Image processing Accelerator

江智端
311551007
資科工碩一
m897420@gamil.com

王昇暉
311551144
資科工碩一
sam1211229068@gmail.com

楊宗泰
311553046
多媒體碩一
jimmywen6666@gmail.com

ABSTRACT

We developed an image-processing accelerator that utilizes parallel programming techniques to enhance convolution performance in image processing. By dividing the image into smaller sections and processing each concurrently on separate processors or devices, we can distribute the workload and decrease the total processing time. This significantly increases the speed of image processing, making it more efficient and effective.

1 INTRODUCTION

Digital image processing uses a digital computer to process digital images through an algorithm. As a subcategory or field of digital signal processing, digital image processing has many advantages over analog image processing. It allows a much more comprehensive range of algorithms to be applied to the input data and can avoid problems such as the build-up of noise and distortion during processing.

Since images are defined over two dimensions (perhaps more), digital image processing may be modeled as a multidimensional system. The generation and development of digital image processing are mainly affected by three factors:

1. The development of computers.
2. The development of mathematics (especially the creation and improvement of discrete mathematics theory).
3. The demand for a wide range of applications in environment, agriculture, military, industry, and medical science has increased.

2 MOTIVATION

The main problem is that image processing is generally a time-consuming process. Parallel Computing provides an efficient and convenient way to address this issue.

Our teammate found a scenario during the internship. In his company, they will use a series of image processing methods to test the defect of the wafer. And the validation machine needs to be more bulky and unportable; it also takes a lot of time due to its old system and serial programming. We can build some image processing tools in a parallel programming way. It can be easily maintained and portable, and we can install the system on small FPGAs like NVIDIA JETSON.

Our primary purpose of this research is to analyze different parallel programming ways such as SIMD, CUDA, Pthreads, OpenMP …etc. Versus serial way. We want to know which one will be the best to improve image processing and that many applications of image processing can be faster.

3 PROPOSED SOLUTION

Convolution is a crucial technique in image processing that involves using a kernel, or a small matrix of numbers, to modify

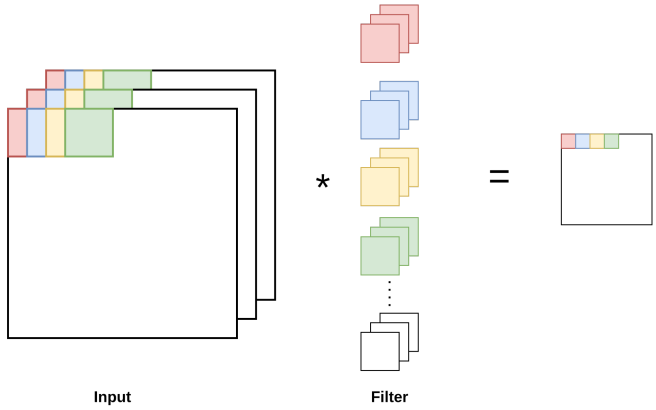

Figure 1: The simple example of coevolution

the input image and generate an output feature map. As depicted in Fig. 1, the process involves multiplying the input image's values by the kernel's values and summing the results to create a new deal for each pixel in the output feature map. The resulting feature map is a transformed version of the original image, with the changes determined by the kernel used in the convolution. This process is applied to every pixel in the image to generate the final output feature map.

Our target is to parallelize the convolution process using several different methods, including SIMD, Pthreads, OpenMP, and CUDA, and evaluate their performance on different image sizes. By parallelizing the convolution, we aim to increase the speed and efficiency of image processing. We will measure the changes in program size under different image sizes to determine the effectiveness of each method.

3.1 SIMD

SIMD, or Single Instruction Multiple Data, is a powerful tool for parallel programming that allows multiple data elements to be processed simultaneously with a single instruction. In image processing, SIMD can store and process various pixels concurrently, improving the speed and efficiency of convolution. As shown in Fig. 2a, SIMD uses a vector to store and calculate eight output pixels simultaneously, compared to the baseline method, which only processes one pixel at a time. By leveraging SIMD, we can significantly increase image processing speed, making it more efficient and effective.

