Comparative Study on Edge Detection Algorithms using OpenACC and OpenMPI on Multicore Systems

Aakashdeep Goyal[1], Zuqing Li[1], Haklin Kimm[2]
Department of Computer Science,
East Stroudsburg University
East Stroudsburg, PA, 18301, USA
[1]{agoyal, lzuqing}@live.esu.edu, [2]hkimm@po-box.esu.edu

Abstract— **In this paper, we present a comparative study on parallel edge detection algorithms upon high-resolution satellite images, implemented on OpenACC, Hybrid OpenMP/MPI, OpenMP, and MPI models on which the Sobel, Prewitt and Canny algorithms were developed using C++ language and OpenCV. The performance of these computing models were measured in terms of speedup and execution time by implementing the edge detection algorithms using various sized images and programming constructs. The program implementations using OpenACC display the largest speedup in CPU time, which is followed by the Hybrid OpenMP/MPI model. The parallel detection algorithms using OpenACC obtain the greatest speedup of around 6.5 over OpenMP model. The parallel Sobel and Prewitt algorithms are relatively 2 times faster than the Canny in all respects.**

Keywords-Edge Detection; OpenACC; OpenMP; OpenMPI; Multicore

I. INTRODUCTION

The field of Computer Vision has grown significantly over the last decade. Today we are at a stage of analyzing digital images where the aim is to interpret image content and make real-time decisions. One way to achieve this goal is by having real-time execution in processing images that are often unmet using the conventional single-threaded computing. Thus, the parallel processing for image analysis is inevitable and leads to the need of high-performance computing for analyzing high-resolution images.

Edge detection and segmentation are one of the most computation intensive processes in image analysis, which have been applied to various areas such as coastline mapping, geologic extraction, object or character recognition. A comprehensive performance analysis of edge detection algorithms for coastline images has been provided in [1]. The edge detection algorithms often involve recurring calculations over independent data contexts, which allow a huge scope of speedup using parallelism or pipelining. Nowadays, computing is drifting away to include a heterogeneous CPU-GPU combination in achieving improved computational performance [2]. In this, the CPU is used as a host working together with the GPU (device) to create a high-performance processing unit. A current example is the supercomputer Titan that uses almost 30,000 CPU cores and 18000 GPU cards [3]. This not only raises the bar for the high-performance computing but also induces a challenge in selecting an architecture-specific programming model that can also deliver the required performance. The parallel and distributed techniques over existing edge-detection algorithms are implemented in our work.

A traditional GPU has been a fixed-function graphic processor that parallelized graphic algorithms using graphic APIs such as OpenGL [4]. The motivation to use GPU for a non-graphic computation led to the introduction of NVIDIA's Tesla architecture in 2006. The architecture enabled parallelism over general purpose GPU (GPGPU) computations by using low-level programming models such as CUDA [5] and OpenCL [6]. Although these models exploit substantial parallelism and offer greater control over execution, of which implementation requires a thorough understanding of the underlying architecture. At the same time, it limits portability to only a few architectures. The second category involves high-level programming models such as OpenACC [7] also known as compiler directives. Unlike their predecessors, the underlying compiler and runtime environment for these models offer a greater level of abstraction thereby resulting in easier coding environments and achieving comparable parallelism. Thus, meeting high-performance standards in domains like multimedia, gaming, big data, auto industry, etc. A case study analysis of OpenACC models over some common image processing operations has been provided in [8].

On the other hand, multicore non-GPU systems use multiple threads across processors or cores to achieve parallelism. Here each thread can use the same instruction to target different data elements representing single instruction multiple-data (SIMD) programming model. OpenMP [9] and MPI [10] are two dominant portable and scalable industry models for multicore systems, on which the host processor divides the work among multiple coprocessors forming a homogenous system. While the created threads on OpenMP programming model have a shared memory, MPI uses a distributed memory model where the memories corresponding to each processor group are connected on a shared network. Various multicore languages and libraries are compared in [11] based on their level of abstraction and performance. MATLAB PCT [12], Intel TBB [13] and Cilk Plus [14] are some more high-level C/C++/MATLAB languages and libraries using data or task parallelism.

978-1-5386-3441-7/17 $31.00 © 2017 IEEE
DOI 10.1109/MCSoC.2017.16

II. PREVIOUS STUDIES AND RELATED WORK

A. GPU Computing

Originally, GPUs were used as a fixed function graphic pipeline model. The underlying hardware was arranged in a set of smaller units that carried out corresponding operations in the pipeline to deliver the final rasterization. The rendering pipeline model was non-programmable and thus based on user-provided configurations rather than user-provided programs [15]. In the early 2000s, this pipeline transitioned from non-programmable to programmable. The release of NVIDIA GeForce 3 in 2001 and ATI Radeon 9700 in 2002 marked the beginning of this evolution and the previous non-programmable units in the fixed-function pipeline model were now programmable by using user-defined shader programs. This not only increased the speed and parallelism in processing graphic primitives but also resulted in more sophisticated GPU applications such as gaming and movie designs. A comprehensive survey of this evolution in the GPU architecture is presented in [16]. Though highly programmable, GPUs were still largely conceived for carrying out graphics applications programmed through API's such as OpenGL due to the inherent graphic pipeline structuring. However, the use of these API's for non-graphics or general-purpose programming was complicated and often required a translation for mapping the non-graphics program to the underlying graphic concentrated pipeline. An illustration showing this translation is given in [17].

