Accelerating Astronomical Image Subtraction on Heterogeneous Processors

Yan Zhao, Qiong Luo, Senhong Wang
Department of Computer Science and Engineering
Hong Kong University of Science and Technology
Email: yzhaoak, luo, swangam@cse.ust.hk

Chao Wu
National Astronomical Observatories
Chinese Academy of Sciences
Email: cwu@bao.ac.cn

***Abstract*—Image subtraction is an effective method used in astronomy to search transient objects or identify objects that have time-varying brightness. The state-of-the-art astronomical image subtraction methods work by taking two aligned images of the same observation area, calculating a space-varying convolution kernel for the two images, and finally obtaining the difference image using the convolution kernel. With the need for fast image subtraction in astronomy projects, we study the parallelization of HOTPANTS, a popular astronomical image subtraction package by Andrew Becker, on multicore CPUs and GPUs. Specifically, we identify the components in HOTPANTS that are data parallel and parallelize these components on the GPU and multicore CPU. We divide the work between the CPU and the GPU to minimize the overall time. In the GPU-based components, we investigate the suitable setup of the GPU thread structure for the computation, and optimize data access on the GPU memory hierarchy. Consequently, P-HOTPANTS (our parallelized HOTPANTS), achieves a 4-times speedup over the original HOTPANTS running on a desktop with an Intel i7 CPU and an NVIDIA GTX580 GPU.**

I. Introduction

Searching for celestial objects that are transient, moving, or changing in brightness, such as supernova, is an important task for astronomy projects. One effective method for such tasks is image subtraction, in which an image is compared with a reference image taken on the same patch of sky to produce a difference image [Tomaney & Crotts 1996, Alard & Lupton 1998, Alard 2000]. The state-of-the-art astronomical image subtraction methods [Alard 2000] use a space-varying kernel to convolute the two images such that the difference between the two images, in the form of least square, is minimized. As a result, these methods work well for not only fields with a lot of celestial objects but also fields of less object density, such as those at a high galactic latitude, which are often targeted for supernova search.

In several upcoming astronomy survey projects, images on sub-areas of the sky will be produced every few seconds and detection of transient objects will be done on the aligned images online so that alerts can be generated timely. However, current CPU-based astronomical image subtraction packages typically take tens of seconds to perform the task on a pair of images, which cannot satisfy the speed requirement. There has been some initial work investigating the use of the GPU (Graphics Processing Unit) for the image convolution using a spatially-varying kernel [Hartung et al. 2012], which shows promising performance results. In this paper, we further develop P-HOTPANTS, a complete image subtraction package that takes advantage of both the multicore CPU and the GPU.

P-HOTPANTS is developed by parallelizing Andrew Becker's popular software package HOTPANTS (High Order Transformation of Psf and Template Subtraction) [Becker]. HOTPANTS is a CPU-based sequential implementation of the Alard's algorithm [Alard 2000] and has been widely used in astronomy projects. We first study the entire workflow of HOTPANTS and identify the components that are data parallel and computation intensive. We find that both the generation of the space-varying convolution kernel and the convolution process using the kernel are highly parallelizable. Furthermore, these two components take around 16% and 67% of the overall time respectively. Therefore, we parallelize these two components on the GPU.

The GPUs on a desktop typically contain hundreds of compute cores capable of running tens of thousands of active threads concurrently. The programming frameworks on the GPU, such as NVIDIA's CUDA (Compute-Unified Device Architecture), support the creation of thread blocks to execute the same GPU kernel program on 1-dimensional to 3-dimensional data arrays; however, the runtime scheduling of threads is managed by the underlying system. As such, the GPU is best for data-parallel, computation-intensive programs with a simple control flow. Additionally, even though multicore CPUs are not as massively parallel as the GPU, they are capable of supporting a considerable amount of parallel computation. Therefore, we not only utilize the GPU for the two data-parallel computation components but also assign part of the image convolution work to the multicore CPU for the best overall performance. The amount of work assigned to the CPU is based on our analysis of the computation speed of the GPU and the CPU.