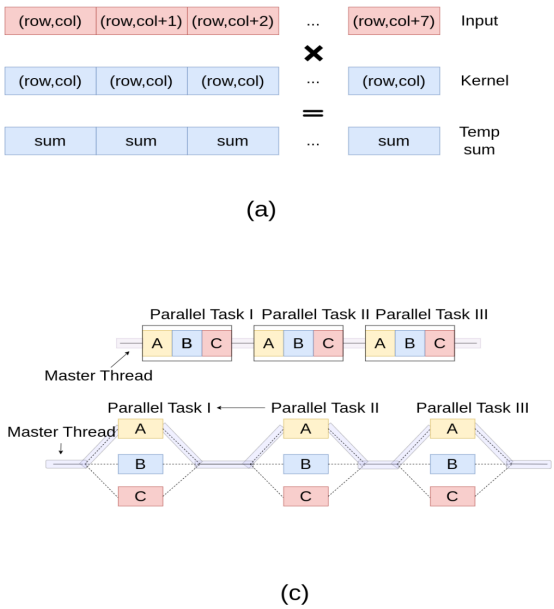

Figure 2: (a) SIMD (b) Pthreads (c) OpenMp (d) CUDA

3.2 Pthreads

Pthreads is a reliable method for implementing threads in a program and can be particularly useful in image processing. By dividing the image into smaller sections and assigning each paragraph to a separate thread, we can concurrently process the image and distribute the workload. As shown in Fig. 2b, if we divide the image into 16 parts by its height, each thread can be responsible for processing one part. By using Pthreads to parallelize the image processing, we can significantly improve the speed and efficiency of the process.

3.3 OpenMP

OpenMP is a valuable tool for parallel programming in C, C++, and Fortran. Image processing can divide the workload and process it concurrently on multiple processors or devices. As shown in Fig. 2c, OpenMP can apply convolution by sharing the sum of the output pixels and calculating each partial sum in parallel. The main thread can then collect the final result. By leveraging OpenMP, we can significantly improve the speed and efficiency of image processing by distributing the workload and reducing the overall processing time.

Responsible for processing one part. By using Pthreads to parallelize the image processing, we can significantly improve the speed and efficiency of the process.

3.4 CUDA

CUDA is a highly effective platform and programming model for parallel computing on GPUs. Image processing can divide the image into smaller sections and concurrently process each section on the GPU. As shown in Fig. 4, the image can be separated into elements row by column, with each element assigned to a separate thread. With 1024 threads available, each can be responsible for processing a single element. By implementing CUDA to parallelize image processing, we can significantly improve the speed and efficiency of the process by distributing the workload and reducing the overall processing time.

Additionally, we will use the OpenCV library function cv::filter2D to compare the performance of different methods.

Based on our experience and the GPU speed, we expect the order of speed to be as follows: cv::filter2D > CUDA > OpenMP > Pthreads > SIMD > Serial. We will use these methods to parallelize the convolution process and evaluate their performance on various image sizes.

4 EXPERIMENTAL METHODOLOGY

The platform for our experiment is as follows:

- CPU : AMD Ryzen 7 5800X (8 cores 16 threads)
- GPU : NVIDIA GeForce RTX 3080
- OS : Window11
- OpenCV : 4.6.

The details of our dataset are shown in the following table:

Data	Size
Input data	2M~100M pixels
Filter kernel	3x3, 5x5, 7x7, ... , 25x25

We will implement each parallelization method in C++ and design a programming model to ensure consistent I/O across all implementations. This will allow us to accurately compare the performance of each way on the same input data and evaluate their effectiveness in optimizing the speed and efficiency of image processing.

5 EXPERIMENTAL RESULTS

Our experiment aims to compare the performance of different methods for parallelizing convolution on various input images and kernel sizes. We will also examine the impact of fixing the kernel size and varying the input size on the performance of the parallelized process. By conducting these experiments, we hope

to gain insight into the most effective approaches for optimizing the speed and efficiency of image processing through parallelization.

5.1 Exploring the Influence of Different Input Sizes on Convolution Performance

In this experiment, we fixed the kernel size and observed the impact of different input sizes on parallel acceleration. As shown in Fig. 3, the results showed that CUDA was the fastest, achieving a 13x speedup over the serial version and outperforming the built-in cv::filter2D function. The other methods followed the expected sorting. As the input size increased, the execution time exhibited close to linear growth. This suggests that a simple formula may be used to predict the future acceleration of parallelization for convolution.

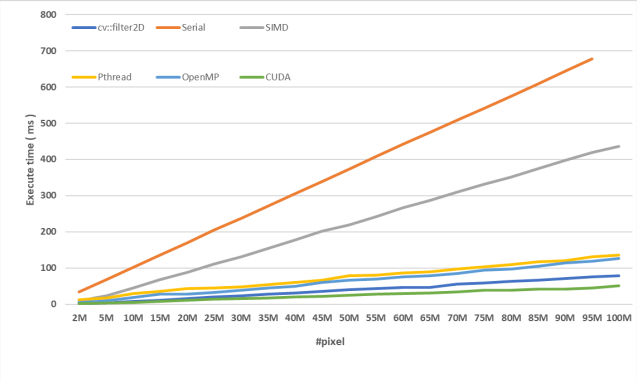

Figure 3: Evaluating the Impact of Input Size on Convolution Performance Using a 3x3 Filter

5.2 Exploring the Influence of Different kernel Sizes on Convolution Performance

The results in figures 4-6 show that different kernel sizes had similar impacts on convolution performance for various input sizes. The serial implementation had significantly longer execution times as the kernel size increased. In contrast, parallelization using CUDA was up to 349 times faster than the serial version, demonstrating its effectiveness in maintaining stable execution times despite changes in kernel size.

