Multiscale Edge Detection Runtime Optimization

Boston University EC 526: Parallel Programming for High Performance Computing Final Project Report

Yahia Bakour and James Dunn

April/May 2019

I. ABSTRACT

Multiscale edge detection is a computer vision technique that finds pixels in an image that have sharp gradients at differing physical scales. The resulting multiscale edge-maps are useful for tasks such as object segmentation and image alignment/registration. At the core of all edge detection algorithms is a convolution of the input image with a kernel approximating the spatial derivative (gradient) of the image brightness. This convolution, as well as other loops in the program running the edge detection algorithm, are prime candidates for a parallel implementation within a GPU or across multiple cores or GPUs.

We present a parallelized implementation of the multiscale edge detection algorithm in c++ using openACC. A separate open MPI-based parallel implementation is also presented. The parallel implementations are compared with both a serial implementation and an implementation that uses the Fourier convolution theorem.

All source code is available at https://github.com/jimmykdunn/multiscaleEdgeDetection. Code was developed and tested on the Boston University shared computing cluster (scc1.bu.edu).

II. INTRODUCTION

The term "edge detection" applies to a variety of mathematical methods that aim to identify pixels in an image where the brightness changes sharply with respect to its neighboring pixels. Edge detection is a fundamental tool in computer vision and image processing used for segmentation and registration/alignment. We use the Sobel edge detection algorithm [1] to generate edge maps, but the methods in this report apply to other edge detection algorithms as well. The Sobel method convolves the original image by two 3x3 kernels to approximate the brightness derivatives in the horizontal and vertical directions.

The multiscale edge detection procedure simply runs the edge detection algorithm at multiple scales. Specifically, it takes the original image, shrinks it by some factor(s), then runs the same edge detection algorithm on the shrunk image(s). The result is a stack of edge-maps that show the edges in the image at different physical scales. The images in Figures 1 through 8 show one potential application of the multiscale edge detection technique.

The non-dependent loops in the multiscale edge detection process make it a prime candidate for a parallel implementation, both on a GPU via openACC, and running edge detection on images across cores via MPI.

We first implement a serial version of the edge detection algorithm in c++ and use it for multiscale edge detection. We then examine two different parallelization techniques with the aim of speeding up the program: an OpenACC implementation run on a Tesla GPU, and an MPI implementation with each edgemap scale run on a separate CPU core. Finally, we compare a serial implementation that uses the Fourier convolution theorem to execute the convolutions in the edge detection algorithm.

In all our implementations, the output edgemaps themselves are identical or nearly identical, but the runtime is markedly different. We use runtime on the target system, the Boston University Shared Computing Cluster (SCC) as our primary performance metric.

III. SERIAL IMPLEMENTATION

The serial implementation of our multiscale edge detection algorithm proceeds as follows. We first shrink the input image by a factor $F \in [2, 4, 6, 8]$, we run the Sobel Edge Detection algorithm on the

Figure 1: Small 2.4 Mpix (1920x1232) purple flowers image. Image credit [2]

Figure 2: Edge detection on full-scale purple flowers image. Notice the fine edges in the flower field.

Figure 3: Edge detection on ½ scale purple flowers image. Notice the clouds starting to be visible.

Figure 4: Edge detection on ¼ scale purple flowers image. Notice the clouds are much more visible, but the flowers are not well segmented.

Figure 5: Large 17.9 Mpix (5184x3456) mountain village image. Image credit [3].

Figure 6: Edge detection on full-scale mountain village image.

Figure 7: Edge detection on ½-scale mountain village image.

Figure 8: Edge detection on ¼-scale mountain village image. Notice the stronger window and roof edges.

shrunk image, then we enlarge the image by a factor F, then repeat for larger and larger factors F. The purpose of this is to gain more and more detail from the image on previously overlooked edges.