On the other hand, high-level languages such as CG/HLSL/GLSL [18-20] allow greater flexibility in GPGPU programming but still require following this pipeline structure. The general-purpose GPU (GPGPU) programming was revolutionized in 2006 with the release of NVIDIA Tesla GeForce 8800 that abstracted the earlier pipelining and unified the underlying stages of the processor. Since then, a processor has also been termed as a streaming multiprocessor (SM). Following which, languages such as Brooks utilized the SM model in eliminating the need to use any graphics terminology. In 2007, the introduction of CUDA by NVIDIA extended this progress and allowed a greater control over GPU thus highlighting an increased trend in GPU computing. It is has remained one of the most widely anticipated platforms in GPGPU computing. An implementation of CUDA low-level model for Canny Edge Detection algorithm can be seen in [21]. In 2010, NVIDIA released Fermi architecture – the first GPU design focused on GPGPU computing. The earliest Fermi-based GPUs featured up to 512 CUDA cores. This paper uses Tesla K40 GPU that features up to 2880 CUDA cores.

With this increased accessibility and productivity in GPU computing, it is no coincidence that the field of computer vision and image processing has been an integral part of it. The fact has propelled the emergence of a diverse range of techniques for achieving high accelerations in performance. Models such as OpenACC abstracted the underlying architecture programming seamlessly with C/C++ and Fortran. In [22], OpenACC has been implemented to speed up some fundamental image processing algorithms like thresholding, rescaling, and blurring. Regarding performance improvement on a hardware level, edge detection algorithms on architectures such as Field-programmable gate arrays (FPGA's) have been implemented, as illustrated in [23] for Sobel Edge detection. An overview of various hardware and software levels for parallel image processing is provided in [24].

B. Edge Detection

The present paper parallelizes Prewitt, Sobel and Canny edge detection techniques for satellite images. An edge represents a discontinuity in pixel intensity. Edge detection is a process of localizing these pixels across the image space using the first and second order derivatives. While the first order derivative maps the spatial change in pixel intensity for an image, the second order derivative locates zero crossings or the edge points. As described in [25], the process can be described as having three important stages namely smoothing, differentiation and labeling. We use gradient vector that uses first order derivate for calculating the bidirectional change in pixel intensity across the x and y directions as shown in (1). Here, the vector is perpendicular to the direction of the edge.

$$\nabla f = \begin{bmatrix} G_x \\ G_y \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} \tag{1}$$

Its magnitude is obtained by adding the x and y components as shown in (2). The pixels having large gradient values are possibly the part of an edge. An edge detection algorithm uses masks (two-dimensional matrices) which calculate and compare this pixel's gradient value with a pre-set threshold value to decide if it is an edge.

$$|\nabla f| = \sqrt{G_x{}^2 + G_y{}^2} \tag{2}$$

$$G(x, y) = -e^{\left(\frac{-\left(x^2+y^2\right)}{2\sigma^2}\right)} \tag{3}$$

$$I_G(x, y) = \sum_{i=-m/2}^{m/2} \sum_{j=-m/2}^{m/2} G(x, y) I(x-i, y-j) \tag{4}$$

In our experiment, we also use Gaussian filtering technique which uses Gaussian masks for noise reduction. These masks use the Gaussian function for calculating the pixel value as given by (3). Equation (4) gives the resulting pixel intensity I_G after applying the Gaussian function to the MXM kernel centered around the point *(x, y)*. Here, *m* is the length of the kernel side. A detailed explanation has been provided in [25].

III. PARALLEL AND DISTRIBUTED COMPUTING MODELS

In this section, we introduce the following multicore programming models – OpenACC, MPI and OpenMP along with their Sobel, Prewitt and Canny C++ implementations. The existing models can broadly be divided as being high-

level and low-level. Accordingly, OpenACC and OpenMP are high-level compiler directives whereas MPI library is relatively at a lower-level often requiring changes in the program structure.

