Parallelizing Astronomical Source Extraction on the GPU

Baoxue Zhao
Department of Computer Science and Engineering, HKUST
Kowloon, Hong Kong
Email: bzhaoad@cse.ust.hk

Qiong Luo
Department of Computer Science and Engineering, HKUST
Kowloon, Hong Kong
Email: luo@cse.ust.hk

Chao Wu
National Astronomical Observatories
Chinese Academy of Science
Beijing, P.R.China
Email: wuchao.lamost@gmail.com

***Abstract*—In astronomical observatory projects, raw images are processed so that information about the celestial objects in the images is extracted into catalogs. As such, this source extraction is the basis for the various analysis tasks that are subsequently performed on the catalog products. With the rapid progress of new, large astronomical projects, observational images will be produced every few seconds. This high speed of image production requires fast source extraction. Unfortunately, current source extraction tools cannot meet the speed requirement. To address this problem, we propose to use the GPU (Graphics Processing Unit) to accelerate source extraction. Specifically, we start from SExtractor, an astronomical source extraction tool widely used in astronomy projects, and study its parallelization on the GPU. We identify the object detection and deblending components as the most complex and time-consuming, and design a parallel connected component labelling algorithm for detection and a parallel object tree pruning method for deblending respectively on the GPU. We further parallelize other components, including cleaning, background subtraction, and measurement, effectively on the GPU, such that the entire source extraction is done on the GPU. We have evaluated our GPU-SExtractor in comparison with the original SExtractor on a desktop with an Intel i7 CPU and an NVIDIA GTX670 GPU on a set of real-world and synthetic astronomical images of different sizes. Our results show that the GPU-SExtractor outperforms the original SExtractor by a factor of 6, taking a merely 1.9 second to process a typical 4KX4K image containing 167 thousands objects.**

***Index Terms*—GPU; Source Extraction; SExtractor; Detection**

I. Introduction

Modern astronomical telescopes, such as Hubble [1] and LSST [2], produce large amounts of high-quality digital images, while many data analysis tasks in astronomy, such as calibration, correlation and cross match, focus on catalog data - tables containing information about celestial objects extracted from digital images. For instance, a typical workflow of searching for optical transients, as shown in Fig. 1, consists of source extraction on preprocessed raw images, followed by cross match on the generated catalogs. With the rapid increase in the speed of image production, the speed of source extraction must be improved significantly to keep the pipeline running smoothly and to enable timely data analysis, such as issurance of OT alerts.

As the GPU (Graphics Processin Unit) has accelerated a wide range of scientific computing applications [3] [4], including astronomy applications [5] [6] [7], we propose to use the GPU to accelerate astronomical source extraction. We start with SExtractor (*Source Extractor*) [8], an open source software package widely used for astronomical source extraction. The input file of SExtractor is a digital image in the FITS (Flexible Image Transport System) format and the output is a catalog file consisting of a list of extracted objects.

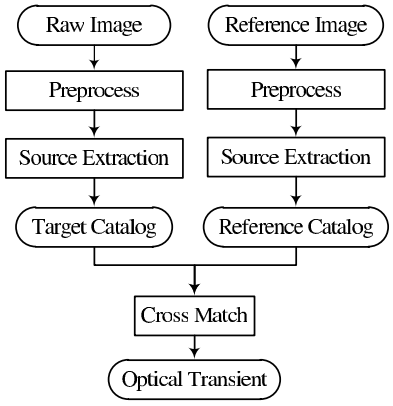

Fig. 1. Source Extraction in Optical Transient Search

The workflow of SExtractor is composed of the following five major steps [9] [10]: (1) Background subtraction, by which the background sky in the input image is subtracted; (2) Raw object detection through scanning the intermediate image resulted from (1); (3) Deblending of raw objects; (4) Cleaning of deblended objects to remove spurious detections; (5) Measurement and output, where the properties of each extracted object is computed and output to the catalog file.

Table I shows the time breakdown of running SExtractor on a desktop PC with an Intel i7 CPU. Statistics about the three real-world astronomical images used in the test are shown in the experiments section. We observe in Table I that (1) with the image size increased to 4KX4K, which is typical for current telescopes, the total time increases to over 13 seconds, far below a speed enabling online processing; and (2) the detection and deblending steps and the cleaning step take around 50% and 30% of the total execution time, respectively.

We further study the implementation of SExtractor and find that the image is scanned multiple times, from background

978-0-7695-5083-1/13 $25.00 © 2013 IEEE
DOI 10.1109/eScience.2013.10

TABLE I
TIME BREAKDOWN OF SEXTRACTOR ON ASTRONOMICAL IMAGES OF VARIOUS SIZES

Time in milli-seconds (%)	2048x2048	3056x3056	4096x4096
background subtraction	92 (9%)	198 (8%)	406 (3%)
detection & deblending	527 (49%)	1146 (47%)	7190 (54%)
cleaning	293 (27%)	727 (30%)	3739 (28%)
measurement	155 (15%)	350 (15%)	1945 (15%)
total	1067	2421	13280

subtraction to object detection and deblending. The pairwise checking between the detected objects for cleaning is also done sequentially. SExtractor indeed supports multi-threading for some of the object measurement functions. However, the most time-consuming detection, deblending and cleaning steps are all sequentially executed.

To facilitate online source extraction and its integration into the astronomical processing pipeline, such as OT finding, we develop GPU-SExtractor using the NVIDIA CUDA (Compute Unified Device Archteciture), following the workflow of SExtractor. First, we copy the FITS input image from main memory to the GPU device memory. Then, we parallelize each step as a GPU kernel program running on many concurrent GPU threads. Finally, we copy the extracted catalog data back to the main memory and output to a catalog file. We make the following technical contributions in GPU-SExtractor:

- We parallelize the entire process of source extraction on the GPU. Therefore, we eliminate any intermediate result transfer between the main memory and the GPU memory on the PCI-E bus, which is often shown as the performance bottleneck in GPU-accelerated programs.
- We propose efficient parallel detection and deblending algorithms, which are based on parallel Connected Component Labelling (CCL) and parallel scan primitives.
- We employ a simple yet efficient parallel cleaning algorithm, which is over 30 times faster than the sequential cleaning algorithm.