Algorithm for Multiscale Edge Detection:

- I : Input Image
- Output: Array containing output images
- F: Array of factors to run multiscale edge detection on, nominally [2, 4, 6, 8].
- ***ShrinkImage***: Function that takes in an image and runs simple average pooling to shrink by a factor F.
- ***EnlargeImage***: Function that takes in an image and uses a simple copy algorithm to create an enlarged version of the image.
- ***SobelEdgeDetn***: Use the sobel operator to build an edge map from the Image.

```
Multiscale(I,Output,F):
-For I in range(Len(F)):
--OutSmallImage = ShrinkImage(Input, F[I])
--OutEdgemap = SobelEdgeDetn(OutSmallImage)
--Output[I] = EnlargeImage(OutEdgemap, F[I])
```

Equation 1: Pseudocode for the multiscale edge detection algorithm.

Sobel Edge Detection is done by convolving the original image by the following two 3x3 kernels:

$$\mathbf{G}_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * \mathbf{A}$$

Where:

- **A** is the original input image
- $\mathbf{G}_x$ and $\mathbf{G}_y$ are the gradients found from the convolution.

We then find the magnitude of the gradient from these components at each pixel and decide whether it is above or below a threshold. If it is above the threshold, then we count it as an edge. Shrinking the input image is done using simple average pooling of the nearest F pixels in each direction, where F is the (integer) scale factor. Enlarging the image is done by simple copying of the previous pixel to the new pixels, i.e. nearest-neighbor interpolation.

IV. OPEN ACC IMPLEMENTATION

We quickly we realized that the serial version of our code had many backwards compatibility issues and we had to rewrite the code for our grayscale, enlarge, shrink, and Sobel edge detection functions to be parallelizable by openACC. This led to a speedup of slightly less than 5x. We then realized that a lot of the time, the arrays necessary for GPU accelerated multiscale edge detection were already on the GPU and didn't have to be copied in and out repetitively. We rewrote our multiscale edge detection function to only copyin the input array (Image) once and then constantly refer to it when needed using openACC's ***present*** clause. We managed to gain a speedup of x7 on the small image from the serial version after modifying it to remove all backwards dependencies and ensuring to *copyin* all static arrays during the multiscale edge detection. We also turned our 2 kernels from 3x3 matrices to 18 single int variables which helped shave off a few more cycles as we didn't have to perform a *copyin* of the 2 kernels every time the function is run.

One of the main issues we faced was decoupling dependent loops to gain independence from one another. When loops were intrinsically parallel, we inserted the `#pragma acc loop independent` directive to tell the compiler that the loops are independent from one another. We also tested using the `#pragma acc loop collapse(n)` directive that collapses the first n loops into one loop but found better results with the first approach.

A naïve ACC implementation of this project would have only gained a speedup of 2x over the serial implementation, but after repeatedly analyzing the code for speed bottlenecks, we were able to maximize our usage of the GPU through the use of OpenACC best practices and accelerate our code.

V. OPEN MPI IMPLEMENTATION

Open MPI provides a smooth and compact way to simultaneously execute a program on multiple "ranks". Each "rank" can be a separate core, CPU, GPU, or even a separate physical computer. Open MPI also provides a standard for communication between the ranks that used behind-the-scenes.

The most clear-cut but naïve way to speed up multiscale edge detection with MPI is to simply run each of the edge scales on its own rank because the scales are completely independent from one another. A less clear-cut way to use MPI would be to split up the image into blocks and assign each block to a different rank. Less obvious still would be to use MPI to split up the edge scales across *GPUs*, and then let openACC parallelize on each GPU.

We implement the first of these methods - simply running each scale in the multiscale edge detection algorithm on its own core of a CPU. This is admittedly suboptimal since the cores running larger scale factors (smaller images) will finish more quickly and then simply wait, but it was the most straightforward to write code for, so it was the first method we tried. Using MPI and ACC simultaneously to run across multiple GPUs was also explored but abandoned for lack of time.

V. FFT IMPLEMENTATION

The primary use of the Fast Fourier Transform (FFT) is to quickly convert data from time space to frequency space and back again with order $Nlog_2(N)$ operations. By comparison, the standard (slow) Discrete Fourier Transform (DFT) uses order N^2 operations. The FFT thus provides the potential for significant decreases in runtime, particularly for large arrays. Unfortunately, the FFT is a recursive algorithm, so it is not amenable to parallelization.

