Parallel Programming Final Project
Gaussin Blur, Different Parallel Platforms Comparison

HWU, AN-FONG
National Chiao Tung University
Undergraduate, Department of Computer Science
alfons.cs04@g2.nctu.edu.tw

LI, HUNG-YANG
National Chiao Tung University
Graduate, Department of Computer Science
chc71340@gmail.com

CHEN, YI-FENG
National Chiao Tung University
Undergraduate, Department of Computer Science
randomfrank8@gmail.com

ABSTRACT

In this project, we aim to focus on accelerating the well-known Gaussian Blur Algorithm, which is a widely used image processing algorithm to smooth and blur images. We use the self-written image IO handler to read in the BMP image file and store the RGB channel value respectively, after successfully read the images, the parallel algorithm can proceed, then write back the blurred, processed image file for correctness validation.

Besides the normal parallelization done on CPU only, heterogeneous computation like CUDA and OpenCL are also implemented. With the aforementioned platforms, we may compare their performance and conclude.

INTRODUCTION

[1] In image processing, a Gaussian blur (also known as Gaussian smoothing) is the result of blurring an image by a Gaussian function (named after mathematician and scientist Carl Friedrich Gauss). It is a widely used effect in graphics software, typically to reduce image noise and reduce detail. The visual effect of this blurring technique is a smooth blur resembling that of viewing the image through a translucent screen, distinctly different from the bokeh effect produced by an out-of-focus lens or the shadow of an object under usual illumination. Gaussian smoothing is also used as a pre-processing stage in computer vision algorithms in order to enhance image structures at different scales—see scale space representation and scale space implementation.

Mathematically, applying a Gaussian blur to an image is the same as convolving the image with a Gaussian function. This is also known as a two-dimensional Weierstrass transform.

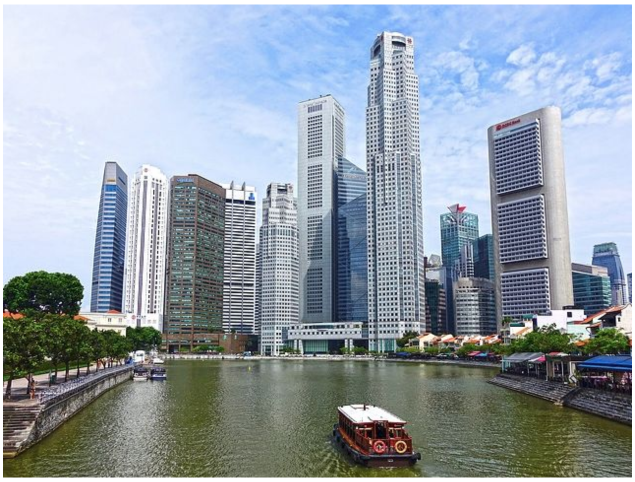


Figure. Original image.

