Gaussian Blur for Sequential and Parallel Algorithms

PARALLEL PROCESSING FOR GPU IMAGE FILTERING

A PREPRINT

Jason Castillo **Carie Navio**

December 7, 2023

ABSTRACT

In this paper, we develop and evaluate a sequential and GPU parallel blur filter solution. We provide the background in filtering as well as a high-level overview of two low-pass image blurring filters, Gaussian and Average. The experimental results and performance metrics, such as execution time and speedup, provide insights into the benefits of GPU acceleration for Gaussian blur operations. This paper demonstrates the feasibility and performance gains of GPU-accelerated Gaussian blur algorithms.

***K*eywords** CUDA, Gaussian Filter, Parallelization, Performance Metrics, Concurrency, Optimization Problems, GPU, Image Blur Processing, Scalable Solutions, Sequential Algorithms

1 INTRODUCTION

Image blur is a fundamental operation in computer vision and image processing, with applications in various domains such as photography, computer graphics, and medical imaging. A substantial application for object research is in the form of facial and object recognition. From individuals using FaceID to unlock their phone, to governments using Clearview AI to apprehend suspected criminals, the ability to better analyze imperfect images is becoming more prevalent. In recent years, the use of mixed training on blurred and sharp images has helped to better train convolutional neural networks (CNNs) on object recognition [6]. By varying the level of blur, image-computing models can better advance the ability for CNNs to perform like the human visual system in terms of global shape information processing. One of the major limitations within computer vision and image processing is within the computational availability and resource intensity needed to perform these tasks. Not only is substantial data needed to train machine learning, the act of processing the data itself can be time-intensive. Using the newest NVIDIA graphics card, we implement GPU acceleration on images, discuss the performance achievable at a small scale, where bottlenecks exist, and how to potentially mitigate those blockers.

2 GROUNDWORK

2.1 Filtering

Image blurring occurs through the use of low-pass filtering on an image source. A low-pass filter allows for frequencies below the threshold to maintain their value while gradually removing the intensity of the frequencies above the threshold. Alternatively, a high-pass filter minimizes low frequencies. High-pass filtering is used to increase the sharpness of an image using similar techniques. A mixture of a low-pass and high-pass filter is called a band-pass filter.

A real-world example of a low-pass filter is in a sub-woofers. Sub-woofers are specifically designed to minimize higher frequency noises in order to better output lower sounds. In image filtering, these noises are related to random variation in brightness or color generated by photograph pixels. Figure 1 is an example of noise that occurs in photographs where you can see random peppered out of context pixels. These noises are a combination of the sensor in the camera, resolution of the photo, and light sensitivity. Image blurring filters will take these noises and perform operations so that the difference in pixel colors is less abrupt and removes this peppered effect. The low-pass filters discussed within this paper are Gaussian and Median blur techniques.

Figure 1: Example of image noise [9]

2.1.1 Filter Kernels

The way the blur effect is produced is by altering a pixels in relations to to the pixels around it. For instance, the average filter blur works by taking pixel values surrounding the pixel being operated on and then taking an average of those pixel to generate a new image as seen in the kernel filter figure. Then these kernels can have other weights applied to them to accomplish Gaussian blur or other types of filters. This works for black and white images and to process color images we just need to consider the RGB colors individually. The pixel filter will be applied to each red, green and blue and then added back to the output image to generate the final filtered image.

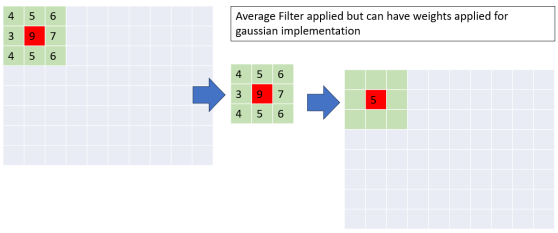

Figure 2: Kernel Visualization

2.1.2 Average/Medium

For simplicity of understanding kernels is how the average of surrounding pixels generates a new pixel. This produces a new image but now it generates artificial pixels as the average value was not present in the original image. The next level of improving image blurring is medium blur where instead of the average we get a medium. This allows the image to be more like original but now makes the function non-linear[8].