The remainder of the paper is organized as follows. Section II introduces the background and related work. III presents the design and implementation of our GPU-SExtractor. We report the performance results in section IV, and conclude in section V.

II. BACKGROUND AND RELATED WORK

In this section we give a brief introduction to SExtractor, on which our work is based. We then discuss related work on GPGPU.

A. SExtractor

Source extraction is an essential step in astronomical data processing. In addition to SExtractor, there are several other source extraction tools, such as the Duchamp source finder

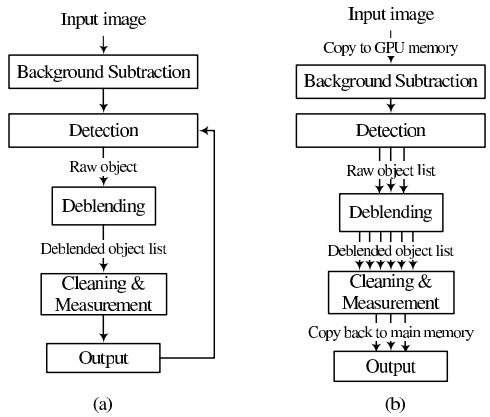

Fig. 2. The workflows of SExtractor and GPU-SExtractor

[11] and the radio interferometry data reduction package Miriad [12]. We chose SExtractor to work on because it is an advanced extraction tool widely used in the astronomy community and it is applicable to optical images, which are our target data form.

Figure 2 (a) shows the workflow of SExtractor. After the input image undergoes Background Subtraction, the Object Detection will extract raw objects. Each raw object then goes through Deblending and the deblended objects from the raw object continues to go through the Cleaning and Measurement component. The final objects with their properties are output to the catalog. The pipeline from detection to output will continue to run until no more raw objects are detected.

1) Background Subtraction: Background subtraction is the first step in SExtractor. As the light value of each pixel is the sum of the background light and the foreground object light at that position, the background light value must be subtracted from the image before object detection. First, the image is divided into a grid of cells, and the local background of each cell is computed using sigma-clipping. Then, the local background values across cells are smoothed with a median filter, and the background value of each pixel is computed by a bi-cubic spline interpolation between the local background values. Finally, the estimated background value at each pixel is subtracted and the image for subsequent object detection (called detection image) is produced by applying a convolution filter on the background-subtracted image.

2) Object Detection: The object detection component takes the resulting image from Background Subtraction as input and outputs a list of raw objects. It is done in two steps: extraction and pre-analysis.

- **Extraction:** This step uses Lutz's extraction algorithm [13]: The image is scanned row by row to identify pixels with a light intensity higher than a threshold (called a primary threshold). Furthermore, neighbouring pixels that have a light intensity higher than the primary threshold are regarded connected and belonging to the same object. The algorithm maintains a list of such pixels for each object under extraction. When a newly scanned pixel is found connected to an object, the new pixel will be

appended to this object's pixel list. If there is no more pixel connected to an object, the extraction of this object is finished and the algorithm checks whether the number of pixels of this object exceeds a threshold. If so, this object will be passed as a raw object to the subsequent pre-analysis step; otherwise, it will be discarded.

- **Pre-analysis:** In this step, the pixel list of an extracted raw object is sequentially traversed to compute some object properties, e.g., integrated and peak light intensity values, which will be used for deblending subsequently.

3) Object Deblending: The primary detection threshold in Object Detection is set low to avoid missing any dim object. However, due to the low threshold, some neighbouring objects may be identified as a single raw object. Therefore, every raw object goes through the deblending component so that a raw object may be split into several objects depending on their brightness values. We call the raw object a parent object and those objects resulted from the split of the parent object its children objects.

SExtractor uses a multi-level thresholding approach for deblending. Each execution of the deblending routine takes a detected raw object, denoted raw_i, as input and outputs a list of children objects of raw_i. The primary threshold used in the Object Detection component is the level 0 deblending threshold, so the raw object detection is also level 0 deblending. The multi-level deblending thresholds from the lowest level to the highest level for raw_i are exponentially or linearly spaced between the primary threshold and a peak value. At each level of threshold, Lutz's algorithm is performed on the area of the detection image that contains raw_i to extract objects that satisfy the current level of threshold.

Since an object extracted at a lower level may be split into several children objects at a higher level, all objects generated from the multi-level deblending of raw_i form an object tree with raw_i as the root. Then, this tree is traversed bottom-up to determine on each node whether the node will be an output object. A node that satisfies all of the following three rules will be an output object:

1) None of its descendants are an output object.
2) The integrated light intensity of the node exceeds a certain ratio of that of the root object raw_i.
3) The node has at least one more sibling that also meets the integrated light intensity ratio property.

With this multi-level deblending, some of the nodes in the object tree are pruned, and the remaining nodes are stored in a list as the intermediate objects deblended from raw_i. Pixels belonging to those pruned objects are re-assigned to these intermediate objects. Finally, this list of intermediate objects is passed to another analysis routine, within which more properties are computed for each object for the subsequent cleaning.

4) Cleaning: Due to the low primary threshold value in object detection, some areas may be falsely identified as objects due to the contribution of other real objects nearby. Therefore, the cleaning process is required to filter the deblended intermediate objects to remove those spurious objects. This cleaning is done through pairwise checking between neighbor objects to see whether an object would be detected if its neighbours were absent.

SExtractor uses a stack to store the deblended objects. When a deblended object arrives, all the objects in the stack are checked to see whether there is a spurious object. If the newly deblended object is spurious, it will be merged to the contributing neighbor object in the stack. If there is a spurious object in the stack to which the new object contributes, these two objects will be merged as one single object in the stack. If the stack is full, the object that is in the lowest row in the image will be removed from the stack and passed to the Measurement component. When all objects are detected, deblended, and cleaned, the remaining objects in the stack will be eliminated and passed to the Measurement.

5) Measurement: The measurement component computes the properties of each object. Recall that part of the measurement is done on an object after it is detected and then on the intermediate objects after deblending. The final measurement is done on each object after it passes the cleaning component.

B. GPGPU

GPUs have been used as parallel computing platforms or hardware accelerators for a wide range of scientific computing applications [4]. Most recently, there has been initial work on accelerating Duchamp using the GPU [14]. That work is not directly applicable to optical images. Furthermore, it does not accelerate the deblending and cleaning steps, which are the most time-consuming in SExtractor.