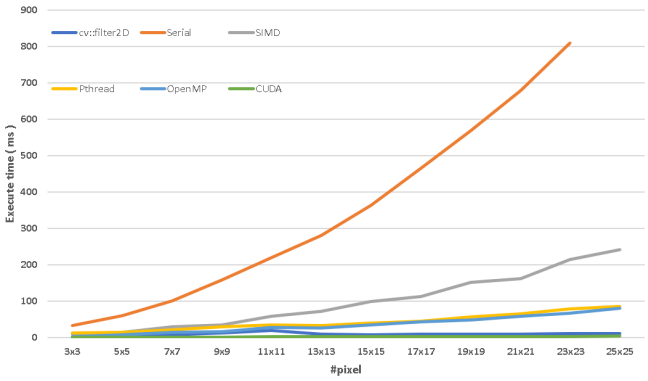

Figure 4: Evaluating the Impact of kernel Size on Convolution Performance Using a 2M Input size

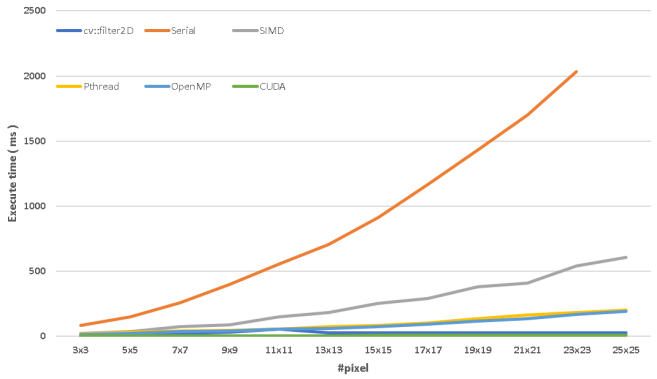

Figure 5: Evaluating the Impact of kernel Size on Convolution Performance Using a 20M Input size

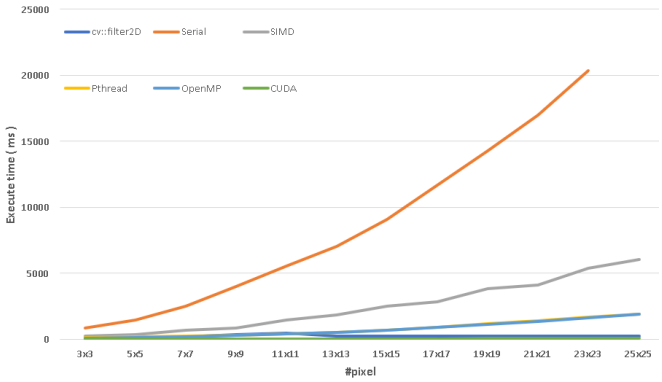

Figure 6: Evaluating the Impact of kernel Size on Convolution Performance Using a 50M Input size

Our experiments showed that CUDA was the clear winner regarding speed for image processing tasks. Its design specifically for parallel computation on GPUs, which have more processing power and memory bandwidth than CPUs, gives it a significant advantage over cv::filter2D. In addition, the fine-grained control offered by CUDA allows for optimal performance. On the other hand, cv::filter2D is a software function that could be optimized for parallel computation and offer a different level of control.

Additionally, our experiments showed that SIMD did not perform as well as the other methods. Since we only processed eight elements at a time, CUDA and Pthreads could handle 1024 and 16 elements simultaneously, respectively. This small number of pixels may only partially utilize SIMD's single-instruction, multiple-data approach. To fully leverage SIMD, it may be necessary to unroll more instructions in the implementation.

7 CONCLUSIONS

In conclusion, our experiments showed that a suitable parallelization method is crucial for adequate image processing acceleration. We found that the input and kernel sizes had little influence on performance when using a solid parallelization method, such as CUDA. CUDA consistently demonstrated the best performance among the tested methods, making it an ideal choice for image processing acceleration.

8 REFERENCE

[1]https://www.allaboutcircuits.com/technical-articles/two-dimensional-convolution-in-image-processing/

[2]https://docs.opencv.org/4.6.0/index.html