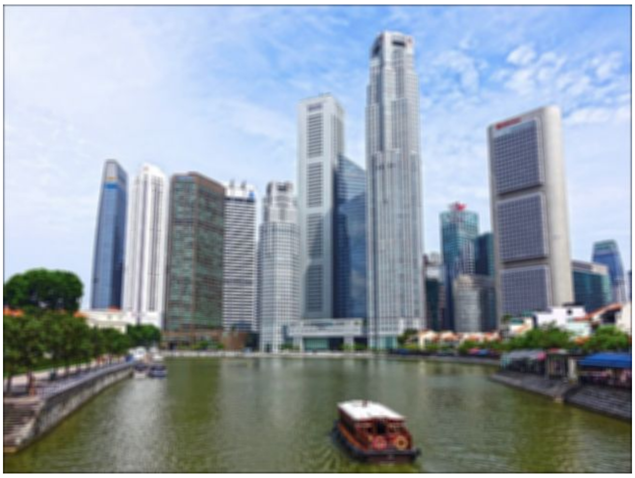


Figure. Gaussian Blur with kernel size 5x5

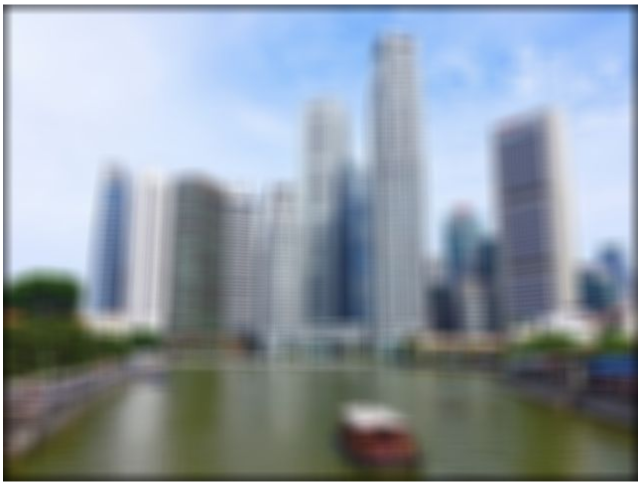
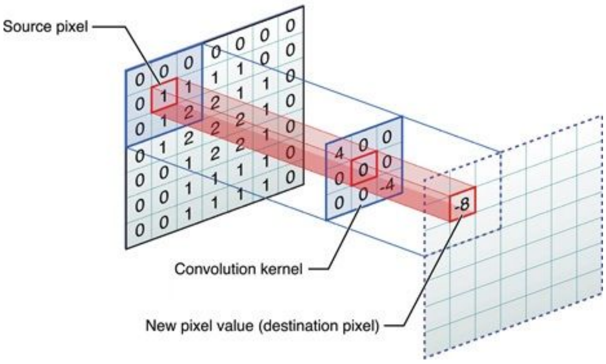


Figure. Gaussian Blur with kernel size 21x21

MOTIVATION

In image processing, gaussian blur is a widely used algorithm in lots of software like Photoshop, GIMP. Typically, it tries to reduce image noise and image detail.
But unfortunately, applying Gaussian blur algorithm is tremendously hard work for calculation. It should calculate the square of nearby filter size pixel for each selected pixel. So the time complexity will be large to O(img_width * img_height * filter_size^2).



[2] Figure. Visualization of Gaussian kernel convolution

EXPERIMENT (BENCHMARK) METHODOLOGY

Hardware

CPU	Intel Core i5-7500
GPU	NVIDIA GTX1070 (1920 CUDA cores)
SSD	Samsung PM981 512GB NVMe
RAM	DDR4 2666 8G x 4
OS	Ubuntu 16.04 64bit

Testing Image. Our image for processing is an 8K image, with size being 7680 * 4320.

Figure. 8K Source image for the benchmark.

Filter Size. Filter size is fixed to 101 * 101 and generated according to two-dimensional Gaussian Distribution.
Computation Amount. 338 billion (3e+12)
Setting and Compared Methods. We feed our testing image to all of the platforms we designed, which is OpenMP, CUDA and OpenCL. Based on each output of three platforms, we compared each blurred image with the original one. However, due to the issue of precision of CUDA and OpenCL on GPU, we designed a validator to show our error in each blurred image.

Moreover, our validating program compares each pixel in origin image with the blurred one on three channel (R, G, B). If anyone channel fails in our test, it will be recorded as an error pixel.
Baseline: Serial version 82m44s (4964sec)

PLATFORMS AND SOLUTIONS

OpenMP: [3] OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming. It uses a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the standard desktop computer to the supercomputer.

OpenCL: OpenCL (Open Computing Language) is the open, royalty-free standard for cross-platform, parallel programming of diverse processors found in personal computers, servers, mobile devices, and embedded platforms.

CUDA: [4] CUDA (Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) model created by Nvidia. It allows software developers and software engineers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing — an approach termed GPGPU (General-Purpose computing on Graphics Processing Units). The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels.

First, for OpenMP, simply and automatically parallelize by using some `#pragma` comments in the source code. Since the CPU for benchmarking is the Intel Core i5-7500 (4 cores, 4 threads). By using the OpenMP pragma (`#pragma omp parallel for`), the image can be segmented into four parts for paralleling, and thus one part for a thread respectively.