In the parallelization on the GPU, a significant challenge is the setup of the GPU thread structure for the computation. Specifically, as we use NVIDIA CUDA for GPU programming, the GPU threads can be structured as 1-, 2-, or 3-dimensional blocks, and these threads are scheduled in the unit of 32-thread warps. In contrast, generating space-varying convolution kernels in HOTPANTS involves identifying a fixed number of stamps, which are sub-images that are centered around bright objects in the original image, and the image convolution process is done on sub-images of a pre-defined

978-0-7695-5083-1/13 $25.00 © 2013 IEEE
DOI 10.1109/eScience.2013.23

kernel size. Therefore, we analyze these parameters in the original HOTPANTS and design the number of dimensions as well as the dimension size of the GPU thread blocks to increase the thread parallelism while keeping the hardware resources, e.g. registers and shared memory, at a suitable amount for the best overall performance.

We have implemented P-HOTPANTS with CUDA on the GPU and OpenMP on the CPU, and compared with the original HOTPANTS, the CUDA-HOTPANTS without OpenMP parallelization on the CPU, and the OpenMP-HOTPANTS without the CUDA-based GPU parallelization. We tested these programs on three pairs of real-world astronomic images with sizes 1K x 1K, 2K x 2K, and 3K x 3K pixels respectively. Our results on a desktop with an Intel i7 CPU and an NVIDIA GTX580 show that (1) the OpenMP-based CPU parallelization, the CUDA-based GPU parallelization, and P-HOTPANTS achieve a speedup of around 3, 4, and 8 times, respectively, on the image convolution component; and (2) P-HOTPANTS is around 4 times faster than the original HOTPANTS on the overall performance.

The remainder of the paper is organized as follows: Section II presents the background of astronomical image subtraction and related work. Section III details our design and implementation of P-HOTPANTS. Section IV reports our experimental results, and Section V concludes the paper.

II. Background and Related Work

In this section, we first review the Optical Image Subtraction(OIS) method [Alard 1998] with a space-varying convolution kernel [Alard 2000], on which HOTPANTS is based. Then, we describe the workflow of HOTPANTS, on which our work is based. Finally, we discuss related work on GPGPU (General-Purpose computing on the GPU), in particular, that related to astronomical data processing.

A. OIS with a space-varying kernel

Image subtraction is an effective method for observing time-varying objects in astronomy; however, pictures taken at different times cannot be subtracted directly even though they are taken by the same apparatus on the same patch of sky, due to changes in atmospheric conditions and optical fluctuations [Alard 2000]. Fortunately, the point spread functions (PSF) of optical images are temporally invariant transfer functions. Therefore, the OIS method computes a *convolution kernel* to match the PSFs of two astronomical images.

Specifically, in OIS, the convolution kernel is a two-dimensional array of a pre-defined size. Thus, it can be viewed as a small image or box. In the following, we describe the OIS method with a space-varying kernel. Notations are listed as follows:

(x,y) - pixel-location in images
(u,v) - pixel-location in the *kernel* box
I - input image
T - template (or reference image)
O - output image (or different image)
$a_n(x,y)$ and $K_n(u,v)$ - the coefficient and basis function of the *kernel* (the default basis function is Gaussian)

The goal of the algorithm is to calculate $O(x,y) = T(x,y) \otimes K(u,v) - I(x,y)$ such that the output image yields a constant field except some significant deviations that reflect the variations in the brightness or spatial locations of the celestial objects.

To find a suitable *kernel* for the image subtraction, we need to minimize $\sum(T(x,y) \otimes K(u,v) - I(x,y))^2$. This minimization is then transformed into a linear least-squares problem with positional weighted basis functions:

$$K(u,v) = \sum_{n=1}^{N} a_n(x,y) K_n \tag{1}$$

The parameter (x,y) in a_n indicates the spacial-varying feature of the *kernel*. The basis function in Eq.(1) is the Gaussian function basis (GFB) by default, which is symmetric:

$$K_n(u,v) = e^{-(u^2+v^2)/2\sigma_k^2} u^i v^j \tag{2}$$

To determine the coefficient a_n of the *kernel*, we solve it with a linear system:

$$Ma = B$$

$$M_{ij} = \int [T \otimes K_i](x,y) \frac{[T \otimes K_j](x,y)}{\sigma(x,y)^2} dxdy \tag{3}$$

$$B_i = \int I(x,y) \frac{[T \otimes K_i](x,y)}{\sigma(x,y)^2} dxdy \tag{4}$$

K_i and K_j refer to the K_n in Eq(2). After we compute K_n, we can fill out both M_{ij} and B_i and perform matrix transposition to get a_i.