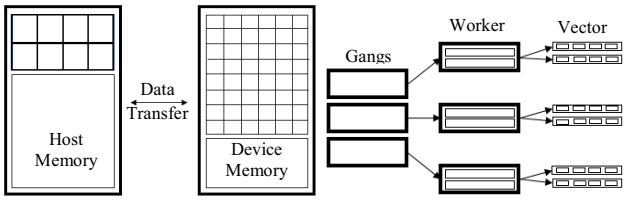

Figure 1. CPU-GPU computing model

A. OpenACC Implementation

As seen earlier, a plethora of programming models has started to emerge for providing parallelism. Popular extensions like CUDA and OpenCL achieve high performances but at the cost of exposing underlying hardware details. At the same time, OpenACC abstract away these details and provides more portability and comparable performance thus making it a widely applicable tool for more than just a few architectures. The model uses a separate host and device memory as shown in Fig. 1 above thus needing an explicit data transfer to the GPU, where limiting data movement and its reusability are crucial for maintaining high speedups. Unlike CUDA, the OpenACC compiler and runtime environment automatically handle the memory management operations related to data movement.

1) Sobel/Prewitt OpenACC: This subsection implements OpenACC compiler directives on Sobel/Prewitt C++ algorithms. We also use OpenCV [26] library for reading and displaying an image, in which the *Mat* object is used to represent an image. The following algorithm is universal to Sobel and Prewitt methods because the two are different in the mask used during convolution.

Algorithm 1: *Sobel/Prewitt OpenACC*

1. **Define** size *k*, *stdev*, Sobel/Prewitt ($k\times k$) E_x and E_y, Gaussian (k) G_x and G_y
2. *cv::Mat*(*src*) ← input image
3. *cv::Mat*(*psrc_gray*) ← grayscale *psrc*
4. Convert *Mat(src_gray)* to *srcarray*[...][...]
5. Declare *des*, *desarray*[…][…]
6. **#pragma acc data pcopyin(*G_x*, *G_y*, *E_x*, *E_y*, *srcarray*) pcopyout(*desarray*) pcreate(*angle*)**
7. **#pragma acc parallel loop gang vector_length(#vectors) collapse(2)**
8. **for** r = 1 to $rows_{srcarray}$ **do**
9. **for** c = 1 to $cols_{srcarray}$ **do**
10. *srcarray*[*r*][*c*] ←*G_kernel*(*r, c*)
11. **end for**
12. **end for**
13. **#pragma acc parallel loop gang vector_length(#vectors) collapse(2)**
14. **for** *row*s = 1 to $rows_{srcarray}$ **do**
15. **for** *cols* = 1 to $cols_{srcarray}$ **do**
16. *desarray*[*r*][*c*] ←*S_kernel*(*r, c*)
17. **end for**
18. **end for**
19. Convert *desarray*[…][…] to *Mat*(*des*)
20. **return** *Mat*(*des*)

The details are illustrated in Algorithm 1 shown above. The *G_kernel*() and *S_kernel*() methods are given as Algorithm 2 and 3 respectively**.** As seen at lines 6, 7 and 13, the directives are enabled by using *pragma* followed by *acc* to denote OpenACC. Here, *loop* is used to parallelize the for loops. The region from lines 6 to 18 is offloaded to GPU CUDA cores, where parallelism has been achieved using the parallel construct. Unlike *kernel* construct where the decision to parallelize is controlled by the compiler's willingness (if considered safe), *parallel* construct forces the compiler to do so. The disadvantage in using parallel over kernel is the need to explicitly define the data movement. The *kernel* construct automatically handles the data movement between host and device and often, it will tend to make redundant data movements for multiple parallel regions requiring the same data, thus degrading performance. As seen at line 6, the same data construct is used across parallel regions shown at lines 7 and 13. The *pcopyin* construct creates space and initializes the data (copied from host) on the device memory, whereas the *pcopyout* construct creates the space for the data copied without initializing them. A *vector* (thread) represents the finest granularity in achieving parallelism where each one handles a single data value thus following the SIMD operation.

Algorithm 2: *G_kernel*(*r, c*) *Gaussian OpenACC*

1. **#pragma acc loop vector reduction (+:*$temp_x$*) collapse(2)**
2. **for** p = *c-stdev*, t = 0 to *c+stdev*, *k* **do**
3. ***$temp_x$*** ← *srcarray*[*r*][*p*] * G_x[*t*]
4. *srcarray*[*r*][*c*] ← ***$temp_x$***
5. **end for**
6. **#pragma acc loop vector reduction (+:*$temp_y$*) collapse(2)**
7. **for** p = *r-stdev*, t = 0 to *r+stdev*, *k* **do**
8. ***$temp_y$*** ← *srcarray*[*p*][*c*] * G_y[*t*]
9. *srcarray*[*r*][*c*] ← ***$temp_y$***
10. **end for**

Algorithm 3: *S_kernel*(*r, c*) *Sobel/Prewitt Kernel*

1. **#pragma acc loop vector reduction (+:*g_x*) collapse(2)**
2. **for** p_r = *r- stdev*, t_r = 0 to *r+ stdev*, *k* **do**

```
3.      for p_c = c- stdev, t_c = 0 to c+ stdev, k do
4.        g_x ← srcarray[p_r][p_c] * E_x[t_r][ t_c]
5.      end for
6.    end for
7.    #pragma acc loop vector reduction (+:g_y)
      collapse(2)
8.    for p_r = r- stdev, t_r = 0 to r+ stdev, k do
9.      for p_c = c- stdev, t_c = 0 to c+ stdev, k do
10.       g_y ← srcarray[p_r][p_c] * E_y[t_r][ t_c]
11.     end for
12.   end for
13.   desarray[r][c] ← sqrt(g_x^2 + g_y^2)
14.   angle[r][c] ← g_y / g_x
```

2) Canny OpenACC: The Canny algorithm can be conceived as an extension of Sobel/Prewitt algorithm with the only difference of that it implements edge thinning and removal of false edges as shown by Algorithm 4 and 5 respectively.