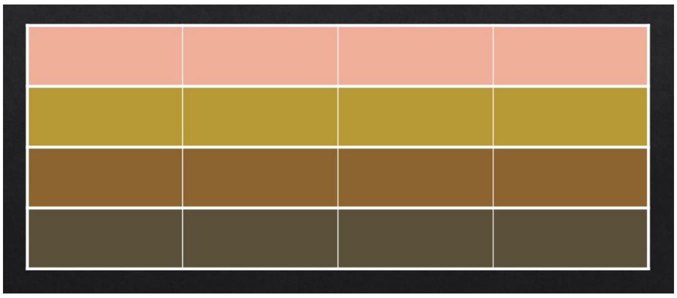


Figure. OpenMP threads explained

Second, the heterogeneous computing architecture including CUDA and OpenCL. Heterogeneous computing refers to systems that use more than one kind of processor or cores. These systems gain performance or energy efficiency not just by adding the same type of processors, but by adding dissimilar coprocessors, usually incorporating specialized processing capabilities to handle particular tasks.

For CUDA and OpenCL, the CPU is called host processing unit and memory is called host memory(DRAM, DDRX) while GPU is called device processing unit and memory is called device memory (VRAM, **G**DDRX). Both of them need first, load the image from host memory to device memory, then move back from device to host after successfully processing image (which sometimes the performance bottleneck may lies here due to hardware limitations such as memory bandwidth.)

In OpenCL, we use 250 threads per processing block and each thread deals with the image processing of one pixel on the image, the 250 threads are self-adjusted and configured on the runtime, thus we do not need to take care of how many threads for each processing unit. In the benchmark image, this will result in a total amount of (4320 * 7680 * 3(RGB channel)/ 250) = 398130 blocks queued to be executed.

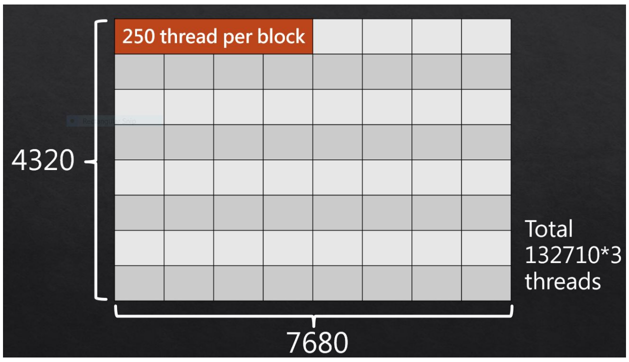


Figure. OpenCL threads explained.

In CUDA, 1024 threads can be executed simultaneously, and same as OpenCL where each thread deals with one pixel on the image. What's different from OpenCL is that how many threads can be executed at the same time are defined by the [5] **CUDA Computing Capability(CUDA-CC)**, in GTX1070 with CUDA-CC 6.1, at most 1024 threads can be executed simultaneously while in TITAN V with 7.1, 2048 of that will be capable of being executed at the same time. Now, a total of 4320 * 7680 * 3(RGB channel)/ 1024 = 97200 blocks queued to be executed.

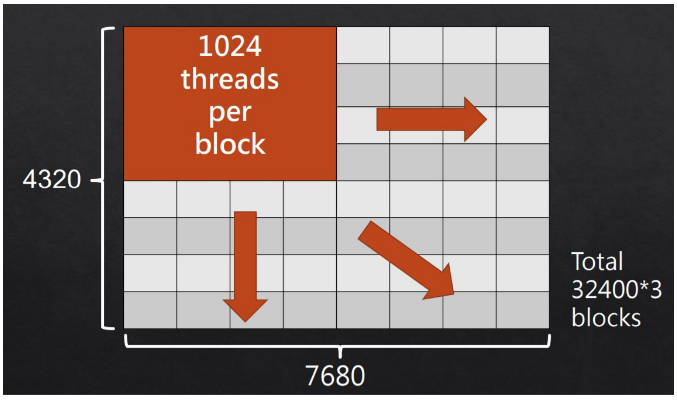


Figure. CUDA threads explained

RESULT OF RESPECTIVE PLATFORM

Metric. We use the executing time to measure our platform performance.

