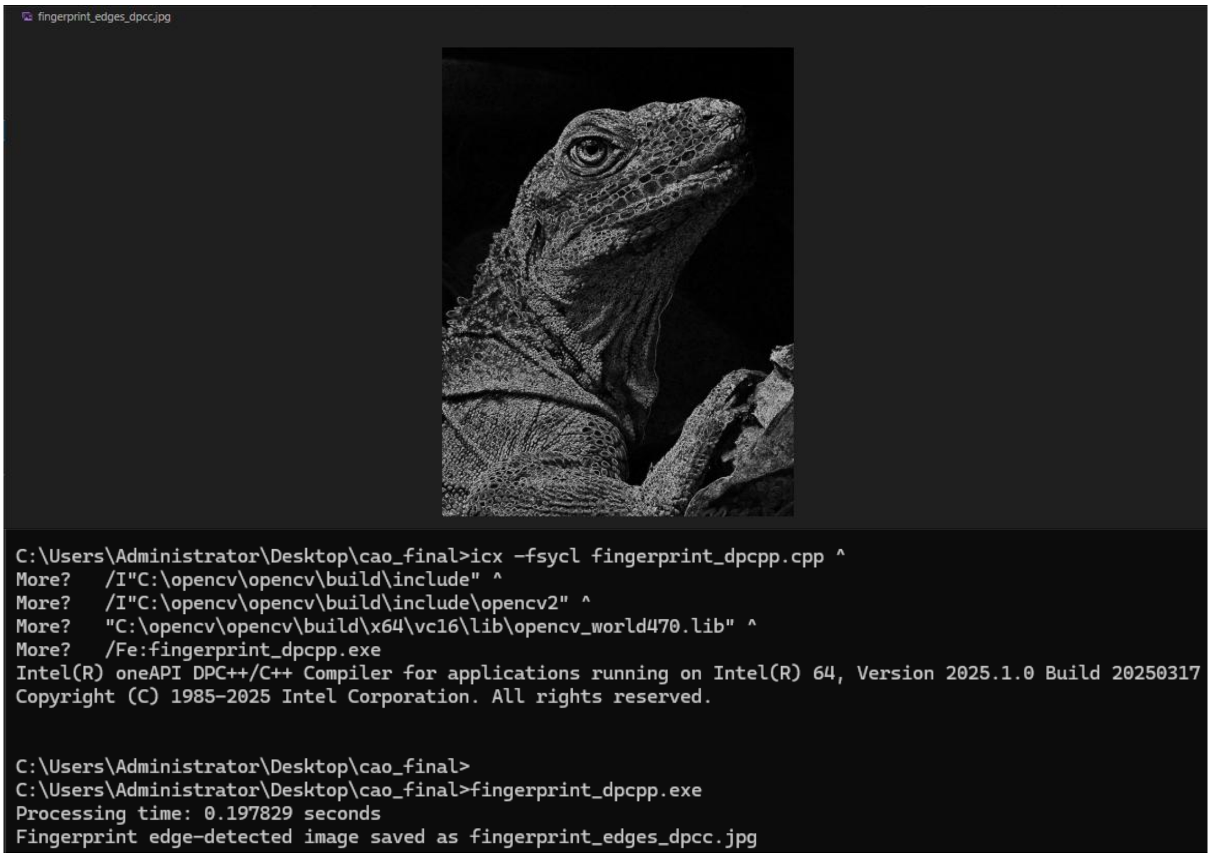
Code :

```
#include <sycl/sycl.hpp>
#include <opencv2/core.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/imgproc.hpp>
#include <iostream>
#include <chrono>
namespace sy = sycl;
using namespace std;
using namespace cv;
int main() {
    // Create a SYCL queue for GPU device using modern selector
    sy::queue q(sy::gpu_selector_v);
 // Start timing
    auto start = chrono::high_resolution_clock::now();
    // Load the fingerprint image in grayscale
    Mat image = imread("fingerprint2.jpg", IMREAD_GRAYSCALE);
    if (image.empty()) {
        cerr << "Error: Unable to load fingerprint image!" << std::endl;
        return -1;
    }
    int width = image.cols;
    int height = image.rows;
    vector<unsigned char> input(image.begin<unsigned char>(), image.end<unsigned char>());
    vector<unsigned char> output(input.size(), 0);
    {
        sy::buffer<unsigned char, 1> in_buf(input.data(), sy::range<1>(input.size()));
        sy::buffer<unsigned char, 1> out_buf(output.data(), sy::range<1>(output.size()));
        q.submit([&](sy::handler& h) {
            auto in = in_buf.get_access<sy::access::mode::read>(h);
            auto out = out_buf.get_access<sy::access::mode::write>(h);
            h.parallel_for(sy::range<2>(height, width), [=](sy::id<2> idx) {
                int y = idx[0];
                int x = idx[1];
                if (x == 0 || y == 0 || x == width - 1 || y == height - 1)
                    return;
                int gx = -in[(y - 1) * width + (x - 1)] - 2 * in[y * width + (x - 1)] - in[(y + 1) * width + (x - 1)]
                         + in[(y - 1) * width + (x + 1)] + 2 * in[y * width + (x + 1)] + in[(y + 1) * width + (x + 1)];
                int gy = -in[(y - 1) * width + (x - 1)] - 2 * in[(y - 1) * width + x] - in[(y - 1) * width + (x + 1)]
                         + in[(y + 1) * width + (x - 1)] + 2 * in[(y + 1) * width + x] + in[(y + 1) * width + (x + 1)];
                int val = sy::clamp((int)sy::sqrt((float)(gx * gx + gy * gy)), 0, 255);
                out[y * width + x] = (unsigned char)val;
            });
        });
    }
    Mat result(height, width, CV_8UC1, output.data());
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double> elapsed = end - start;
    cout << "Processing time: " << elapsed.count() << " seconds" << std::endl;
    imwrite("fingerprint_edges_dpcc.jpg", result);
    cout << "Fingerprint edge-detected image saved as fingerprint_edges_dpcc.jpg" << std::endl;
    return 0;
}
```