Currently major GPU vendors include NVIDIA, AMD, and Intel. NVIDIA's CUDA is one of the earliest programming frameworks for GPGPU applications. OpenCL is a later standard for GPU programming across different GPUs. However, the OpenCL implementations are still vendor-specific, and are actively evolving. Additionally, the performance of OpenCL programs are less optimized than that of programs implemented in a framework native to the GPU hardware. Due to performance considerations, we use NVIDIA's CUDA for GPU programming in our work.

In CUDA, developers write GPU programs, called kernels, to process data elements in arrays. Each kernel program is executed by a number of $threadblocks$ in parallel. Developers can specify the number of thread blocks and the number of threads per block; however, thread scheduling is done by the runtime system in the unit of 32-thread $warps$. Threads in a block can share a piece of small (usually at tens of kilobytes) but fast, on-chip memory, called the shared memory. In contrast, the GPU device memory is of several gigabytes and has a high bandwidth, but the latency is also high. A useful feature in GPU memory is coalesced access, which happens when threads in a block access consecutive memory addresses and these accesses are done through a single memory transaction. Consequently, we follow the following principles for performance: (1) data-parallel programming, i.e., utilizing data-parallel primitives such as sort, scan, scatter, filter, and so on, to fully exploit the massive thread parallelism

TABLE II
NOTATIONS USED IN THE PAPER

Notation	Explanation
w, h	The width and height of the input image
μ_{ij}, σ_{ij}	The mean and standard deviation of the light intensity within cell (i, j) of the grid
N_{raw}	Number of raw objects detected
$N_{p>primary}$	Number of pixels above primary threshold
$N_{p \in raw}$	Number of pixels belonging to raw objects
$N_{p \in debj}$	Number of pixels belonging to objects detected at level j
deb_nlevel	Number of deblending levels
$N_j (0 \leqslant j < deb\_nlevel)$	Number of objects detected at level j
$M_j (0 \leqslant j < deb\_nlevel)$	Number of object deblended (after pruning) at level j
M_{total}	Total number of object deblended (after pruning) at all levels
$raw_i (0 \leqslant i < N_{raw})$	The i-th raw object
$t_{ij}, (0 \leqslant i < N_{raw}, 0 \leqslant j < deb\_nlevel)$	The j-th level deblending threshold for the i-th raw object

in the GPU; (2) use of the shared memory for low latency; (3) use of coalesced access for memory bandwith utilization.

III. PARALLEL SOURCE EXTRACTION USING GPU

A. Preliminaries

An input 2-D image is stored in a 1-D floating point array in row-major format, with each element representing the light intensity of the corresponding pixel. To facilitate coalesced memory access in CUDA, a list of objects with n properties are stored in n arrays, with each array storing one of the n properties for all the objects. Table II summaries the notations used in the following parts of the paper.

SExtractor employs an on-the-fly extraction style, which is shown in Fig. 2 (a). Whenever an raw object is detected, it is deblended immediately, and the detection scan does not go on until all these deblended objects are processed and output. While the on-the-fly style saves memory space since it does not need to store all the intermediate result, it is not suitable for GPU parallel implementation. Since the GPU needs all threads in a kernel behave uniformly to achieve better performance. Fig. 2 (b) shows our parallel approach, in which all the raw objects are detected in parallel and stored in GPU memory, and then all raw objects are processed by deblending component simultaneously.

B. Parallel Background Subtraction

Our parallel background subtraction is performed in the following five steps.

Step 1: We use one GPU kernel program to compute the mean and standard deviation (μ_{ij} and σ_{ij}, respectively) for each cell at position (i, j) of the 2-D grid of the image, and construct a histogram. Each block of threads is responsible for one cell. This way, all threads are executing the same task on different pixels concurrently. To reduce access latency, the pixel data are loaded into the shared memory, and the subsequent pixel fetches are performed on the shared memory.

Step 2: The histogram constructed for each cell is clipped by a second GPU kernel program in which clipping is done concurrently for each cell. After the clipping terminates, μ_{ij} and σ_{ij} are re-computed on the remaining pixels of cell (i, j). These statistics represent the current local background of cell (i, j). The local background is then smoothed by a third kernel prorgram, in which thread with index (i, j) reads μ_{ij} and its neighbours within the filter mask, computes the median value from them, and assigns the median to μ_{ij}. The σ_{ij} value for each cell is also smoothed this way.

Step 3: The global background for the entire image is computed by first sorting the local background values (with the sorting primitive in CUDPP [15]) produced in step 2, and then taking the median of the sorted results as the global background. The detection threshold is computed from the global background.

Step 4: The bicubic-spline interpolation is performed in two sub-steps: the first sub-step performs cubic interpolation in the x direction of the image and each thread processes one row of the local background values. The second sub-step performs interpolation in the y direction and each thread processes one column. After this interpolation, we get the background value for each pixel, and subtract it from the original image in parallel.

Step 5: In the final step, the background-subtracted image is processed by a convolution kernel. Each thread is assigned to one pixel. The convolution produces the detection image, which is stored in the detection pixel array (*DPA*), for the subsequent object detection.

C. Parallel Object Detection

Our parallel object detection component also takes the two steps as in the original SExtractor: extraction and pre-analysis. Even though Lutz's extraction algorithm used in SExtractor performs only one scan of the image and is efficient for sequential execution, it is not suitable for parallelization. Therefore, we choose and optimized the parallel Connected Component Labelling(CCL) algorithm [16] for our parallel object detection.

Our parallel object detection works as follows. First, we compact the pixels by filtering out those pixels below the detection threshold and recording the indices of the remaining pixels. Then, we perform our optimized parallel connected component labelling on the indices. Finally, we sort the indices according to their labels to get all connected pixel segments and prune the segments. In the pre-analysis step, we compute the properties of the extracted objects by scanning each pixel segment in parallel using the segmented scan primitive in CUDPP. In the following, we describe these steps in more detail.