B. Hotpants

HOTPANTS follows the method proposed in [Alard 2000]. It is a serial program written in C. Algorithm 1 contains the pseudo code of HOTPANTS.

We divide Algorithm 1 into five steps - initialization (lines 1-2), filling (lines 3-10), checking (lines 11-20), convolution (lines 21-23), and output (line 24). The first two steps are conducted on both the input image and the template. In particular, both images are divided into a fixed number of stamps in step 1, and the parameters of the stamps, including the centers of the sub-stamps (the interesting area within a stamp) are filled in step 2.Then, step 3 chooses a subset of the stamps that are of higher quality than the others and use it to fit the *kernel*. Finally, convolution is done in step 4 and result outputs in step 5. In the following, we describe each step in more details.

- Initialization
  This step allocates memory space and reads the input image and the template into memory. Both images are stored in files in the FITS (Flexible Image Transport System) format. It computes K_n in Eq(2), divides both images into n_s stamps, and initialize the parameters for

Algorithm 1 Hotpants

```
Input:
    input image I and template T
Output:
    output difference image O and its estimated quality
1:  $K_n(u,v) \leftarrow e^{-(u^2+v^2)/2\sigma_k^2} u^i v^j$
2:  divide I and T into $n_s$ stamps
3:  for each stamp of I and T do
4:      filling the parameters for each stamp, including the locations of sub-stamp centers;
5:      $W_m \leftarrow [T \otimes K_m](x,y)$;
6:      $Q_{m1,m2} \leftarrow \sum_{x,y}[W_{m1} * W_{m2}](x,y)$;
7:      $b_m \leftarrow \sum_{x,y} I(x,y) * W_m](x,y)$
8:  $n = q + m * nc_1$
9:  $M_{n1,n2} \leftarrow \sum_{n_s} Q_{m1,m2} * P_{q1} * P_{q2}$
10: $B_n \leftarrow \sum_{n_s} b_m * P_q$
11: while some sub-stamps are clipped out do
12:     for each remaining stamp $S$ do
13:         if its sub-stamp is an outlier then
14:             if no other sub-stamp is available then
15:                 reject $S$
16:             else
17:                 replace the outlier with its neighbouring sub-stamp
18:                 update $W_m$, $Q_{m1,m2}$, $b_m$
19:     Update $M_{n1,n2}$ and $B_n$
20: $a \leftarrow M^{-1}B$
21: for each kernel-size image do
22:     $K(u,v) \leftarrow \sum_n a_n(x,y) K_n(u,v)$
23:     $O(x,y) \leftarrow T(x,y) \otimes K(u,v) - I(x,y)$
24: return difference image and kernel-sum
```

each stamp, including the locations of three sub-stamp centers.

- Filling
  This step computes W_i, Q_{ij} and b_i for each stamp of the input image and the template. Then it computes M_{ij} and B_i without the denominators $\sigma(x,y)^2$ in Eq(3) by summing up Q_{ij} and b_i for the input image and the template respectively. Not shown in the algorithm is that at the end of this step, the algorithm decides whether the convolution will be performed on the input image or the template, based on some quality metric of the two images. The quality metric is initialized as the kernel sum and can also be the average pixel residual, the width of the histogram of residuals, or some other metric defined by users.
- Checking
  Suppose step 2 decides to perform convolution on the input image. This checking step is done in two sub-steps.
  Substep 1 (lines 11-18):
  For each stamp S of the image, do:
  - If an outlier is found, sigma-clip it out and replace it with a neighboring sub-stamp in the same stamp, and update W_i, Q_{ij}, b_i and other information of S;
  - If all sub-stamps in S have been sigma-clipped out, reject S;