Algorithm 4: *Edge_Thinning*() *OpenACC*

```
1.  #pragma acc parallel loop gang vector_length
    (#vectors) collapse(2)
2.  for r = 1 to rows_desarray do
3.    for c = 1 to cols_desarray do
4.      temp_angle ← angle(r, c)
5.      temp_grad ← desarray(r, c)
6.      #pragma acc loop vector collapse(2)
7.      for p_r = r - stdev to r+stdev do
8.        for p_c = c-stdev to c+stdev do
9.          if angle(p_r, p_c) = temp_angle and
                desarray(p_r, p_c) > temp_grad then
10.           desarray(r, c) ← 0
11.         end if
12.       end for
13.     end for
14.   end for
15. end for
```

Algorithm 5: *Remove_WeakEdges(T1, T2)* *OpenACC*

```
16. #pragma acc parallel loop gang vector_length
    (#vectors) collapse(2)
17. for r = 1 to rows_desarray do
18.   for c = 1 to cols_desarray do
19.     if T1 < desarray(r, c) < T2 then
20.       if desarray(r, c) not connects
          {desarray(r, c) < T2} then
21.         desarray(r, c) ← 0
22.       end if
23.     else if desarray(r, c) < T2 then
24.       desarray(r, c) ← 0
25.     end if
26.   end for
27. end for
```

As seen at lines 9 & 10 (Algorithm 4), unwanted pixels near the edge are removed by finding the local maxima of the edge. In (Algorithm 5), pixels are checked for their gradient values against threshold limits *T1* and *T2* (*T1 > T2*) to determine weak edges.

B. OpenMP Implementation

OpenMP is a high-level, portable and scalable application programming interface (API) being used for achieving parallelism on multicore architectures. The present interface is based on the shared memory model (SMP) where different processors share a common memory. In today's generation, the SMPs have advanced from having a single-core processor to multiple cores having independent caches. Thus, OpenMP generates multithreaded applications based on familiar programming languages such as C/C++ and Fortran, thus achieving parallelism by executing multiple threads across several cores or processors. The OpenMP API is owned and managed by ARB (Architecture Reviews Board) that is a non-profit organization [27] with members including AMD, IBM, NVIDIA, Cray, Intel, etc. [28].

1) Sobel/Prewitt OpenMP: Parallelism is achieved through multithreading on shared memory processors, where the directives are enabled by using pragma followed by omp for OpenMP as seen in Algorithm 6.

Algorithm 6: *Sobel/Prewitt OpenMP*

```
1.  Define size k, stdev, Sobel/Prewitt (k×k) E_x and E_y,
    Gaussian (k) G_x and G_y
2.  cv::Mat(src)← input image
3.  cv::Mat(psrc_gray)← grayscale psrc
4.  Declare cv::Mat(des)
5.  #pragma omp parallel shared(default) private(r,c)
    num_threads (#threads) schedule(static)
      #pragma omp for collapse(2)
      for r = 1 to rows_srcgray do
6.      for c = 1 to cols_srcgray do
7.        Mat(src_gray) at (r, c) ←G_kernel(r, c)
8.      end for
9.    end for
10.   #pragma omp for collapse(2)
11.   for r = 1 to rows_srcarray do
12.     for c = 1 to cols_srcarray do
13.       Mat(des) at (r, c) ←S_kernel(r, c)
14.     end for
15.   end for
16. end for
```

As seen in Algorithm 6, the lines 5-16 represent the work-sharing region by using the *parallel* construct. Multiple for loops exist in this parallel region (lines 6 and 11) which are

parallelized by using *for* with *omp.* Here, work is distributed among different threads running on multiple cores (also explained by the master-slave thread distribution as shown in Fig. 2). Note that the variables like loop iterations need to be kept private to each thread. Unlike OpenACC, OpenMP does not allow nested *for* constructs. Hence, the *G_kernel*() and *S_kernel*() functions at lines 8 and 9 are executed sequentially without using any pragmas inside.

2) Canny OpenMP: Similar procedure is followed in Canny where the steps of edge thinning and weak edge-removal are now accompanied by *omp pragmas* instead of *acc pragmas*. Thus, lines 1 and 16 shown in Algorithms 5 and 6 respectively are now replaced with *#pragma omp for collapse(2)*. Line 6 in Algorithm 6 cannot be replaced with a nested for. Thus, the line will be blanked in case of OpenMP.