1) Parallel Object Extraction: To build the compacted pixel index, we use the compaction mask array (*CMA*) to mark the pixels that are above the detection threshold. The label array (*LA*) and the equivalence array (*EQA*) are used

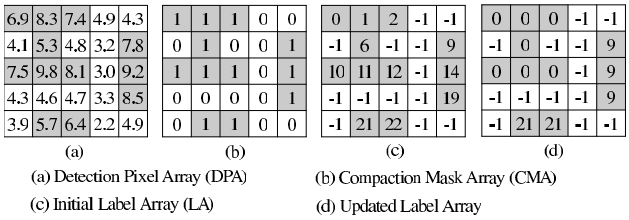

Fig. 3. The pixel array, compaction mask array and label array used for parallel detection. The primary detection threshold is 5.0

to record the label and equivalence information during the iterative labelling process. The extraction is initialized as follows: if a pixel in *DPA* is above the detection threshold, the corresponding compaction mask in *CMA* is set to 1, and the label and equivalence values are set to the current pixel index. Otherwise, the compaction mask is set to 0, and the label and equivalence are set to -1. For example, in Fig. 3, (a) is a 5×5 pixel array with detection threshold 5.0, whereas (b) and (c) are the initialized *CMA* and *LA*, respectively. The *EQA* is initialized the same way as the *LA*. For illustration purpose, all these 1-D arrays are shown in 2-D corresponding to the detection image. Next, we compact the initial *LA* according to the *CMA* using the CUDPP compaction primitive, and get the compacted index array (*CIA*). Fig. 4(a) shows the compacted pixel index array for the pixel array in Fig. 3(a).

Optimized Parallel Connected Component Labelling. In the original parallel CCL (PCCL) algorithm proposed by K.A.Hawick (Kernel *D* in [16]), each thread examines the neighbors of one label. However, we observe that in astronomical images, a large number of pixels are below the detection threshold. For instance, in a test image of size 4096×4096, the total number of pixels above the detection threshold is around 3,6 million, which is less than $1/4$ of the total pixels. As such, many threads, which process these below-threshold pixels, will be idle, in the original PCCL algorithm. Therefore, we propose to eliminate the inactive threads using the compacted pixel index in our optimized PCCL.

Specifically, in our optimized PCCL, we follow the Label Equivalence method [16] with two improvements: 1) We consolidate PCCL's analysis phase and labelling phase within the same iteration into one kernel, and 2) we set the number of threads equal to the *CIA* size. With these improvements, each thread reads one index value from *CIA*, reaches the "real label" pointed by the index, and processes this label the same way as in the original algorithm [16]. Since a scatter operation is required to reach the "real label" from the index value, these "real labels" for consecutive threads may be non-consecutive, in which case the un-coalesced memory access will increase. However, because the kernels are consolidated and the total number of threads needed is reduced dramatically, the performance gain outweighs the un-coalesced memory access overhead. Fig. 3 (d) shows the updated label array after the CCL is finished.

The output of the CCL process is the updated *LA*. Labels belonging to the same connected component have the same

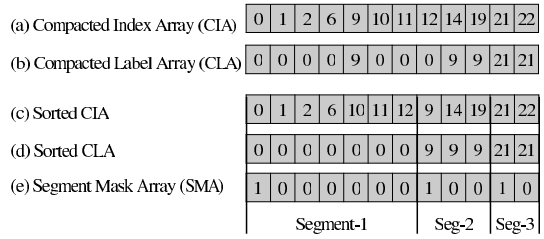

Fig. 4. The arrays used for extraction post-processing

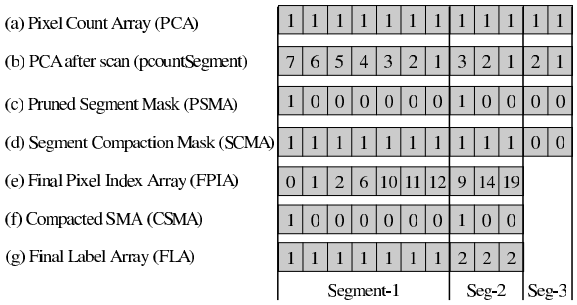

Fig. 5. The arrays used for segment pruning (the *DETECT_MINAREA* is 3)

value, and labels with value -1 correspond to those below-threshold pixels. To facilitate the computation of object properties in the pre-analysis step, the label and pixel arrays need to be rearranged such that the labels with the same values are aligned in a consecutive segment. We perform the post-processing in a few steps. First, we retrieve the label values from the updated *LA* according to the *CIA*, and store them in the compacted label array (*CLA*) (Fig. 4(b)). Second, we sort the *CIA* according to the *CLA* using the CUDPP key-value radix sort primitive. Since the CCL has assigned the same value for labels belonging to the same component, the sorting operation will produce the sorted segments(Fig. 4(c) and (d)). After sorting, we perform a scan on the labels to mark the start position of each segment by comparing each label with its predecessor. If two consecutive labels are different, the second label lies in the start position of a new segment. The markers are stored in the segment mask array (*SMA*), shown in Fig. 4(e).

Segment Pruning. In SExtractor, a group of connected pixels is counted as one object only if the number of pixels exceeds the *DETECT_MINAREA* threshold. We compute the segment lengths by first initializing each element of the pixel count array (*PCA*) to 1 (Fig. 5 (a)) and then performing a backward segment scan on *PCA* to get the pixel count segment (*pcountSegment*). After the scan, the pixel count value at the start position of each segment is the length of that segment (Fig. 5 (b)). Next, we initialize the pruned segment mask (*PSMA*) according to the *SMA* and *pcountSegment*: if the length of a segment exceeds the *DETECT_MINAREA* threshold, the segment start position in *PSMA* will be set to 1; otherwise it will be set to 0. Fig. 5 (c) shows that the start position of segment3 is set to 0 since its segment length is less than 3. The segment compaction mask (*SCMA*) is initialized by performing a segment scan on *PSMA* using *SMA* as the segment mask (Fig. 5(d)).