  Substep 2 (lines 19-20):
  Update M_{ij} and B_i by accumulating the Q_{ij} and b_i of high-quality stamps and compute the kernel sum to check. If no sub-stamps are clipped out, we use M_{ij} and B_i to get the *kernel* solution; otherwise, go back to substep 1.
- Convolution
  Compute the space-varying *kernel* in Eq(1) and perform convolution.
- Output
  Write the difference image into the output file and compute the RMS (Root-Mean-Square) of the distribution across sub-stamps [Becker]

C. Related Work

General-purpose computing on graphics processing units (GPGPU) has greatly expanded the range of applications of high-performance computing(HPC), including astronomy [Fluke et al. 2010]. Astronomers facing problems on processing and analyzing great amounts of new astronomical data start to looking into exploiting the GPUs' computational power [Berriman 2011]. The CUDA programming framework of NVIDIA has greatly improved the programmability of graphics hardware and helped the utilization of the GPU's massive thread parallelism in applications. Recent interests are also on the OpenCL standard; nevertheless, due to its short time on the market and variations between vendor implementations, much fewer GPGPU applications are done in OpenCL.

For solving the general image convolution problem in CUDA, NVIDIA's CUDA programming manual [Podlozhnyuk] provides a lot of guidance, including the solution to the some problems like Apron. Specific to astronomical image subtraction, initial work [Hartung et al. 2012] has focused on accelerating on the GPU the convolution process with the Dirac delta function basis (DFB), which is more computation-intensive than the Gaussian function basis. An earlier paper [Ryoo et al. 2008] also studied image convolution as one of the GPGPU applications. Neither work considers the entire process of image subtraction, including the construction of the convolution kernel.

There have been recent studies on utilizing hybrid systems with both multicore CPUs and GPUs [Philip et al. 2011] [Yang 2012]. So far, little work has been done in the area of astronomical data processing on such hybrid systems. In typical CUDA-based applications, the CPU programs are responsible for starting the GPU kernels, and the GPU kernels are in charge of the massive computation using "brute-force parallelization"[Berriman 2011]. In comparison, we go further to utilize the computation power of both the GPU and the CPU to archive the best overall performance.

C. Alard's work has made great contribution to image subtraction. Tools based on it have been used in several large astronomical projects, such as Large Synoptic Survey Telescope (LSST)[Heller 2005], the Panoramic Survey Telescope & Rapid Response System (Pan-STARRS). We expect our P-HOTPANTS to be useful for upcoming new astronomical projects, especially the GWAC project in China.

III. Design and Implementation

In this section, we first analyze the performance and algorithmic features of HOTPANTS to get initial design ideas for P-HOTPANTS. Then we discuss our detailed design of P-HOTPANTS as well as the implementation issues. Finally, we consider CPU-GPU co-processing on the convolution through data partitioning.

A. Design Ideas

Recall that the workflow of HOTPANTS is divided into five steps (1)Initialization, (2) Filling, (3) Checking, (4) Convolution, and (5) Output. We first run the original HOTPANTS on our experimental machine and observe its performance breakdown by these algorithmic steps to get some design ideas.

TABLE I
Time Breakdown of HOTPANTS

Step	Init.	Fill	Check	Conv.	Output	Total
Time(seconds)	0.9	1.52	0.38	7.96	1.04	11.8
Percentage(%)	7.6	12.9	3.2	67.5	8.8	100

Table I shows the time spent on each step in HOTPANTS and its percentage of the total time, based on the running time on a 3K x 3K real-world astronomic image.