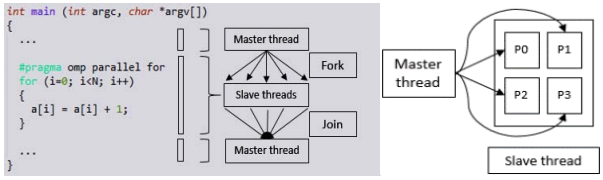

Figure 2. Master-Slave thread distribution

C. MPI Implementation

The Message Passing Interface (MPI) library implementation uses multiple shared memory processors distributed across a shared network. The processors can be grouped into sets with each set representing a node. Thus, MPI is a method of seamlessly utilizing multiple nodes in the cluster to achieve parallelism and speedups. Here, one node acts as a host and the rest as guests. The image is copied onto each node which then reads only a certain offset of the image thereby distributing it among its processors using the shared memory. In this way, we minimize traffic delays in distributing the image across the outlying processors on different nodes. As we will see in the next section, MPI requires changes in the algorithm structure to accommodate this request.

1) Sobel/Prewitt MPI: The current implementation is shown in Algorithm 7. Here, *slots* represent the number of processors on each node. The input *N* set the limit on the number of processes used across the cluster. As seen in line 6, MPI groups the processes into their sets or node specific ID's. In line 10-18, the first process running on every node reads the offset of image represented by *prsc*.

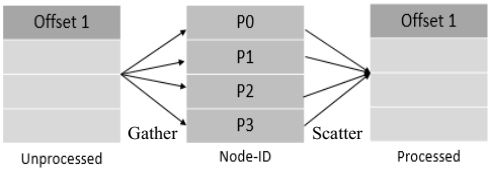

Figure 3. Gather and Scatter

Furthermore, *psrc* is split into sub-parts (depending upon the number of processors on each node) represented by *ppsrc*. MPI Scatter distribute the *psrc* into multiple *ppsrc*'s on every node, whereas Gather combines multiple *ppdes*'s into a single *pdes* for every node. The operation can be seen in Fig. 3.

Algorithm 7: *Sobel/Prewitt MPI (N)*

1. **Define** size *k*, *stdev*, Sobel/Prewitt ($k\times k$) E_x and E_y, Gaussian (k) G_x and G_y, nodes *n*, slots per node *s*
2. *MPI_Init(...)*
3. $prank_{world} \leftarrow$ *MPI_Comm_rank*(...)
4. $psize_{world} \leftarrow$ *MPI_Comm_size*(*N*)
5. *stdev* ← (*k-1*)/2
6. $d \leftarrow psize_{world}/n$
7. $pset \leftarrow prank_{world}/d$
8. *MPI_Comm_split(pset)*
9. $prank_{set} \leftarrow$ *MPI_Comm_rank(...)*
10. **if** $prank_{set} = 0$ **then**
11. *Mat*(*src*)← input image
12. $offset_{row} \leftarrow rows_{src} / n$
13. *Mat*(*psrc*)← crop *src*(0, *pset* * $rows_{src}$, $cols_{src}$, $offset_{row}$)
14. *Mat*(*psrc_gray*) ← grayscale *psrc*
15. **for** *node* = 1 to *n* **do**
16. *MPI_Send*($rows_{psrc\_gray}$, $cols_{src}$)
17. **end for**
18. **end if**
19. *MPI_Recv*($rows_{psrc\_gray}$, $cols_{src}$)
20. **Declare** *ppsrc, ppdes*
21. $rows_{ppsrc,\ ppdes,\ ppdes1} \leftarrow rows_{psrc\_gray}$/s
22. $cols_{ppsrc,\ ppdes,\ ppdes1} \leftarrow cols_{src}$
23. *MPI_Scatter(psrc_gray, ppsrc)*
24. **for** *r* = 1 to $rows_{ppsrc}$ **do**
25. **for** *c* = 1 to $cols_{ppsrc}$ **do**
26. *Mat*(*ppdes*) at (*r, c*) ← *G_kernel* (*r,c*)
27. **end for**
28. **end for**
29. **for** *r* = 1 to $rows_{ppdes}$ **do**
30. **for** *c* = 1 to $cols_{ppdes}$ **do**
31. *Mat*(*ppdes*) at (*r, c*) ←*S_kernel*(*r,c*)
32. **end for**
33. **end for**
34. *MPI_Gather(ppdes, psrc_gray)*
35. *MPI_Comm_free*(…)
36. *MPI_Finalize(...)*

2) Canny MPI: Similar procedure is followed in Canny where the steps of edge thinning and weak edge-removal are carried out using the *Mat(ppdes)*. We will see later that these extra steps in Canny algorithm make it far more computationally expensive than Sobel and Prewitt.