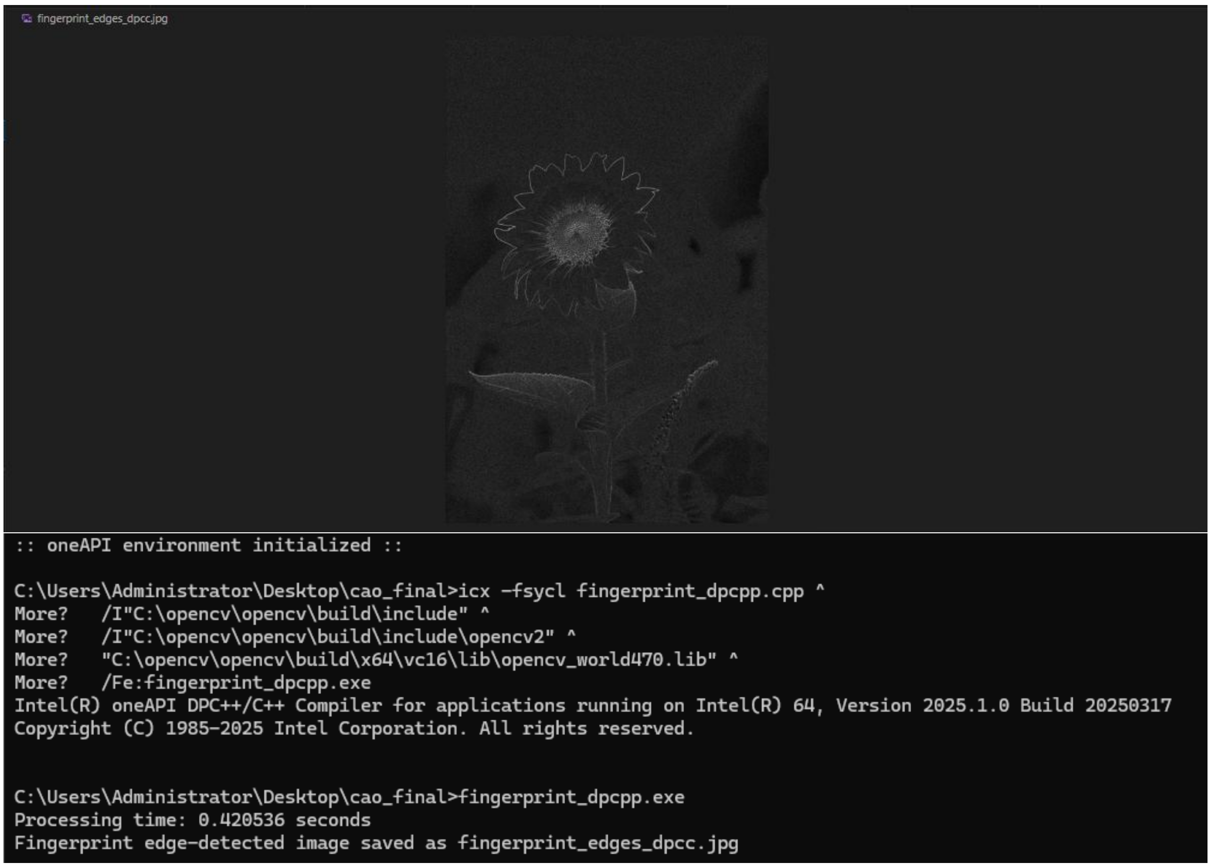
Output with Execution Time – (0.0124509s , 0.197829 s , 0.420536s respectively)



```
C:\Users\Administrator\Desktop\cao_final>icx -fsycl fingerprint_dpcpp.cpp ^
More?   /I"C:\opencv\opencv\build\include" ^
More?   /I"C:\opencv\opencv\build\include\opencv2" ^
More?   "C:\opencv\opencv\build\x64\vc16\lib\opencv_world470.lib" ^
More?   /Fe:fingerprint_dpcpp.exe
Intel(R) oneAPI DPC++/C++ Compiler for applications running on Intel(R) 64, Version 2025.1.0 Build 20250317
Copyright (C) 1985-2025 Intel Corporation. All rights reserved.


C:\Users\Administrator\Desktop\cao_final>fingerprint_dpcpp.exe
Processing time: 0.0124509 seconds
Fingerprint edge-detected image saved as fingerprint_edges_dpcc.jpg
```



```
C:\Users\Administrator\Desktop\cao_final>icx -fsycl fingerprint_dpcpp.cpp ^
More?   /I"C:\opencv\opencv\build\include" ^
More?   /I"C:\opencv\opencv\build\include\opencv2" ^
More?   "C:\opencv\opencv\build\x64\vc16\lib\opencv_world470.lib" ^
More?   /Fe:fingerprint_dpcpp.exe