$$medium[A(x) + B(x)] \neq medium[A(x)] + medium[B(x)] \tag{1}$$

2.1.3 Gaussian

The Gaussian function, also known as the normal distribution or bell curve in statistics, is the basis for the Gaussian blur. When Gaussian is applied to images, it creates concentric circles the size of the standard deviation σ, with weights for the surrounding pixels that correspond to the bell curve, depending on their distance from the center. The each pixel in the resulting image is the average of circle of size σ. The Gaussian blur can be applied in two separate one dimensional (1D) calculations. Rather than applying a two dimensional (2D) matrix, it is possible to implement a series of 1D matrices horizontally and then repeating vertically. This linear filtering makes it suitable for many types of image processing tasks, which is why it's one of the most commonly used filters.

$$GaussianBlur = \frac{\exp\left(-\frac{x^2+y^2}{2\sigma^2}\right)}{2\pi\sigma^2} \tag{2}$$

2.2 CPU and GPU

The Central Processing Unit (CPU) is used to process computer tasks. It consist of Random Access Memory (RAM), cache, control units, and arithmetic units, with a control or arithmetic unit taking a clock cycle to complete. The CPU operates by receiving an input from memory, using the control and arithmetic units to process the input, and storing the results in memory. Since CPU has a clock speed average of 3 GHz, the CPU is able to perform sequential tasks quickly. Current day CPU's have multiple clusters of cores, with each cluster independent of another, allowing for a level of parallel-task completion.

The Graphical Processing Unit (GPU) is used to compute tasks in parallel. The GPU consists of interfaces which allow it to output video formats, Video Random Access Memory (VRAM), and cores. GPU cores exist within clusters and are specifically designed to perform video related tasks such as general arithmetic or vector computation. The GPU generally operates by receiving instructions from a queue, typically in the form of a vector. It then scheduling threads to pass the instructions to core clusters. Each core cluster assigns the instructions to each core for processing, depending on the processing need. Then, each core processes their instructions in parallel and the output is displayed. Thus, you can think of graphics displayed on a screen as a collection of ever-changing millions of processed vectors.

2.2.1 CUDA

No standard exists for GPU computing specifications. AMD refers to their core counts as Compute Units while NVIDIA refers to their core counts in terms of Computer Unified Device Architecture (CUDAcore), ray tracing (RT), and Tensor cores. These counts are not equatable since Compute Units are groups of processing elements whereas a CUDAcore for example, is a single processing element. For the purpose of this paper, we focus on the NVIDIA nomenclature.
Ada Lovelace is the specific configuration of CUDA, RT, and Tensor cores which makes up the architecture of the GPU. CUDAcores are used for general parallel computing tasks. It produces high accuracy and high throughput since each core completes in a clock cycle. RT cores are specifically designed to handle the computation necessary to cast light and shadows into digital images. Tensor cores are designed for vector and matrix operations in order to aide in AI and deep learning tasks. Since Tensor cores are able to perform an operation at faster than a clock cycle, they're useful when faster compute speed is preferred over precision. CUDA is the NVIDIA framework that utilizes the hardware in order to achieve GPU acceleration.

3 EXPERIMENT

3.1 Setup

The hardware used for bench marking consists of an AMD Ryzen 5 3600x 6-core CPU, 64GB RAM, and a GigaByte NVIDIA GeForce 4090 GPU. Additionally, a laptop with a 11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz, 2995 Mhz, 4 Core(s), 8 Logical Processor(s), 32GB Ram, with a NVIDIA GeForce GTX 1650 Ti with Max-Q GPU was used to compare performance. Windows 10 Operating System utilized Visual Studio Code 2022. The CUDA Toolkit V12.3 is a downloadable package for Visual Studio Code 2022 and did not require extra dependencies to setup.

3.2 Design

For the entirety of the code, opted not to use any thrid-party libraries that exist for image manipulation or can be used in conjunction with CUDA toolkit. Rather, we built everything in C++17 using standard C++ libraries and baseline CUDA toolkit. The average blur filter was initially built as a basis for understanding the image processing requirements. The sequential algorithm consists of reading the image, creating a copy of the image, running the Gaussian Blur through the image, and saving the output to a new image. Psuedocode for sequential Gaussian Blur is in the Appendix Algorithm 1. The parallel version of the Gaussian Blur consisted of utilizing the CUDA toolkit to allocate memory to and from the host (CPU) to the device (GPU) for processing. The block size and grid size for the parallel computing was determined by the limitations of the hardware. This specific GPU has a block size of 1024 threads (or 32 x 32 blocks). Grid size is determined as function of block size and the image dimensions.

$$gridSize[x][y] \equiv [\frac{width + blockSize.x - 1}{blockSize.x}], [\frac{height + blockSize.y - 1}{blockSize.y}] \tag{3}$$

Upon completion of the kernel programming, the data was reallocated from the device to the host and saved to a new file. Equation 3 shows the calculation used to determine grid size for the kernels. The pseudocode for parallel Gaussian Blur is in the Appendix Algorithm 2.