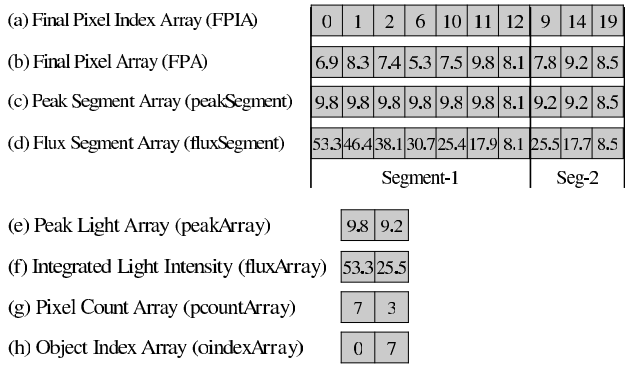

Fig. 6. The arrays used for parallel pre-analysis

The final step of segment pruning is performed by compacting the sorted *CIA* and *SMA* using *SCMA* as the compaction mask. The compacted *CIA* is the final pixel index array (*FPIA*) (Fig. 5 (e)), and the compacted *SMA* (Fig. 5 (f)) is passed to another scan operation to get consecutive labels for adjacent segments as the final label array (*FLA*) (Fig. 5 (g)). The lengths of both *FPIA* and *FLA* are the total number of detected pixels.

2) Parallel Pre-analysis: In the pre-analysis step, four properties of the remaining object segments are computed: peak light, integrated light intensity (flux), pixel count and object index in *FPIA*. More object properties will be computed in later steps. The object properties are computed in several steps instead of once and for all because a large portion of the raw objects will be pruned in deblending and the pixel list of the remaining objects will be updated by pixel re-assignment. Therefore, we compute object properties only when they are needed.

To compute the peak light and flux, we firsst retrieve the pixels of each object segment from the *DPA* according to *FPIA*, and store them in the final pixel array (*FPA*). Secondly, we perform a backward segment max scan and a backward segment sum scan on *FPA* with *CSMA* as the segment mask, and store the scan results in peak segment array (*peakSegment*) and flux segment array (*fluxSegment*), respectively. Last, we get the peak light array (*peakArray*) and flux array (*fluxArray*) by a compaction on *peakSegment* and *fluxSegment*, so that only the values at the segment start position remain. The pixel count property (stored in *pcountArray*) is computed by a compaction on *pcountSegment* using *PSMA* as the compaction mask. The object index property (stored in *oindexArray*) records the position of each object in the *FPIA* and is computed by an exclusive prefix sum scan on *pcountArray*. Fig. 6 (a)-(h) shows an example of computing the four object properties.

D. Parallel Object Deblending

Through parallel object detection, all raw objects are produced at once and fed to the parallel deblending component, which also processes all raw objects simultaneously.

By the definition of multi-level deblending, we have the the following facts:

Fact 1: If a pixel is above level i threshold, it is also above level $i-1$ threshold ($i > 0$).

Fact 2: An object extracted at level i must be contained in another object extracted at level $i-1$ ($i > 0$) in the detection image.

Our parallel object deblending component takes the following steps: 1) initialization, 2) multi-level detection, 3) object tree pruning, and 4) analysis. Algorithm 1 shows the skeleton of the parallel deblending.

Algorithm 1: Parallel deblending

input : A list of detected raw objects, *DPA*, *peakArray*, *FPIA*

output: A list of deblended objects

1 *Initialize MTA and RLA* ;
2 **for** $i \leftarrow 1$ **to** $deb\_nlevel - 1$ **do**
3 $\quad InitLevel(LA, EQA, CMA, i)$;
4 $\quad Compact(FPIA_{i-1}, FLA_{i-1}, CIA, PLA, CMA)$;
5 $\quad ParallelCCL(LA, EQA, CIA)$;
6 $\quad$ *Perform segment pruning and pre-analysis*;
7 *Perform object tree pruning*;

1) Deblending Initialization: In the deblending initialization (Line 1 in Algorithm 1), the multi-level deblending thresholds for each raw object are computed in parallel: each GPU thread reads the peak light of a raw object from *peakArray*, computes the deblending thresholds for the raw object, and stores the result in the multi-level threshold array (*MTA*). Since in the subsequent multi-level detection step we need to get the raw object label given an arbitrary pixel index, we use a root label array (*RLA*) parallel to *LA* to store the raw object label for each pixel. All elements in *RLA* are set to -1 and then initialized in parallel: Each thread reads a pixel index from the *FPIA* and the corresponding raw object label from the *FLA*, and then assigns the label to the elements pointed by the pixel index in *RLA*.

Algorithm 2: Multi-level detection initialization

input : DPA, LA, EQA, CMA, $FPIA_{j-1}$, RLA and MTA, level j

output: Input arrays with values updated

1 **for** *each thread with global index* tid *in parallel* **do**
2 $\quad pid \leftarrow FIPA_{j-1}[tid]$;
3 $\quad rootlabel \leftarrow RLA[pid]$;
4 $\quad pixel \leftarrow DPA[pid]$;
5 $\quad thresh \leftarrow MTA[(j-1)N_{raw} + rootlabel - 1]$;
6 $\quad$ **if** $pixel >= thresh$ **then**
7 $\quad\quad LA[pid] \leftarrow id$;
8 $\quad\quad EQA[pid] \leftarrow id$;
9 $\quad\quad CMA[tid] \leftarrow 1$;

2) Multi-level Detection: In this step (Line 2-6 in Algorithm 1), we perform parallel detection from level 1 to level

$deb\_nlevel-1$. This multi-level detection process differs from the raw object detection in the following three aspects:

1) In deblending, the thresholds used for multi-level detection vary by different raw objects and different levels. The $InitLevel$ routine in Algorithm 1 works as follows: First, the LA and EQA are set to -1, and the CMA is set to 0; then we use Algorithm 2 to initialize them.
2) According to Fact 1, when performing detection at level j, it is sufficient to check only those pixels detected at level $j-1$. Therefore, the $Compact$ routine in Algorithm 1 compacts $FPIA_{j-1}$ and outputs the compacted index array for level j (CIA_j).
3) According to Fact 2, each detected object at certain level j ($j > 0$) has only one parent object, whereas one object can have multiple children. To save memory space, we use a parent label array (PLA_j) to store the parent object label for each object. The FLA_{j-1} is compacted in parallel with the $FPIA_{j-1}$ in $Compact$ routine to maintain the parent label for each pixel.

After the multi-level detection is finished, each level maintains its own output object property lists and pixel index list. The parent-child relations between objects are maintained through the "parent label" property. Since all raw objects are deblended in parallel, the output of the multi-level detection form a forest, with each raw object being the root of a tree in the forest and all children objects (nodes) pointing to their parents.

3) Object Tree Pruning: The parallel object tree pruning routine (Line 7 in Algorithm 1) includes three steps: marking, compaction, and re-assignment. The routine is shown in Algorithm 3.