First, we see that step 1 Initialization takes less than a second, which is less than 8% of the total time. As Initialization is mostly serial involving file reading and memory allocation, we must keep this part on the CPU. Also, the computation of K_n consists of base-*e* exponentials, which may cause numerical result differences on the CPU and on the GPU. Since this K_n is the basis of the subsequent computations, we decide to keep its computation on the CPU to produce an identical result to that in the original HOTPANTS. Finally, given the computation of K_n can be parallelized, we will use OpenMP to parallelize this computation on the CPU,

We see in the table that step 2, Filling, takes around 1.5 seconds, 13% of the total time. As the parameter Filling step is entirely computation, we will transfer all data to the GPU, and parallelize the computation on the GPU. Because this computation is by the unit of stamps, the GPU-based parallelization will also assign stamps of data to each block of GPU threads. When we examine the computation more closely, we find that the computation of Q_{ij} and b_i can be done in parallel, as these two are independent from each other. So are M_{ij} and B_i. Note that as the number of stamps is pre-defined, and the computation is restricted in the sub-stamps, whose sizes are also fixed, the amount of computation in this step is input-size-insensitive. Another implication of this stamp-based parallelization is that the parallelism is limited to the pre-defined number of stamps.

The next step, Checking, takes the least time among all five steps, only 3% of the overall time. As checking involves branching on condition testing, which is unsuitable for the GPU, we will keep the condition testing on the CPU. Nevertheless, the computation is the same as in step 2; therefore, we will parallelize the computation on the GPU similar to that in step 2. As the testing only involves a flag, not actual data, there is little data transfer between the CPU and the GPU.

Step 4, image convolution takes the most time, nearly 8 seconds or 67.5% of the total time. This step is the most computatition intensive and is entirely data-parallel. Therefore we consider parallelizing it on the GPU. Different from the computation in steps 2 and 3, the convolution is done on each pixel, and each sub-image of the size of the convolution *kernel-size* shares the same *kernel* data. Consequently, the thread parallelism in convolution will be one thread for each pixel in the sub-image, and threads for a sub-image shares their convolution *kernel* data. As a result, the convolution time is dependent on the image size, and the number of GPU threads in a thread block depends on the size of the convolution *kernel*.

Finally, the last step, Output takes a slightly longer time than the first step Initialization, and is around 9% of the total time. Since the last step contains little computation and writes results to the output file, it should be completed on the CPU.

B. Detailed Design and Implementation

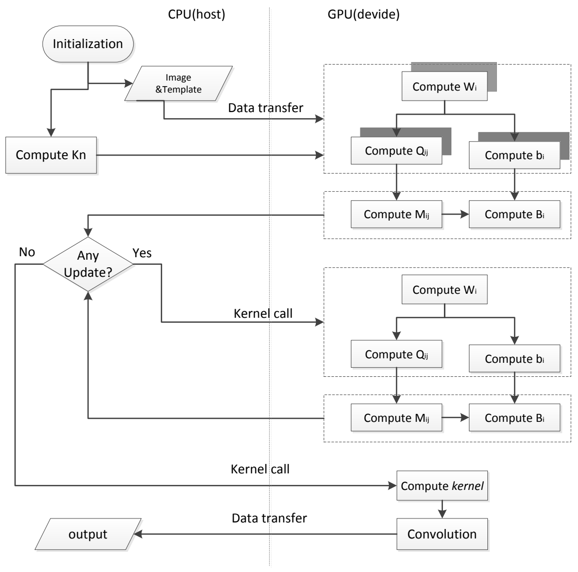

Fig. 1. Running time of original C program

Based on the design ideas, Figure 1 shows the workflow of the CUDA-based HOTPANTS.

We name the kernel programs computing W_i, Q_{ij} and b_i as W, Q and B, and the one for M_{ij} and B_i as M and B2 respectively. The shadowed boxes of GPU kernel program W, Q and B work on all stamps of an image whereas those without shadow only work on one or a few stamps depending on the data. Essentially, the GPU kernel programs M and B2 are the same for steps 2 and 3, whereas the GPU kernel programs W, Q and B may work on less data in step 3 than in step 2 due to the testing condition. Another two GPU kernel program will produce the *convolution kernel* in Eq(1) and the convolution.

In implementing the CUDA-based HOTPANTS, we consider the amount of thread parallelism, the amount of shared resources such as registers and the shared memory. In the following we discuss the implementation issues.