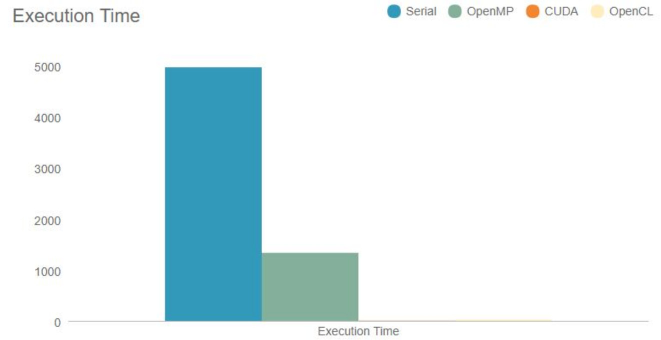


Figure. Execution time comparison (CUDA and OpenCL is too slow to see in the scale)

We first compare our parallel method with serial one. We then see a great enhancement on the program's efficiency.

Platform	Execution Time(sec)
Serial	4964
OpenMP	1332
CUDA(1024 threads)	6.33
OpenCL	5.31

Since OpenMP just use CPU to accelerate pixel processing. It only has about 4 times speedup. CUDA and OpenCL provide heterogenous computation ability to utilize GPU as out accelerating tool. With tremendous thread pool in GPU, we can have a strong speedup in both platforms.
However, due to the portability issue, OpenCL performs worse than CUDA.

Figure. The result of OpenMP platform, accelerate 4x

Figure. The result of CUDA platform, accelerate 787x

Figure. The result of OpenCL platform, accelerate 224x

By using the image comparator, the OpenMP version is identical with the original image due to its execution on CPU. CUDA and OpenCL result in a 1% error (error pixels / total pixels). Nevertheless, this error is too tiny to notice due to, first, we compare R G B respectively, which is rather strict, second, the error is almost unnoticeable in such large and blurred image.

EXTRA TOPIC: IMPROVE CUDA WITH CACHED CONSTANT MEMORY

Can we make CUDA faster? First of all, we tried to focus on the relationship between threads per block and time and the results are as follow. Obviously, the 1024 threads reach the peak performance.

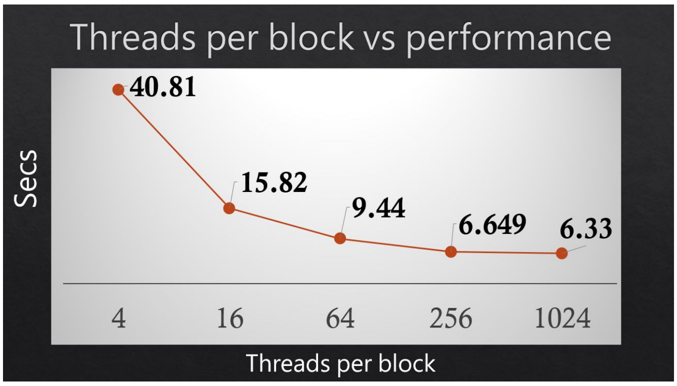


Figure. CUDA threads per block and corresponding performance on the same 8K image.

Hence, we tend to focus on modifying the memory architecture, and here comes the **Constant Memory.** Constant Memory (CM) is a memory with short latency, high bandwidth. It is read-only for the device and host can read or write data on it. CM is closer to the stream processor in GPU (or thread) compared to the standard global memory, acting like the cache memory in CPU. Since filter array is frequently and repeatedly accessed, we may cache it in the constant memory to improve the performance, with results as follow.

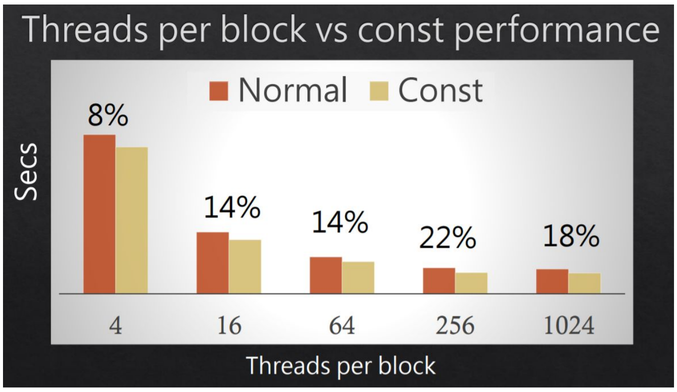


Figure. CUDA threads per block and corresponding performance on the same 8K image with CM enabled (The percent number is performance improvement compared to that without CM)

The fastest combination will be **1024 threads per block with constant memory**, yielding the execution time reduced to 5.31s, accelerate rate boosted from 787x to the surprisingly 935x, which is quite a significant improvement.