Intel(R) oneAPI DPC++/C++ Compiler for applications running on Intel(R) 64, Version 2025.1.0 Build 20250317
Copyright (C) 1985-2025 Intel Corporation. All rights reserved.


C:\Users\Administrator\Desktop\cao_final>
C:\Users\Administrator\Desktop\cao_final>fingerprint_dpcpp.exe
Processing time: 0.197829 seconds
Fingerprint edge-detected image saved as fingerprint_edges_dpcc.jpg
```



```
:: oneAPI environment initialized ::

C:\Users\Administrator\Desktop\cao_final>icx -fsycl fingerprint_dpcpp.cpp ^
More?   /I"C:\opencv\opencv\build\include" ^
More?   /I"C:\opencv\opencv\build\include\opencv2" ^
More?   "C:\opencv\opencv\build\x64\vc16\lib\opencv_world470.lib" ^
More?   /Fe:fingerprint_dpcpp.exe
Intel(R) oneAPI DPC++/C++ Compiler for applications running on Intel(R) 64, Version 2025.1.0 Build 20250317
Copyright (C) 1985-2025 Intel Corporation. All rights reserved.


C:\Users\Administrator\Desktop\cao_final>fingerprint_dpcpp.exe
Processing time: 0.420536 seconds
Fingerprint edge-detected image saved as fingerprint_edges_dpcc.jpg
```