Algorithm 3: Parallel object tree pruning

input : Object lists detected at each level
output: Intermediate object list after pruning

```
1  for j ← deb_nlevel − 1 to 0 do
2      if j > 0 then
3          decide in parallel whether each level j object
           should be pruned;
4      M_j ← #remaining level j objects;
5      M_total+ = M_j ;
6  allocate pcountArray_deb for total remaining objects;
7  for j ← 0 to deb_nlevel − 1 do
8      if level > 0 then
9          pcountArray_deb+ = M_j ;
10     compact pcountArray_j into pcountArray_deb;
11 pixel reassignment;
```

Marking. In this step (Line 1-5 in Algorithm 3), we decide on whether each node in the forest is an output object by the three rules as in the original SExtractor. We check the nodes from bottom to the top in a tree, and mark the decision for each node in its ok auxiliary property, with 1 meaning the node satisfy the rules and 0 otherwise.

Compaction. In this step (Line 6-10 in Algorithm 3), we remove the nodes that have been marked 0. Specifically, we use the level j *ok* array as the compaction mask to compact the object property arrays, which are $oindexArray_j$, $pcountArray_j$ and $dthreshArray_j$. In Algorithm 3, we illustrate the compaction of pixel count property; other properties are compacted similarly. The output of the compaction represents all the intermediate objects deblended from the raw objects. However, their properties may be changed by pixel reassignment in the next step.

Reassignment. In this step (Line 11 in Algorithm 3), each pixel that belongs to certain pruned node is assigned to an intermediate object that shares the same root with the pruned node. The parallel reassignment takes the following sub-steps.

First, the property arrays of the intermediate objects are sorted based on their root labels so that objects with the same root label form a segment. We reuse the label array (*LA*) to record the assignment map for each pixel and set all labels in the array to -1. The map is then initialized by traversing the pixel list of each intermediate object and assigning the object label to the map.

Second, by Fact 2, we know that $FPIA$ contains all the pixels extracted at each deblending level. We perform the pixel reassignment in parallel: Each thread reads an index from $FPIA$ and gets the label from the assignment map. If the label is non-negative, the pixel belongs to an intermediate object and no reassignment is required. Otherwise, we use the same reassignment approach employed by SExtractor [8]: the thread gets the root label from RLA, and scans the segment of intermediate objects with the same root label. The contribution to the pixel from each object in the segment is computed using a bivariate Gaussian fit to the object profile, and then is transformed into the probability of the pixel belonging to that object. The pixel will be assigned to the object with the highest probability by setting its label in the assignment map to the intermediate object's label.

Last, the *FPIA* is sorted according to the assignment map and stored as $FPIA_{deb}$, with each segment of consecutive indices belonging to the same intermediate object. The intermediate object index array ($oindexArray_{deb}$) and pixel count array ($pcountArray_{deb}$) are also updated according to the start position and length of each segment on the $FPIA_{deb}$

After the object tree pruning is finished, each pixel pointed by $FPIA$ will have been assigned to certain intermediate object. The pixel lists for the intermediate objects are also updated.

E. Parallel Cleaning and Measurement

In our parallel cleaning, we apply an optimized method proposed in the GOODS project [17] for its effectiveness. In this method, the pixels belonging to falsely detected objects are discarded, instead of being merged to their contributing neighbors, to eliminate discontiguous objects. Our parallel algorithm uses a temporary flag array with each element initialized to 0 to mark whether each object is a false detection

or not. The GPU kernel program for cleaning has a total of $M_{total} \times stack\_size$ threads in x-y dimensions. Each thread with the global index (id_x, id_y) checks the id_x-th object and the $(id\_x + id\_y + 1)$-th object in the intermediate object array. If one object can be merged to the other, the flag of the former object will be set to 1, indicating that the former object is a false detection.

Our parallel measurement is completely data-parallel. Each sub-step measurement is performed using one GPU kernel, in which each intermediate object is measured independently by one thread.

F. Memory Consumption

Our GPU-based parallel approach to source extraction operates on all objects concurrently in each phase. The major concern in this approach is the peak memory consumption at any point in time, as the GPU has only a few gigabytes of device memory in total. Therefore, we analyze the memory cost in the following:

In the parallel background subtraction, we use square cells for simplicity. Suppose the image is $w \times h$ and the cell width is m. Then there are totally $(w \times h/m^2)$ cells. The statistics, including the number of pixels, μ, σ and the histogram for each cell, are stored in the GPU memory. Therefore, the space complexity for this component is $O(w \times h/m^2)$. After the estimation is finished, the background is subtracted from the original image and the memory space is freed.

In the parallel detection, CMA, LA, EQA are all integer arrays of a length $w \times h$. The lengths of CIA, CLA, SMA, PCA, $pcountSegment$, $PSMA$ and $SCMA$ are $N_{p>primary}$, the number of pixels above the primary threshold. The lengths of $FPIA$, FLA, FPA, $CSMA$, $peakSement$ and $fluxSegment$ are $N_{p \in raw}$, the number of pixels belonging to raw objects. The lengths of $peakArray$, $fluxArray$, $pcountArray$ and $oindexArray$ are N_{raw}, the number of raw objects. We use 4 bytes to store each integer and float. Therefore, totally $12w \times h + 28N_{p>primary} + 24N_{p \in raw} + 16N_{raw}$ bytes of memory will be consumed.

In the parallel deblending, the lengths of MTA and RLA are $N_{raw} \times deb\_nlevel$ and $w \times h$, respectively. In the multi-level detection step, we reuse the temporary arrays from the detection step and allocate new space for $FPIA_j$, FLA_j, PLA_j $pcountArray$, $oindexArray$, $fluxArray$ and $dthreshArray$, in each level of detection. In the object tree pruning step, we allocate seven arrays to store the deblended intermediate object properties, and each array has a length of M_{total}, the total number of deblended objects. Therefore, the extra space required in parallel deblending is

$N_{raw} \times deb\_nlevel + w \times h + \sum_{j=1}^{deb\_nlevel-1}(20N_j + 8N_{p \in debj}) + 28M_{total}$ bytes, where N_j is the number of objects detected at level j and $N_{p \in debj}$ is the number of pixels belonging to the objects detected at level j.