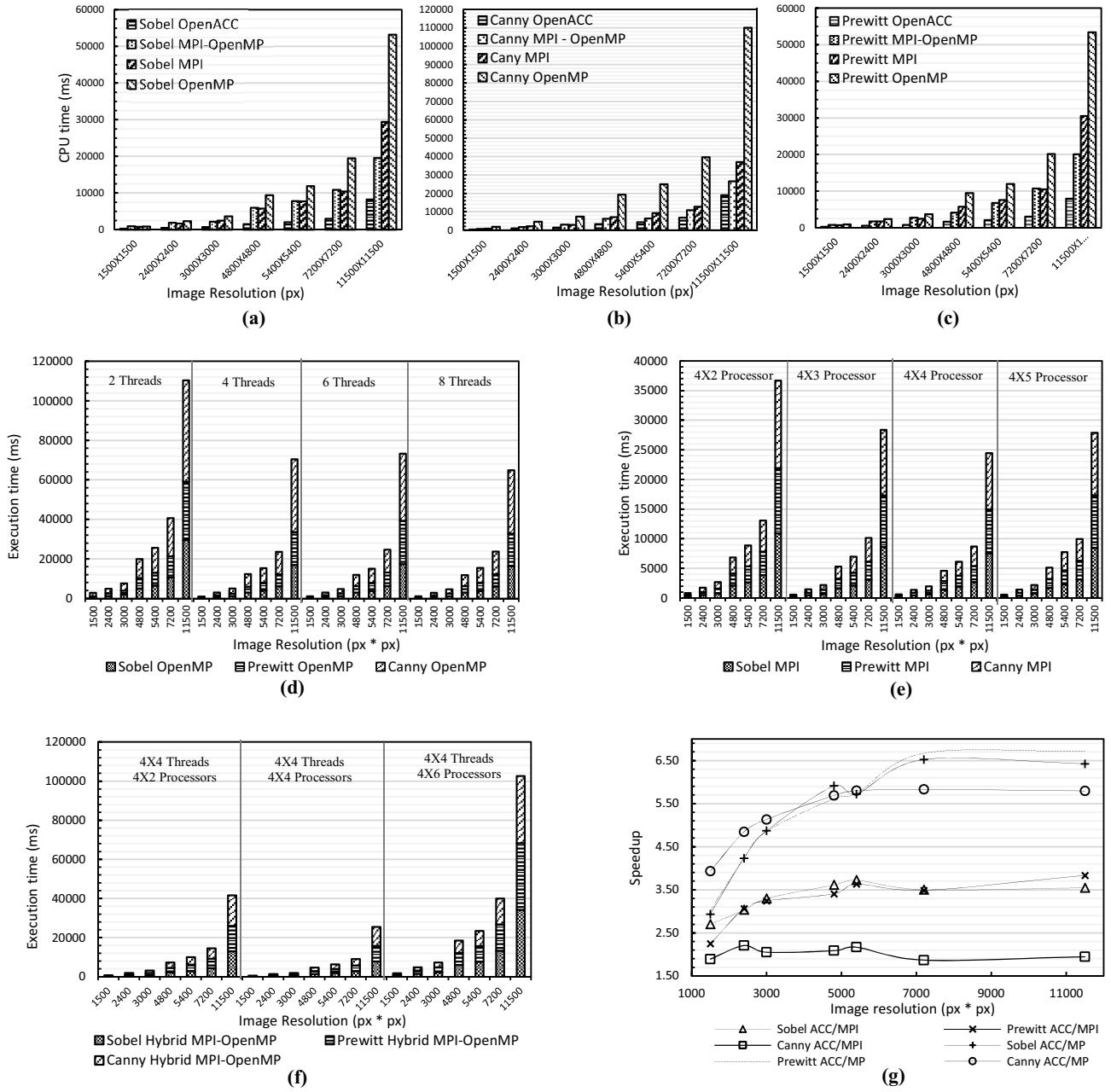

Figure 4. (a)-(c) CPU-time vs. Image resolution; (d)-(f) Execution time vs. Image resolution; (g) Speedup

D. Hybrid OpenMP/MPI Implementation

The current combination is designed to overcome the shortcomings of both the models: OpenMP and MPI. This distributed shared memory processor system puts the memory usage to an optimum level where the memory on each MPI is now being shared among multiple OpenMP threads, thus reducing the chances of memory oversubscription at an early stage. The computing capacity of the cluster can also be better utilized with each process on a node using multiple light-weight threads to access the cores. This results in better scalability and lower latency. For Hybrid Sobel/Prewitt, algorithm 7 is used for the MPI part with the combination of OpenMP *pragmas* (as used at lines 5,6 and 11 in algorithm 6). In algorithm 7 for the *for* loops, we use *pragma omp parallel* on top of line 24 and then parallelize each loop by using *pragma omp* for at lines 24 and 29. For Hybrid Canny, the algorithm is extended to include the last two steps (Algorithms 4 and 5) with OpenMP *pragmas* as earlier seen with the Canny OpenMP implementation.