Number of dimensions of the GPU kernel program. In the convolution step, each block of GPU threads is responsible for computing on the pixels of a subimage. The default size of a sub-image in HOTPANTS is 21 x 21. A natural mapping is to set the GPU threads as a two-dimensional block of 21 x 21. Since the GPU threads are scheduled in the unit of warp, or 32 threads, there will be a waste of 34% of the threads in a warp if we are using a GPU kernel program with 21 x 21 two-dimensional blocks. In this case, even though the thread block setting is intuitive and simple, the negative impact on the program efficiency can be significant. Therefore, we choose to set the GPU threads in a one-dimensional thread block. The downside is that such one-dimensional thread blocks require extra registers to store the IDs of their corresponding two-dimensional threads. Another issue is that, as the threads in a one-dimensional block work in accordance with the size of a warp, instead of with the size of the width or height of the input, the address of the data being accessed by the threads in the same warp may be non-consecutive. Nevertheless, in our experiments, the performance benefit outweighs the disadvantages, and 1-dimensional thread block setting improves the speed of image convolution by 20%.

Size of data partitioning. The image data are two-dimensional, and we need to partition the data for the GPU threads. For example, in step 3 by the default setting of HOTPANTS, the GPU will have 292 x 292 threads to calculate the matrix, and we need to partition the data for this number of GPU threads. A common solution is to partition the data in accordance with the full number of threads in one thread block, i.e. 32 x 32. However this amount of data items is too big for a block of GPU threads in this step, and the registers and shared memory assigned to each thread is very limited. As such, through experiments, we divide the two-dimensional data into partitions of 32 x 4 data items. This data partitioning size achieves the best performance for three reasons. First, this setting is in accordance with the size of warp, 32, so that thee threads in the same warp are in the same working/idle condition on the partition of data. Second, it facilitates coalesced access of the data items among the threads in a block. Third, this approach wastes no threads as the partition size of the data can be divided exactly by the size of thread block.

Use of the GPU memory hierarchy. The GPU has fast, on-chip shared memory; however, this shared memory is very small, at tens of kilobytes. To use the fast but small shared memory, we load data from the GPU global memory to the shared memory in small chunks. Specifically, in the GPU computation, we split a large loop into small ones each sized for a thread block. These thread-block-sized small loops for loading can make full use of each block of threads. In each small loop, we load data into the shared memory, synchronize the threads in a block, and perform the computation on the data in the shared memory. This thread synchronization inevidently has performance overhead; nevertheless, with properly sized GPU kernel programs, the synchronization overhead will be minimized, and the use of shared memory will improve the performance considerably.

Another issue is the reuse of the GPU global memory. In our experiments we find that the time of device memory allocation is considerable. Therefore, instead of re-allocating memory space, we reuse the same global memory to store various data at different points in time. In addition to saving time, this reuse can also save memory space.

Combining multiple GPU kernel programs. Multiple GPU kernel programs related in the execution logic can be merged into a single one. This merging can potentially improve the program efficiency [Qin 2012]. The reasons for the improvement are the following. First, starting a GPU kernel program takes time, so starting a single GPU kernel program takes less time than starting multiple one by one. Second, in some cases, a merged GPU kernel program may be able to reduce repeated loading of data from the global memory to the shared memory. However, in our case of implement CUDA-based HOTPANTS, the performance advantage of this approach is very limited. There are specific reasons for this phenomenon. First, in our GPU computation kernel programs the number of threads per block vary greatly due to the algorithmic requirement. As such, the combination of these GPU kernel programs makes some threads idle in some portion of the merged program that requires fewer threads than the earlier or later portions. Second, when merging two GPU kernel programs, it usually requires thread synchronization between the two component programs to ensure the correctness of the merged program, and this synchronization will generate inevitable delays. Finally, for safe-keeping, CUDA will terminate a GPU kernel program if the program does not complete after running for 0.5 second. Thus, when the size of input data is large, e.g. a big input image and template for the convolution, the convolution GPU kernel may hang. Therefore, we leave most of the GPU computation kernel programs separate, and only combine a couple that have the same thread block size and can complete without hanging after merging.