In the parallel cleaning, we need one flag array of length M_{total}. In the parallel measurement, we need to store the final properties for each objects. Suppose each object needs t bytes, totally $t \times M_{total}$ bytes will be required.

TABLE III
INFORMATION OF TESTING IMAGES

Type	ID	Size in Pixels	# expected objects	# pixels above primary threshold
R	1	1601×1601	$5,244$	$93,306$
R	2	2048×2048	$15,809$	$672,832$
R	3	3056×3056	$35,845$	$1,451,646$
R	4	4096×4096	$167,344$	$7,782,654$
S	5	1024×1024	$4,752$	$415,953$
S	6	2048×2048	$19,461$	$1,659,577$
S	7	3072×3072	$42,951$	$3,716,156$
S	8	4096×4096	$75,854$	$6,620,759$
S	9	4096×4096	$9,423$	$1,186,699$
S	10	4096×4096	$30,112$	$3,484,201$
S	11	4096×4096	$50,140$	$5,089,089$
S	12	4096×4096	$70,233$	$6,281,493$
S	13	2048×2048	$38,907$	$2,224,075$
S	14	3072×3072	$35,787$	$3,347,649$
S	15	4096×4096	$34,956$	$3,761,026$

TABLE IV
PERFORMANCE OF BACKGROUND SUBTRACTION (MILLISECONDS)

Image	GPU	CPU	Image	GPU	CPU
5	25.51	28.86	9	309.94	323.01
6	73.73	87.90	10	309.95	329.63
7	173.78	183.19	11	315.43	325.60
8	313.59	332.66	12	310.33	323.44

In our experiments with real-world and synthetic astronomical images, our GPU-EXtractor works well within the 4GB device memory on our desktop machine.

IV. PERFORMANCE EVALUATION AND ANALYSIS

A. Experimental Setup

We evaluated our GPU-EXtractor in comparison with the original SExtractor on a desktop PC, which is equipped with an Intel Core i7-3770 CPU with 32GB main memory and an NVIDIA GeForce GTX 670 GPU. It runs the Ubuntu Linux 12.04 Desktop x64 operating system, with CUDA SDK v5.0 and SExtractor 2.8.6 installed. The GPU has 7 multiprocessors with 192 CUDA cores per multiprocessor, and 4GB device memory. The transfer bandwidth of the PCI-e bus between the GPU and the CPU is 4GB/second.

We used both synthetic and real astronomical images in our evaluation. The synthetic images are generated using SkyMaker [18]. Table III show the information about the images we used. Type R means real image and type S means synthetic image.

B. Performance Results and Analysis

We compare the execution time of each phase in the original SExtractor (CPU) and our GPU-EXtractor (GPU). For background subtraction, we use two groups of simulated images. The first group includes image #5, #6, #7 and #8, while the

TABLE V
PERFORMANCE OF OBJECT DETECTION (MILLISECONDS)

ID	GPU Time	CPU Time
13	$18.09 + 3.46$	$13.21 + 34.81$
14	$23.93 + 4.18$	$29.38 + 38.92$
15	$28.83 + 4.63$	$52.33 + 40.78$
9	$17.26 + 2.18$	$169.91 + 10.85$
10	$29.13 + 4.31$	$213.32 + 36.91$
11	$38.20 + 5.50$	$246.44 + 62.83$
12	$44.51 + 6.66$	$272.81 + 98.34$

second group include image #9, #10, #11 and #12. Table IV shows the result. The first group of images have nearly the same object density but different image sizes, whereas the second group have the same image size but different object density values. On the first group we note that the time for both CPU and GPU increases significantly as the image size gets larger, whereas on the second group, both the GPU and CPU time keep nearly constant for different object densities. This effect is because the background subtraction involves no object operation, thus the time is only affected by image size. On both groups of images, the GPU time is slightly shorter than the CPU time.

We study the performance of object detection using two groups of simulated images - image #13, #14, #15 as the first group and image #9, #10, #11 #12 as the second group. The first group of images are of different image sizes but the number of objects contained in each image is similar. The second group are images of the same size and different object densities. We measure the time consumption of extraction and pre-analysis steps separately and show the results in the form of extraction time + pre-analysis time in Table V. On the first group we see the extraction time increases with the image size. This increase is because the extraction step involves detecting pixels from images. However, the GPU extraction time increases much slower than the CPU extraction time since we have adopted an optimized CCL approach and the number of threads required is reduced. In the pre-analysis step, since all the scan and compaction operations involve only the detected pixels, the time is independent of the image sizes and is only related with the number of detected objects and pixels. On the second group of images, we observe that both the extraction time and pre-analysis time increases with the number of objects, and the GPU speedup on pre-analysis is much higher when there are more objects.

We study the performance of the deblending phase similar to what we do with object detection, and the results are shown in Table VI. We see in the results that the GPU deblending speedup increases with the image size and the number of expected objects.

Table VII shows the performance of the cleaning component. The results on the first group of images show that image size has little effect on the cleaning time, but the time increases significantly with the number of expected

TABLE VI
PERFORMANCE OF DEBLENDING (MILLISECONDS)

Image	GPU	CPU	Image	GPU	CPU
13	434.59	2214.91	9	256.61	549.39
14	523.37	2151.40	10	523.50	1784.97
15	561.94	1988.51	11	724.42	2969.99
			12	881.66	4093.45

TABLE VII
PERFORMANCE OF CLEANING COMPONENT (MILLISECONDS)

Image	GPU	CPU	Image	GPU	CPU
13	27.57	843.26	9	5.26	174.10
14	23.75	768.10	10	19.60	643.38
15	22.82	752.15	11	33.47	1096.41
			12	47.71	1548.25

objects. This phenonmenon is because cleaning involves only pairwise checking among objects without any image operation. Compared with the other components, the GPU achieves a much higher speedup on cleaning: For all images in Table VII, the GPU cleaning algorithm is more than 30 times faster than the CPU counterpart. The reason is that the sequential pairwise checking among deblended objects is a very time consuming process, which takes $1/3$ of SExtractor's overall execution time. In contrast, the pairwise checking is fully parallelized on the GPU with each thread performing one checking operation independent of the others.