IV. RESULTS

In this experiment, we compare the performance of our parallel implementations, for the existing algorithms based on their CPU time and execution time (as summarized in Fig. 4). As observed in Fig. 4 (a), (b) and (c), the order of models in the increasing level of performance is OpenMP, MPI, Hybrid MPI-OpenMP and OpenACC which is seen in each edge-detection algorithm. The difference in CPU run times for the Hybrid and MPI approach is not profound for images with a smaller resolution as observed in Sobel (4a) and Prewitt (4b). In some cases, the Hybrid approach even takes longer CPU time than MPI which can be attributed to the combined overheads, thus outperforming the speedup levels for lower image resolutions. The threshold limit for this overtake is 11500X11500 for Sobel, Prewitt and 5400X5400 resolution for the Canny algorithm 4(c). Note that the performance improvement using MPI over OpenMP is largest for Canny among Sobel and Prewitt as seen in Fig. 4(c).

The OpenACC and MPI models consistently maintain good speedups over the OpenMP model with the increasing image resolution, also shown in Fig. 4 (g). Canny implementation is more computationally expensive than Sobel and Prewitt. The reason is that the later step of *Remove_WeakEdges()* offers lesser inherent parallelism than the initial steps. This loss is represented in Fig. 4 (g), where Canny MPI achieves greater speedup over OpenMP but for smaller image resolutions (< 4800x4800). Furthermore, the Canny MPI is overtaken by the rest of its counterparts-SobelACC/MP and PrewittACC/MP. Also, the speedup of CannyACC over MPI is far less than Sobel ACC and Prewitt ACC. Minute differences are observed between the speedups of Sobel and Prewitt.

The second part of this implementation compares the performance using the execution time that is essential in observing performance variations by varying model specific constructs. In case of using OpenMP, performance improves while extending from 2-threaded version to 8-threaded programs running on multiple cores with peak performance achieved for 8-threaded version also shown in Fig. 4(d). Consistently Sobel and Prewitt MP models perform better than the Canny MP. In Fig. 4 (e), the algorithms are evaluated by varying the number of allowable processors on each node in a cluster, where a local minimum can be observed when going from 8 processors to 20 processors. We can deduce that oversubscribing more than a certain threshold limit can also result in performance degradation. In Fig. 4 (f), hybrid models are observed for performance improvement by varying the number of processors utilized on each node and the amount of multi-threading on each node. A serious performance drop occurs for the third configuration of 4x6 processors.

TABLE I. ANALYSIS OF EXECUTION TIMES

Image Resolution 11500 X 11500 (px)			
Execution time (ms)	Hybrid	MPI 4 nodes	OpenMP 4 threads
Sobel	7871	7521	16379
Prewitt	7871	7473	16457
Canny	9637	9469	32135

TABLE II. PERFORMANCE COMPARISON FOR SOBEL, PREWITT AND CANNY ALGORITHMS

Resolution (px) 11500 X 11500	OpenACC	Hybrid	MPI 4 nodes	OpenMP 4 threads
Parallel Sobel				
CPU time (ms)	8292	19564	29436	53272
Speedup/OpenMP	6.42	2.72	1.8	–
Parallel Prewitt				
CPU time (ms)	7960	20082	30516	53468
Speedup/OpenMP	6.72	2.66	1.75	–
Parallel Canny				
CPU time (ms)	19016	26592	37036	110184
Speedup/OpenMP	5.79	4.14	2.97	–

Table I and II above summarize the respective performances in terms of the Execution and CPU time for the three algorithms. Table III below gives the specification of the computational devices used for implementing OpenACC, OpenMP and MPI cluster nodes.

TABLE III. DEVICE CONFIGURATION

Cluster node	
Device Memory	7884 MB
Device Processor	Intel Core i7-3770 CPU
CPU frequency	3.40 GHz
Graphics	Intel Ivybridge Desktop
OS type	64-bit
GPU device	
Graphics Processor	NVIDIA Tesla K40c
CUDA Cores	2880
Graphics Memory	11520 MB
Bus Type	PCI Express X16 Gen2
PCIe Generation	Gen2
Device Memory	5939 MB
Device Processor	Intel Xeon(R) CPU W3530
CPU frequency	2.80 GHz
OS type	64-bit

V. CONCLUSION AND FUTURE WORK

The OpenACC model clearly achieves the greatest speedup of CPU time, which is followed by the Hybrid OpenMP/MPI model that eventually performs better than MPI approach for larger image resolutions accompanied by OpenMP. As shown in Table II, the parallel Prewitt OpenACC obtains the greatest speedup of 6.72 over the OpenMP computation, and there is not much difference upon performance between Sobel and Prewitt algorithms. The former two algorithms are relatively faster than the Canny algorithm in all respects, which is 2 times slower than the Prewitt and Sobel on OpenACC model as shown in Table II.