3.3 Test Framework

Our testing consisted of running the sequential and parallel algorithms on three sizes of images, with three sizes of σ. Bitmap (BMP) with 24-bit per pixel (bpp) was chosen as the image file type and size format to streamline input parameters. The three sizes tested were: 426x240 (240p), 1280x720 (720p), and 2560x1440 (1440p). Each image ran against a σ value of 5, 10, and 15. The sequential was only ran on the AMD Ryzen 5 3600x 6-core CPU and not the other PC as the main focus was GPU processing.

4 RESULTS

It was found that when setting up the program to run a sequence of tests, even though the program is calling to access the image independently every time, the compiler saves certain information which makes the first image passed through take significantly longer than the following images. Seeing this trend, we decided to run each image and σ as an independent executable, ensuring that memory was not saved from previous instructions that would altar the time outputs. We also saw that when running the test repeatedly, if the image existed, it would take longer to rewrite the image rather than create a new one. Thus, for the data we collected, we ensured that the output images did not exist prior to the programming running, meaning that none of the tests required to rewrite an existing BMP file.

The act of reading the data and writing it to the output file was sequentially done so we did not include it in the processing time for our Gaussian Blur data. Instead, for the sequential run, we calculated the time to be how long it took for the Gaussian Blur function to iterate through image. In the parallel runs, we calculated the time to be how long it took for program to copy the image to the GPU, process the image, and return the image to the CPU. The following figures showcase the exponential increase of performance between CPU and GPUs processing.

The data comparison for time is displayed in Figure 3, Figure 4, and Figure 5. The outputs from the code are in the Appendix: Figure 6.

Figure 3: Parallel vs Sequential Performance of 240p Image

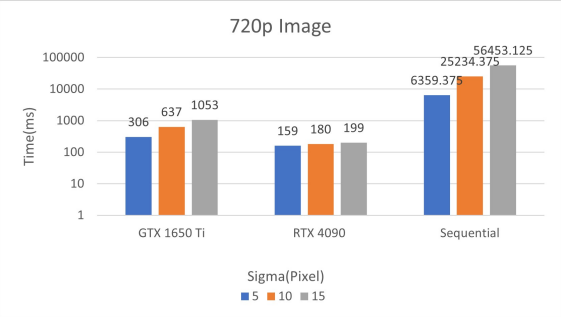

Figure 4: Parallel vs Sequential Performance of 720p Image

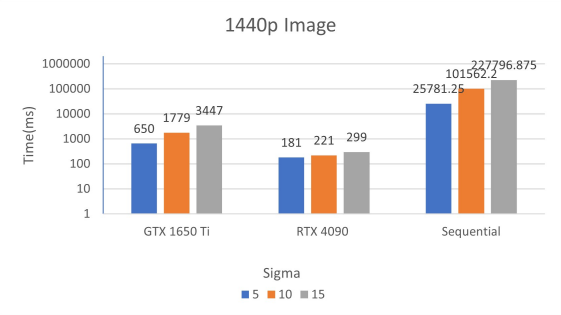

Figure 5: Parallel vs Sequential Performance of 1440p Image

5 CONCLUSION

Since the sequential algorithm is solely completed on the CPU, we can see see the see a vast difference in processing time from 5x (smallest image) to 752x (largest image) when compared to the desktop GPU. As expected there was an a direct relation of processing time to sigma value and pixel count. For an increase in sigma there is an increase in how many pixels that is inputted into the kernel for computations, which would increases the for loop in the gaussianWeight function.