Table VIII shows the performance of the final measurement component on the CPU and the GPU. The measurment time slightly increases among the first group of images with the increase of image size. This increase is mainly due to the increased object size in number of pixels: the more pixels there are in each object, the longer the measurement takes. Similar to the performance trend in the cleaning component, the measurement time increases with the number of expected objects. Nevertheless, this increase is much less significant on the GPU than on the CPU. As a result, the GPU has a higher speedup when there are more objects. The highest speedup is about 6.3x, on image #12.

Finally, we report the overall performance on the CPU and the GPU on three real-world astronomical images in Fig. 7. The GPU time is the end-to-end time, includes the time of transferring the input image from the host memory into the device memory, performing parallel source extraction on the

TABLE VIII
PERFORMANCE OF MEASUREMENT COMPONENT (MILLISECONDS)

Image	GPU	CPU	Image	GPU	CPU
13	70.45	481.24	9	74.38	137.20
14	91.24	491.73	10	124.10	444.57
15	113.34	499.71	11	133.95	715.57
			12	152.29	962.17

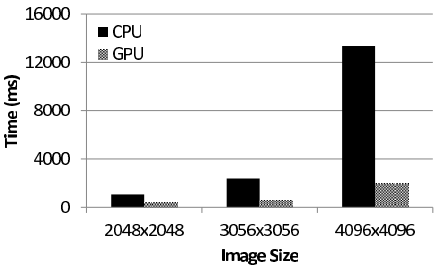

Fig. 7. In-Memory Overall Performance Comparison

GPU, and copying the final result from the device memory back to the host memory. The time of reading images from disk and writing results to disk is excluded. The figure shows that the GPU outperforms the CPU significantly, and the speedup increases with image size and number of objects. The highest speedup is at 6.7x, on image with size 4096x4096.

V. Conclusion

We have presented the design and implementation of GPU-SExtractor, a parallel source extractor based on the NVIDIA CUDA. Our GPU-SExtractor has the same functionality as the widely used, CPU-based SExtractor, but runs entirely on the GPU. In contrast to the pipelined execution of SExtractor, we adopt the data-parallel execution for the entire workflow of GPU-SExtractor. Furthermore, this data-parallelism is implemented effectively in each component of GPU-SExtractor: we adopt and optimize the Parallel Connected Component Labelling algorithm for the object detection, parallelize the mutli-level detection and deblending through data parallel primitives on auxiliary arrays, and parallelize an optimized method for object cleaning. We have evaluated our GPU-SExtractor in comparison with SExtractor. The results show that, despite the fact that astronomical source extraction software is complex in the workflow and processing, we have effectively parallelized it on the GPU. The speedups of our parallel approach increases with the image size and the expected numbers of detected pixels and objects. In particular, on real-world astronomical images, we achieve a speedup of up to 6.7 times, reducing the overall time from over 13 seconds to a couple of seconds.

Acknowledgement

This work was supported by grants 616012 and 617509 from the Hong Kong Research Grants Council and M-RA11EG01 from Microsoft SQL Server China R&D. The work of Chao Wu was also supported by the National Basic Research Program of China (973 Program 2009CB824800), and National Natural Science Foundation of China (grant 10903010).

References

[1] NASA. (2013) NASA - Hubble Space Telescope. [Online]. Available: www.nasa.gov/hubble/

[2] C. W. Stubbs, D. Sweeney, J. Tyson, and L. Collaboration, "An overview of the large synoptic survey telescope (lsst) system," in *Bulletin of the American Astronomical Society*, vol. 36, 2004, p. 1527.

[3] J. D. Owens, M. Houston, D. Luebke *et al.*, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[4] C. Yang, W. Xue, H. Fu *et al.*, "A peta-scalable cpu-gpu algorithm for global atmospheric simulations," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2013, pp. 1–12.

[5] R. B. Wayth, L. J. Greenhill, and F. H. Briggs, "A gpu-based real-time software correlation system for the murchison widefield array prototype," *Publications of the Astronomical Society of the Pacific*, vol. 121, no. 882, pp. 857–865, 2009.

[6] A. Hassan, C. Fluke, and D. Barnes, "A distributed gpu-based framework for real-time 3d volume rendering of large astronomical data cubes," *Publications of the Astronomical Society of Australia*, 2012.

[7] C. J. Fluke, D. G. Barnes, B. R. Barsdell *et al.*, "Astrophysical supercomputing with gpus: critical decisions for early adopters," *Publications of the Astronomical Society of Australia*, vol. 28, no. 1, pp. 15–27, 2011.

[8] E. Bertin and S. Arnouts, "Sextractor: Software for source extraction," *Astronomy and Astrophysics Supplement Series*, vol. 117, no. 2, pp. 393–404, 1996.

[9] Bertin, E. (2006) SExtractor v2.5 Users manual. [Online]. Available: http://terapix.iap.fr/IMG/pdf/sextractor.pdf

[10] B. W. Holwerda, "Source extractor for dummies v5," *arXiv preprint astro-ph/0512139*, 2005.

[11] M. T. Whiting, "Duchamp: a 3d source finder for spectral-line data," *Monthly Notices of the Royal Astronomical Society*, vol. 421, no. 4, pp. 3242–3256, 2012.

[12] R. J. Sault, P. J. Teuben, and M. C. Wright, "A retrospective view of miriad," *Astronomical Data Analysis Software and Systems IV*, 2006.

[13] R. Lutz, "An algorithm for the real time analysis of digitised images," *The Computer Journal*, vol. 23, no. 3, pp. 262–269, 1980.

[14] G. Resnick, M. Kuttel, and P. Marais, "GPU Accelerated Source Extraction in Radio Astronomy: A CUDA Implementation," Department of Computer Science, University of Cape Town, Tech. Rep., 2010.

[15] (2011) CUDPP: CUDA Data Parallel Primitives Library. [Online]. Available: http://www.gpgpu.org/developer/cudapp

[16] K. A. Hawick, A. Leist, and D. P. Playne, "Parallel graph component labelling with gpus and cuda," *Parallel Computing*, vol. 36, no. 12, pp. 655–678, 2010.

[17] E. Vanzella, S. Cristiani, M. Dickinson *et al.*, "The great observatories origins deep survey," *Astronomy and astrophysics*, vol. 454, no. 2, pp. 423–435, 2006.

[18] E. Bertin, "Skymaker: astronomical image simulations made easy," *Mem. Soc. Astron. Ital*, vol. 80, pp. 422–428, 2009.