As an example of evaluation of the performance impact of some factors, we examine the following optimized convolution GPU kernel algorithm (we call it the top-line).

Memory Hierarchy. The access time to the registers, the shared memory and the global memory are almost no delay, 1 to 2 cycles and 500 cycles delay respectively. In our example, each 21 x 21-thread block performs 21 x 21-pixel subimage convolution and needs the shared memory to store 4 x 21 x 21 pixels of the input images and 21 x 21

Algorithm 2

```
1: x= threadIdx.x % fwKernel;
2: y= threadIdx.x / fwKernel;
3: load image & kernel into shared memory;
4: syncthreads();
5: for each $i \in [0, fwKernel)$ do
6:     for each $j \in [0, fwKernel)$ do
7:         $q+ = image_{x+i,y+j} * kernel_{fwKernel-i,fwKernel-j}$;
8: $output_{x,y} = q$;
```

pixels convolution kernel, totaling 10.4KB. Because the input image is accessed consecutively, coalesced access to the shared memory is achieved. Without shared memory optimization, Algorithm 3 is the code using the global memory only.

Algorithm 3

```
1: for each $i \in [0, fwKernel)$ do
2:     for each $j \in [0, fwKernel)$ do
3:         $output_{threadIdx.x\%fwKernel,threadIdx.x/fwKernel} + =$
4:         $image_{threadIdx.x\%fwKernel+i,threadIdx.x/fwKernel+j} *$
           $kernel_{fwKernel-i,fwKernel-j}$;
```

Computation Order. In matrix multiplication or image convolution, the basic computing structure is iteration. Nested loops frequently occur because of these two-dimensional block-by-block computation. The choice of the inner and outer layers of nested loops can also affect the efficiency of the implementation. Algorithm 4 changes the order of the computation and Algorithm 5 changes the thread sequence for pixels.

Algorithm 4

```
1: x= threadIdx.x % fwKernel;
2: y= threadIdx.x / fwKernel;
3: load image & kernel into shared memory;
4: syncthreads();
5: for each $j \in [0, fwKernel)$ do
6:     for each $i \in [0, fwKernel)$ do
7:         $q+ = image_{x+i,y+j} * kernel_{fwKernel-i,fwKernel-j}$;
8: $output_{x,y} = q$;
```

Algorithm 5

```
1: x= threadIdx.x / fwKernel;
2: y= threadIdx.x % fwKernel;
3: load image & kernel into shared memory;
4: syncthreads();
5: for each $i \in [0, fwKernel)$ do
6:     for each $j \in [0, fwKernel)$ do
7:         $q+ = image_{x+i,y+j} * kernel_{fwKernel-i,fwKernel-j}$;
8: $output_{x,y} = q$;
```

In Figure 2, we compare the optimized algorithm with the three less optimized versions of the convolution program. The figure shows that the best performance is achieved in the optimized version with suitable use of the shared memory, and right choice of inner loop versus outer loop as well as thread coordinate. With any of the three factors missing, the performance degrades significantly, in the range of 30% to 40%.

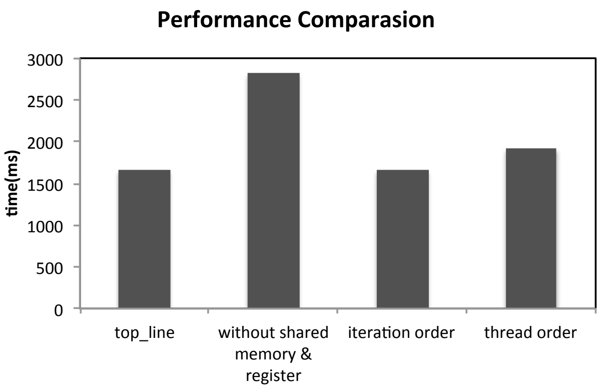

Fig. 2. Performance factors in convolution

C. Co-processing on the GPU and the multicore CPU

We study the co-processing of HOTPANTS on the GPU and on the multicore CPU, as current off-the-shelf computers are commonly equipped with multicore CPUs.