Then we see that performance between the two GPUs to be from 2x to 11x increase in performance which is due to the increase of GPU memory and allowable threads. For pixel size, the amount of pixels is directly related to the amount of kernels needed. The laptop GPU has 1024 CUDA cores while the desktop GPU has 16,384 CUDA cores available. This means as the photo size increases but the sigma remains small, the laptop GPU will run out of parallel computing power and will need to schedule certain threads to process after the completion of other threads, whereas the desktop GPU has come computing power for parallel tasks at a larger thread scale. We also tested the code using a 16x16 block size rather than 32x32 but found the difference to be negligible, especially at larger resolutions.

6 FUTURE WORK

OpenCV is an open source image processing library we kept finding during our research. We would like to compare our implementation to the one available with OpenCV. We would like to have tested if there is a correlation between the kernel size and resolution for non-standard sized images versus standard sized images. We ran our tests using the general memory copy function available in CUDA. It would be interesting to see the performance difference of using the general copy between host and device versus the asynchronous memory transfer.

REFERENCES

[1] Dipesh Gyawali. Comparative Analysis of CPU and GPU Profiling for Deep Learning Models. In *arXiv:2309.02521*, DOI:10.48550/arXiv.2309.02521, 2023.

[2] Hiroaki Yamaura, Masayuki Tamura, and Satoshi Nakamura. Image Blurring Method for Enhancing Digital Content Viewing Experience. In *Human-Computer Interaction. Theories, Methods, and Human Issues: 20th International Conference, HCI International 2018*, pages 355-370, DOI:10.1007/978331991238729, 2018.

[3] Nahla M. Ibrahim, Ahmed Abou ElFarag and Rania Kadry. Gaussian Blur through Parallel Computing. In *IMPROVE 2021 - International Conference on Image Processing and Vision Engineering*, pages 175-179, DOI:10.5220/0010513301750179, 2021.

[4] Samil Karahan, Merve Kilinc Yildirim, Kadir Kirtac, Ferhat Sukru Rende, Gultekin Butun, and Hazim Kemal Ekenel. How Image Degradations Affect Deep CNN-Based Face Recognition. In *2016 International Conference of the Biometrics Special Interest Group (BIOSIG)*, pages 1-5, DOI:10.1109/BIOSIG.2016.7736924, 2016.

[5] Sanjib Das, Jonti Saikia, Soumita Das, and Nural Goni. A Comparative Study of Difference Noise Filtering Techniques in Digital Images. In *2019 8th International Conference on Modeling Simulation and Applied Optimization (ICMSAO)*, pages 180-191, DOI:10.1109/ICMSAO.2019.8880389, 2019.

[6] Sou Yoshihara1, Taiki Fukiage, and Shin'ya Nishida. Does training with blurred images bring convolutional neural networks closer to humans with respect to robust object recognition and internal representations? In *Frontiers in Psychology*, 14:1047694, DOI:10.3389/FPSYG.2023.1047694, 2023.

[7] (2018, September 9). https://static.igem.org/mediawiki/2018/9/9a/T–Cornell–ElectronicBandPass.png.

[8] (2011, September 9). https://www.nanophys.kth.se/nanolab/afm/icon/bruker-help/Content/SoftwareGuide/Offline/ModifyCommands/Gaussian.htm.

[9] (2020, January 22). https://www.topazlabs.com/learn/what-causes-noise-in-your-photos-and-how-you-can-fix-it.