One of the other uses of the FFT is to perform fast convolutions, which could make the convolution step of our edge detection algorithm faster. This is made possible by use of the Fourier Convolution Theorem [4]. The Fourier Convolution Theorem states that the inverse Fourier transform of the product of two Fourier transformed functions is mathematically equivalent to convolution. In pseudocode:

$$A \otimes B = invFFT(FFT(A) * FFT(B))$$

The most intuitive way to understand this is that the shift operation is performed with multiplication in frequency space. Mathematically, a shift of x by δ is performed with:

$$e^{ik(x+\delta)} = e^{ikx} e^{ik\delta}$$

Convolution is merely a series of shifts and multiplications, so this simple equation brings light to why the Fourier convolution theorem works.

Naïvely, one would expect that simply taking the serial edge detection code and replacing the corresponding 4x nested for loops with a FFT-based convolution would offer an N^4 to $2N^2 log_2(N)$ speedup. Upon careful consideration however, this arithmetic does not accurately describe the convolutions that are performed in edge detection.

Let the image that we wish to convolve by be NxN pixels, and let the edge detection kernel be one of the usual 3x3 Sobel kernels. The algorithm contains 4 nested for loops with size (N, N, 3, 3). Thus one direction of the Sobel convolution (x or y), takes only $O(9N^2)$ operations.

The Fourier convolution theorem requires the two arrays being convolved to be the same size, because the product in the theorem is an element-by-element product. The FFT-based convolution must act on two N^2 images instead of one N^2 image and one 3^2 image. Consequently, in order to convolve two arrays that are not the same size, in our case an NxN image and a $3x3$ kernel, the 3x3 kernel must be zero-padded to the size of the larger array before performing the FFT. Thus, instead of getting the $9N^2$ to $2log_2(3)\,Nlog_2(N)$ speedup that we might have expected, we instead get $9N^2$ vs $2N^2 log_2(N)$. This actually means the FFT convolution will be a *slower* algorithm for any N>22.

In our implementation, the FFT convolution method actually does somewhat worse than $2N^2 log_2(N)$ due to the type conversions from the pixels' native uint_8 to the 64-bit complex required by the FFT and the associated additional memory allocations. Our experiments on 2.4 MP and larger images (see results section) confirm this.

VI. RESULTS

Trials using each of the implementations defined above were run 10 times on the BU SCC to gather statistics. Four images with sizes from 2.4 Mpix to 17.9 Mpix were used to show how runtime scales with problem size.

The serial implementation was compiled with g++ using only the "-std=c++11" flag. No compiler optimizations were used. The open ACC implementation was run on a single Tesla M2070 GPU cards with 6 GB of Memory. Open ACC compilation was performed with "pgc++ -Minfo=accel -ta=tesla ..." and run with 1 CPU and 1 Tesla GPU in an interactive terminal via "qrsh -pe omp 1 -P paralg -l gpus=1.0 -l gpu_c=6.0". Open MPI compilation was perfomred with mpicxx, and run with 4 cores ("-np 4") using the same qrsh command to get an interactive terminal.

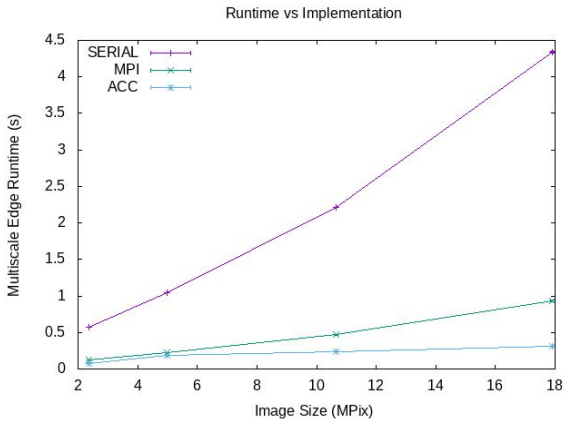

Figure 9: Runtime results for the various implementations of multiscale edge detection on the target hardware. Averages and standard deviations are calculated over 10 independent trials.