Data Aquired

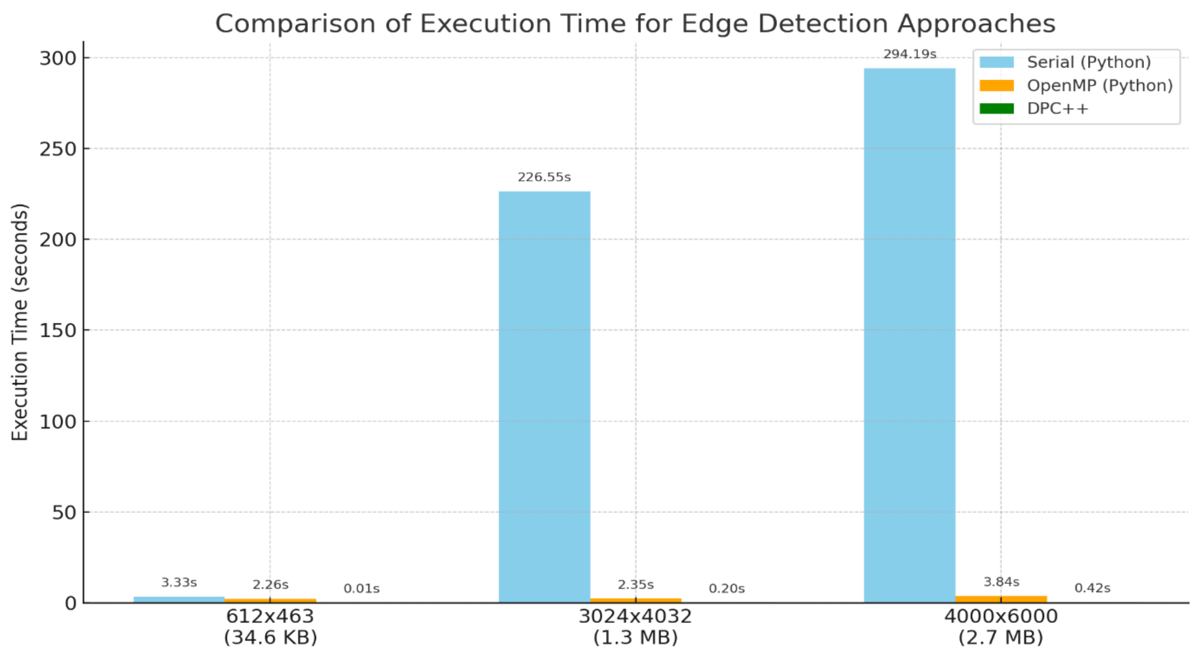
Execution Times :