7 APPENDIX

Algorithm 1: Gaussian Blur Algorithm

Function *gaussianWeight*(x, y, σ)**:**
  return $\exp\left(-\frac{x^2+y^2}{2\sigma^2}\right)/(2\pi\sigma^2)$;
Function *gaussianBlur*$(image, x, y, \sigma)$**:**
  $size \leftarrow \lceil 3 \cdot \sigma \rceil, center \leftarrow \lfloor size/2 \rfloor$ $totalWeight, totalRed, totalGreen, totalBlue \leftarrow 0.0$;
  for $i \leftarrow 0$ **to** $size - 1$ **do**
    for $j \leftarrow 0$ **to** $size - 1$ **do**
      $newX \leftarrow x - center + i$ **and** $newY \leftarrow y - center + j$;
      for $newX \geq 0$ ***and*** $newX < image.size()$ ***and*** $newY \geq 0$ ***and*** $newY < image[0].size()$ **do**
        $weight \leftarrow gaussianWeight(i - center, j - center, \sigma)$;
        $totalRed \leftarrow totalRed + weight \cdot image[newX][newY].red$;
        $totalGreen \leftarrow totalGreen + weight \cdot image[newX][newY].green$;
        $totalBlue \leftarrow totalBlue + weight \cdot image[newX][newY].blue$;
        $totalWeight \leftarrow totalWeight + weight$;
  $blurredPixel.red \leftarrow (totalRed/totalWeight)$;
  $blurredPixel.green \leftarrow (totalGreen/totalWeight)$;
  $blurredPixel.blue \leftarrow (totalBlue/totalWeight)$;
  return $blurredPixel$;
Function *applyGaussianBlur*$(image, \sigma)$**:**
  `// Apply Gaussian blur to the image`
  $blurredImage \leftarrow image$;
  for $x \leftarrow 0$ **to** $image.size() - 1$ **do**
    for $y \leftarrow 0$ **to** $image[x].size() - 1$ **do**
      $blurredImage[x][y] \leftarrow gaussianBlur(image, x, y, \sigma)$;
  $image \leftarrow blurredImage$;
Main `// Main function to take user input`
Read user input for parameters $inputfile$ *,* σ*, and* $outputfile$*;*
Call applyGaussianBlur with parameters $image$ *and* σ*;*
*Write output image to output file;***:**

Algorithm 2: Gaussian Blur Algorithm with CUDA

Function *gaussianWeight*(x, y, σ)**:**
  return $\exp\left(-\frac{x^2+y^2}{2\sigma^2}\right)/(2\pi\sigma^2)$;
Function *gaussianBlurKernel*($inputImage, outputImage, width, height, \sigma$)**:**
  $x \leftarrow blockIdx.x \times blockDim.x + threadIdx.x$ **and** $y \leftarrow blockIdx.y \times blockDim.y + threadIdx.y$;
  if $x < width$ ***and*** $y < height$ **then**
    $size \leftarrow \lceil 3 \cdot \sigma \rceil$, $center \leftarrow \lfloor size/2 \rfloor$, **and** $totalWeight, totalRed, totalGreen, totalBlue \leftarrow 0.0$;
    for $i \leftarrow -center$ **to** $center$ **do**
      for $j \leftarrow -center$ **to** $center$ **do**
        $newX \leftarrow x + i$ **and** $newY \leftarrow y + j$;
        `// Ensure that the indices are within bounds`
        if $newX \geq 0$ ***and*** $newX < width$ ***and*** $newY \geq 0$ ***and*** $newY < height$ **then**
          $weight \leftarrow gaussianWeight(i, j, \sigma)$;
          For each RGB color do:
          $totalColor \leftarrow totalColor + weight \cdot inputImage[newX + newY \times width].color$;
          $totalWeight \leftarrow totalWeight + weight$;
    For each color:
    $outputImage[y \times width + x]. \leftarrow static\_cast < unsignedchar > (totalColor/totalWeight)$;
Function *applyGaussianBlurCUDA*($image, width, height, \sigma$)**:**
  $d\_inputImage, d\_outputImage \leftarrow cudaMalloc(width \times height \times sizeof(Pixel))$; `// Allocate GPU memory`
  $cudaMemcpy(d\_inputImage, image.data(), width \times height \times sizeof(Pixel), cudaMemcpyHostToDevice)$;
  `// Copy input image to GPU`
  $blockSize \leftarrow (16, 16)$; `// Define block dimensions`
  $gridSize \leftarrow ((width + blockSize.x - 1)/blockSize.x, (height + blockSize.y - 1)/blockSize.y)$; `// Define grid dimensions`
  $gaussianBlurKernel <<< gridSize, blockSize >>> (d\_inputImage, d\_outputImage, width, height, \sigma)$;
  `// Launch the CUDA kernel`
  $cudaDeviceSynchronize()$; `// Sychronize`
  $cudaMemcpy(image.data(), d\_outputImage, width \times height \times sizeof(Pixel), cudaMemcpyDeviceToHost)$;
  `// Copy result back to CPU`
  $cudaDeviceSynchronize()$;
  $cudaFree(d\_inputImage)$; `// Free GPU memory`
  $cudaFree(d\_outputImage)$; `// Free GPU memory`
Main `// Main function to take user input`
Read user input for parameters $input file$ *,* σ*, and* $output file$*;*
Call applyGaussianBlur with parameters $image$ *and* σ*;*
*Write output image to output file;***:**

Figure 6: Blur Image Results