Based on execution time, OpenMP model achieves the peak performance when an 8-threaded version is applied. While the MPI model shows a maximum improvement for 16 processes (4 per node), the Hybrid OpenMP/MPI model saturates for a configuration of 4 processors and 4 threads per node (in total 4 nodes). In near future, we see the extension of our current cluster system of OpenMPI and OpenACC frameworks by adding more number of GPUs. Furthermore, the edge detection algorithms will be optimized and tested with larger resolutions to observe more profound divergence in accelerations.

VI. ACKNOWLEDGMENT

We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Tesla K40 GPU used for this research.

VII. REFERENCES

[1] N. Abolhassani and H. Kimm, "Performance Analysis on Edge Detection Algorithms for Coastline Image Detection", Visualization, Imaging and Image Processing: Modelling and Simulation: Wireless Communications, 2012

[2] S. Mittal and J. Vetter, "A Survey of CPU-GPU Heterogeneous Computing Techniques", ACM Computing Surveys, vol. 47, no. 4, pp. 1-35, 2015.

[3] Shane Cook., CUDA Programming: A Developer's Guide to Parallel Computing with GPUs, 1st ed. Morgan Kaufmann Publishers, 2013.

[4] Opengl.org. (2017). *OpenGL - The Industry Standard for High Performance Graphics* [Online]. Available: https://www.opengl.org/.

[5] Nvidia.com. (2017). *Parallel Programming and Computing Platform | CUDA | NVIDIA* [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html.

[6] The Khronos Group. (2017). *OpenCL-The open standard for parallel programming of heterogenous systems* [Online]. Available: https://www.khronos.org/opencl.

[7] Openacc.org. (2017). *OpenACC Home* [Online]. Available: http://openacc.org.

[8] M. Misic, D. Dasic and M. Tomasevic, "An analysis of OpenACC programming model: Image processing algorithms as a case study", Telfor Journal, vol. 6, no. 1, pp. 53-58, 2014.

[9] OpenMP.org. (2017). *Home – OpenMP* [Online]. Available: http://openmp.org.

[10] Open-mpi.org. (2017). *OpenMPI: Open Source High Performance Computing* [Online]. Available: https://www.open-mpi.org/.

[11] Hahn Kim, and R. Bond. "Multicore software technologies." Signal Processing Magazine, IEEE 26.6 (2009): 80-89. © 2009 IEEE.

[12] Mathworks.com. (2017). *Parallel Computing Toolbox - MATLAB* [Online]: https://www.mathworks.com/products/parallel-computing.html.

[13] Threading Building Blocks. (2017). [Online] Available: https://www.threadingbuildingblocks.org/.

[14] CilkPlus. (2017). *CilkPlus* [Online] Available: https://www.cilkplus.org/.

[15] Khronos.org. (2017). *Fixed Function Pipeline* [Online] Available:http://khronos.org/openGL/wiki/Fixed_fuction_pipeline.

[16] C. McClanahan, "History and Evolution of GPU architecture", 2010.

[17] Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., & Phillips, J. C. "GPU computing." Proceedings of the IEEE, vol. 96, no. 5, pp. 879-899, 2008

[18] Nvidia.com. (2017). *Cg_language* [Online]. Available: http://developer.nvidia.com/Cg/Cg_language.html.

[19] Msdn.microsoft.com. (2017). *HLSL (Windows)* [Online]. Available:https://msdn.microsoft.com/en-us/library/windows/desktop/bb509561(v=vs.85).aspx.

[20] Khronos.org. (2017). *OpenGL Shading Language* [Online]. Available:https://khronos.org/opengl/wiki/OpenGL_Shading_Language.

[21] K. Ogawa, Y. Ito and K. Nakano, "Efficient Canny Edge Detection Using a GPU", 2010 First International Conference on Networking and Computing, 2010.

[22] M. Misic, D. Dasic and M. Tomasevic, "Use case analysis of OpenACC directives in the implementation of image processing algorithms", 2013 21st Telecommunications Forum Telfor (TELFOR), 2013.

[23] Parth V. Parikh, Bhinjan A. Dalwadi, Ghanshyam D. Zambare, "Sobel Edge Detection Using FPGA-Artix®-7", International Journal of Scientific & Technology Research, vol. 5, no. 6, 2016.

[24] M. Hemnani, "Parallel processing techniques for high performance image processing applications", 2016 IEEE Students' Conference on Electrical, Electronics and Computer Science (SCEECS), 2016.

[25] Ziou, Djemel, and Salvatore Tabbone. "Edge detection techniques-an overview." Pattern Recognition and Image Analysis C/C of Raspoznavaniye Obrazov I Analiz Izobrazhenii 8, pp. 537-559, (1998).

[26] Opencv.org. (2017). *OpenCV library* [Online] Available: http://opencv.org/.

[27] OpenMP.org. (2017). *About Us - OpenMP* [Online] Available: http://www.openmp.org/about/about-us/.

[28] OpenMP.org. (2017). *Members – OpenMP* [Online] Available: http://www.openmp.org/about/members/.