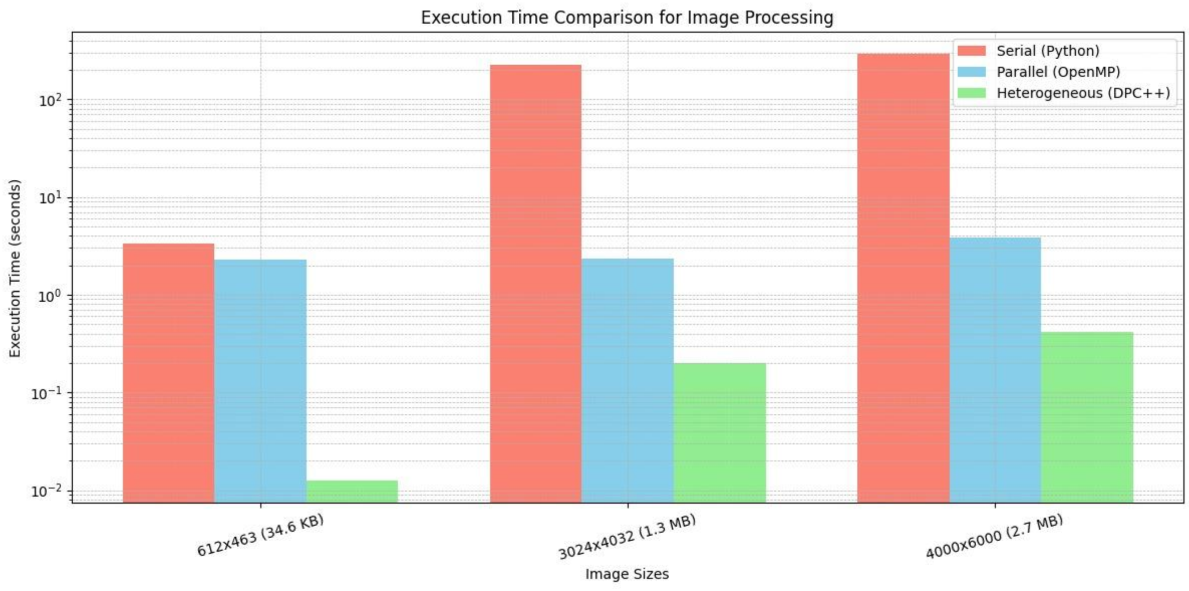
serial using python - (3.3286 s ,226.5476 s ,294.1919 s respectively)
Parallel using openMP - (2.258867 s ,2.34850 s , 3.84288 s respectively)
Heterogenous using dpc ++ – (0.0124509s ,0.197829 s,0.420536s respectively)

Sizes of 3 images used in running the above 3 codes

(612 x 463 - 34.6 KB,3024 x 4032 - 1.3 MB,4000 x 6000 - 2.7MB respectively)

Graph :

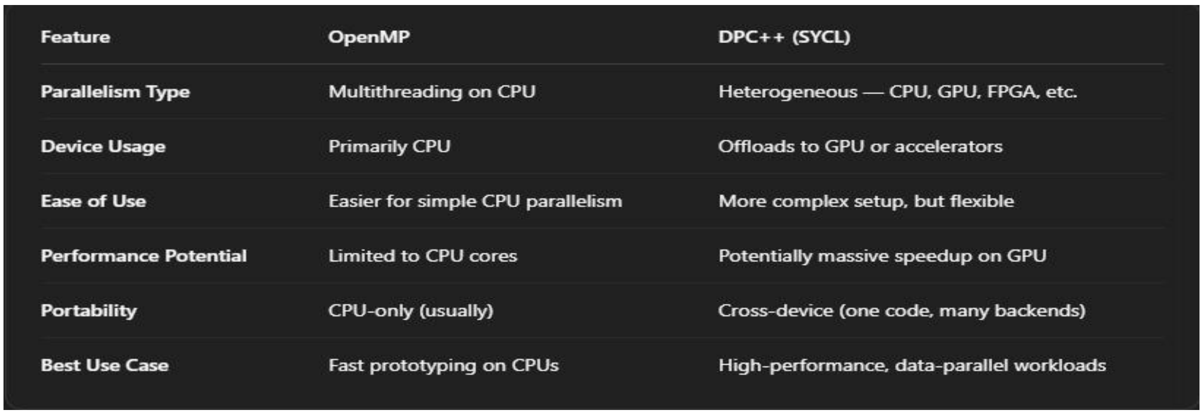




Outcome:

Based on the performance comparison across three image sizes
(34.6 KB, 1.3 MB, and 2.7 MB) , the following key observations are made:

- Serial (Python) implementation shows significantly higher execution time, especially as image size increases.
- OpenMP (Python) provides a considerable improvement over the serial version, maintaining relatively low execution times across all image sizes.
- DPC++ (heterogeneous approach) exhibits the best performance by a large margin, with execution times nearly negligible for small images and still very efficient even for large images.