CONCLUSION

The best parallel platform does not exist, each one gets its own characteristic. For the best power on NVIDIA GPU, CUDA is favored since it is dedicated for NVIDIA and proprietary architecture. For cross-platform portability on GPU, use OpenCL instead since it is an open standard, not tuned or optimized for a certain architecture. For the simplest modification on the source code, OpenMP will be the choice, just simply add #pragma on some non-data hazardous loop for unrolling, the accelerating effect is right to go.

FUTURE WORK

Try to separate x and y-axis because Gaussian filter is separable.
Gaussian filter is

$$g(x,y) = \frac{1}{2\pi\sigma^2} \sum_{i=-N}^{N} \sum_{j=-N}^{N} e^{-(\frac{i^2+j^2}{2\sigma^2})} f(x+i,\ y+j)$$

$$= \frac{1}{2\pi\sigma^2} \sum_{i=-N}^{N} e^{-(\frac{i^2}{2\sigma^2})} \sum_{j=-N}^{N} e^{-(\frac{j^2}{2\sigma^2})} f(x+i,\ y+j)$$

Let $h(x,\ y) = \frac{1}{\sqrt{2\pi}\cdot\sigma} \sum_{j=-N}^{N} e^{-(\frac{j^2}{2\sigma^2})} f(x,\ y+j)$, then

$$g(x,\ y) = \frac{1}{\sqrt{2\pi}\cdot\sigma} \sum_{i=-N}^{N} e^{-(\frac{i^2}{2\sigma^2})} h(x+i,\ y).$$

So Gaussian filter can be done in $O(img\_width \times img\_height \times filter\_size)$, which is asymptotically more efficient than our proposed algorithm. First do Gaussian filter on the y-axis, then on the x-axis.

Pad image boundary and remove bound check code
Because Gaussian filter can reference to pixels outside the image, our code must do boundary checking. When we pad the image enough, out program no longer accesses to outside the image, thus removing the need of bound check code and (maybe) making the program run faster. (Need validation in the future)

Use SIMD parallelization
SIMD is a kind of data parallelism. [6]It describes computers with multiple processing elements that perform the same operation on multiple data points simultaneously. Such machines exploit data-level parallelism, but not concurrency: there are simultaneous (parallel) computations, but only a single process (instruction) at a given moment. SIMD is particularly applicable to common tasks such as adjusting the contrast in a digital image or adjusting the volume of digital audio. Most modern CPU designs include SIMD instructions to improve the performance of multimedia use where Gaussian Blur is also an multimedia algorithm.

MORE TEST DATA AND RESULTS

Please refer (https://docs.google.com/spreadsheets/d/1Z3RHz_0p12b4tstwzZUZJe4MMb9etsUpkl_rjo_UKAY/edit?usp=sharing) to find more results.

Please refer (https://www.youtube.com/watch?v=DIqVXbUo7EE) for the demo video. (Recorded with SimpleScreenRecord on Linux and editing with Adobe Premiere Pro)

SOURCE CODE LINK

https://github.com/Alfons0329/Parallel_Programming_Fall_2018/tree/master/Final%20Project

REFERENCES

[1] Gaussian Blur in Wikipedia (https://en.wikipedia.org/wiki/Gaussian_blur)
[2] Apple Developer Site (https://developer.apple.com/library/archive/documentation/Performance/Conceptual/vImage/ConvolutionOperations/ConvolutionOperations.html)
[3] OpenMP in Wikipedia (https://en.wikipedia.org/wiki/OpenMP)
[4] CUDA in Wikipedia (https://en.wikipedia.org/wiki/CUDA)
[5] CUDA Compute Capability from NVIDIA (https://developer.nvidia.com/cuda-gpus)
[6] SIMD in Wikipedia (https://en.wikipedia.org/wiki/SIMD)
OpenCL Official Website (https://www.khronos.org/opencl/)
CUDA Occupancy calculator (https://devtalk.nvidia.com/default/topic/368105/cuda-occupancy-calculator-helps-pick-optimal-thread-block-size/)