The FFT implementation for 3x3 Sobel kernels is 161x slower than the serial implementation, for reasons mentioned earlier in section V. To demonstrate at what size kernel the FFT would start to outperform the serial and parallel (non-FFT) implementations, we can artificially enlarge the edge detection convolution kernel. This slows the serial implementation down, but has a minimal effect of the FFT implementation. We found that enlarging the kernel from 3x3 to 47x47 made the serial implementation have comparable runtime to the FFT implementation, as in Table [1].

Implementation	Kernel size	Runtime ± σ (s), 1920x1232 image
Serial	47x47	97.47 ± 0.32
FFT convolutions	47x47	98.28 ± 0.36

Table 1: Runtimes for artificially-enlarged convolution kernels. Kernel sizes are specifically chosen so that runtimes are comparable to the FFT method. Averages and standard deviations are calculated over 10 independent trials.

Note that the artificial enlargement of the convolution kernel used for Table 1's results is done with zeropadding and consequently does not change the output edgemaps. This makes the serial implementation artificially inefficient for the purpose of demonstration. We could envision an edge detection kernel or some other algorithm that utilizes convolution with large kernels, where the FFT method would legitimately outpace both the serial and parallel methods.

VII. CONCLUSIONS

We have successfully parallelized all of the parallelizable loops in a multiscale edge detection algorithm implemented in c++. The best runtime improvements were shown using openACC on a Tesla GPU.

By an apples-to-apples runtime comparison against a serial version of the same code on the same system (Boston University SCC), we found an improvement of ***7.036x*** for a small image (2.4 Mpix), and ***13.9x*** for a large image (17.9 Mpix) using ACC. Parallelization across CPUs with MPI yielded the approximately the expected speedup of ***4.625x*** on the small image and ***4.6767x*** on the large image by letting each scale run on its own core. The full-scale edge detection took the most time to run, and so we were able to execute all scales of edge detection in the same time as the full scale.

To complement the parallelization effort, we took the convolution part of the serial code and implemented it using an FFT with the Fourier convolution theorem. The resulting 161x slower runtime shows that the FFT is at a clear runtime disadvantage for the small 3x3 pixel kernels used for Sobel edge detection. We ran with larger kernels to determine what kernel size would be needed for the FFT to be a more efficient implementation. We found that the FFT implementation was more efficient than the serial implementation for kernels larger than 47x47 pixels for the images we calculated performance on.

Future work would include implementing the MPI parallelization more efficiently by having each rank run a slice of the image *at all scales*, rather than each rank running a single scale. This is straightforward, but it involves communication between ranks at the boundaries and the associated additional code. Additionally, the pgc++ compiler (ACC) could be linked to from MPI. This would allow us to use MPI to split up the task between multiple *GPUs*, rather than across multiple CPU cores, resulting in the best of both ACC and MPI. We attempted the ACC+MPI fusion methods in [7] and [8], but neither was configured properly to work on the SCC. Further effort in this area would likely result in additional speed gains.

VIII. REFERENCES

1. "Sobel operator" , https://en.wikipedia.org/wiki/Sobel_operator, accessed 4/21/19
2. "Purple Petaled Flower Field", https://www.pexels.com/photo/purple-petaled-flower-field-1131407/, accessed 4/27/19
3. "Architecture-cliffside-cold", https://images.pexels.com/photos/789380/pexels-photo-789380.jpeg?cs=srgb&dl=architecture-cliffside-cold-789380.jpg&fm=jpg , accessed 4/22/19
4. "Convolution Theorem", https://en.wikipedia.org/wiki/Convolution_theorem, accessed 4/25/19
5. "OpenACC Tutorial - Data movement", https://docs.computecanada.ca/wiki/OpenACC_Tutorial_-_Data_movement#C.2B.2B_Classes, accessed 4/24/19
6. "Advanced OpenACC" , http://icl.cs.utk.edu/classes/cosc462/2017/pdf/OpenACC_3.pdf, accessed 4/23/19
7. "Multiprocessor Programming: TechWeb : Boston University", http://www.bu.edu/tech/support/research/software-and-programming/programming/multiprocessor/#MPI accessed 4/30/19
8. "OpenACC + MPI", http://www.speedup.ch/workshops/w43_2014/tutorial/html/openacc_mpi_1.html, accessed 4/30/19