Feature	OpenMP	DPC++ (SYCL)
Parallelism Type	Multithreading on CPU	Heterogeneous — CPU, GPU, FPGA, etc.
Device Usage	Primarily CPU	Offloads to GPU or accelerators
Ease of Use	Easier for simple CPU parallelism	More complex setup, but flexible
Performance Potential	Limited to CPU cores	Potentially massive speedup on GPU
Portability	CPU-only (usually)	Cross-device (one code, many backends)
Best Use Case	Fast prototyping on CPUs	High-performance, data-parallel workloads

Conclusion:

The experimental evaluation of edge detection across three implementations—Serial (Python), Parallel (OpenMP), and Heterogeneous (DPC++)—demonstrates significant performance differences.
As the image size increased, the limitations of serial execution became evident with a sharp rise in execution time.
OpenMP in Python provided a substantial improvement in speed by leveraging multi-core CPU parallelism. It consistently reduced execution times compared to the serial version, especially for medium-sized images, proving to be a practical enhancement for CPU-bound systems.
However, the most remarkable performance gain was observed using DPC++, which achieved the lowest execution times even for large images. This is attributed to DPC++'s ability to exploit heterogeneous computing environments, including GPU acceleration. The execution time improvements

confirm that GPU-assisted parallelism offers superior performance for computationally intensive image processing tasks.
Therefore, DPC++ stands out as the most efficient method among the three for large-scale edge detection, validating the advantages of heterogeneous computing in high-performance applications.

References

Gonzalez, R. C., & Woods, R. E. (2008). *Digital Image Processing* (3rd ed.). Pearson Education.

Szeliski, R. (2010). *Computer Vision: Algorithms and Applications*. Springer.

Stallings, W. (2020). *Computer Organization and Architecture: Designing for Performance* (11th ed.). Pearson.

Kirk, D. B., & Hwu, W.-M. W. (2016). *Programming Massively Parallel Processors: A Hands-on Approach* (3rd ed.). Morgan Kaufmann.

Chapman, B., Jost, G., & van der Pas, R. (2007). *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press.

Dagum, L., & Menon, R. (1998). OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5(1), 46–55.

Intel Corporation. (2022). *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*. Springer.

Gao, M., Song, Y., & Li, H. (2020). Performance Comparison of CPU and GPU for Image Processing Applications. *Journal of Computer Science and Technology*, 35(1), 22–32.

Tanase, M., & Suciu, G. (2020). Parallel Processing Approaches for Edge Detection Algorithms. *Procedia Computer Science*, 176, 2787–2796.

Marin, G., & Mellor-Crummey, J. (2004). Cross-architecture performance predictions for scientific applications using parameterized models. *Proceedings of the Joint ACM SIGMETRICS/Performance Conference*, 2–13.

OpenMP Architecture Review Board. (2023). *OpenMP Application Programming Interface Version 5.2*. Retrieved from https://www.openmp.org/specifications/

Intel Corporation. (2023). *Intel oneAPI Toolkits Documentation*. Available at:

https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html
Gupta, M., Dinesh, S., & Singh, P. (2023). Accelerating Image Processing Using OpenMP in Python. *International Journal of Scientific Research in Computer Science*, 11(3), 45–51.
Xiong, J., & Chen, Y. (2021). Optimization of Edge Detection Algorithms using GPU Acceleration in DPC++. *IEEE Access*, 9, 89998–90006.
Hasan, R., & Sharma, P. (2022). A Comparative Study on Image Edge Detection Techniques Using Python. *International Journal of Computer Applications*, 184(17), 1–6.
Realix Software. (2024). *HWiNFO – Comprehensive Hardware Analysis, Monitoring and Reporting Tool*. Retrieved from https://www.hwinfo.com/
Realix Software. (2023). *HWiNFO User Manual and Documentation*. Available at: https://www.hwinfo.com/documentation/
TechPowerUp. (2023). *HWiNFO: System Monitoring and Diagnostics*. Retrieved from https://www.techpowerup.com/download/hwinfo/