First, GPUs are inherently suitable for image processing, with a high degree of thread parallelism, 512 or more in general at the moment. In comparison, the current multicore CPUs typically only have less than a dozen of concurrent hardware threads. Therefore, the GPU will be more powerful than the multicore CPU on large-scale image data processing, such as the image convolution in HOTPANTS.

However, multi-core CPUs still have an irreplaceable position due to its functional versatility. First, the limited degree of parallelism of the CPU is a good match to a program with such inherent degree of parallelism program; in contrast, such a program cannot fully utilize the threads and resources on the GPU. Second, due to the warp scheduling of the GPU, parallel programs with a large amount of branching do not perform well on the GPU whereas they may perform much better on the multicore CPU. Third, the GPU exposes both the threading configuration and multi-level memory hierarchy to the programmers such that fully exploiting the thread parallelism and memory features requires expertise, advanced design, and tuning. In contrast, the multicore CPU together with OpenMP is a much simpler parallelization framework. An additional issue is that the present GPUs are suitable for addition and multiplication of single-precision data, yet not well-optimized for the double-precision data. As in HOTPANTS there is computation of both types of data, the amount of acceleration is negatively impacted by this factor on the GPU.

In our work, we use both OpenMP on the CPU and CUDA on the GPU to parallelize HOTPANTS. For example, in the checking step, the CPU and the GPU are responsible for different jobs in the co-processing. First, the CPU checks all stamps if their selected sub-stamp are an outlier in the distribution. If so, the CPU calls the GPU kernel programs W, Q and B with the identifier and other information of the stamp. As these calls are asynchronous, a synchronization between the CPU and the GPU will happen when the GPU completes the computation. The CPU then checks if some update has happened. If so, the CPU calls the GPU kernel programs M and B2 to be executed. The information transferred between the GPU and the CPU is very little, essentially just the locations of the neighbouring sub-stamps, thus very little latency.

Since the convolution step is the most time-consuming and is highly parallelizable, we partition the work between the GPU and the CPU. This data-partitioning approach takes advantage of the two different types of processors. In particular, as the image may not be divided exactly by the size of the convolution kernel, some GPU threads are idle when convolving on the boundaries of the image, whereas the multicore CPU is more robust to this problem. Also, as the CPU and the GPU are performing the work independently, there is little data transfer between them. To divide the work in accordance to the speeds of both processors on convolution, we adopt a simple profiling approach: we perform the convolution work entirely on the multicore CPU, and entirely on the GPU, and calculate the work split ratio based on their speeds. This simple approach works sufficiently well in our experiments.

IV. Evaluation

We evaluate the performance of P-HOTPANTS in comparison with CUDA-HOTPANTS (GPU-only), OpenMP-HOTPANTS (multi-threaded CPU-only), and the original HOTPANTS.

A. Experimental Setup

We conduct the experiments on a server with an Intel i7-2600K 3.4GHz CPU and an NVIDIA GTX580 GPU. The CPU has 4 cores and 8 threads. The GTX 580 has 512 CUDA cores and is at 2.0 compute capability. The main memory is 16GB, whereas the GPU device memory is 3GB. The bandwidth between the host (the CPU) and the device (the GPU) is 5GB per second.

The operating system is Scientific Linux 6.2, with gcc version 4.4.5 and nvcc version V0.2.1221.

We use HOTPANTS v 5.1.10, which is the latest version available online. We use the default setting of HOTPANTS, including (1) using the Gaussian basis function; (2) selecting three sub-stamps in each stamp; (3) using the kernel sum as the quality metric to sigma clip outliers; (4) generating the convolution kernels for both the input image and template and then selecting the better one from the two to use.

We also use the following pre-defined parameters with their default values in HOTPANTS, as listed in Table II.

TABLE II
Pre-defined Parameters

Name	Description	Value
fwKSStamp	substamp width in number of pixels	31
fwKernel	kernel width in number of pixels	21
nStamps	number of stamps	10*10
nCompKer	number of dimensions of convolution kernel	49
nbgVectors	degree of the polynomial used to model the differential background variation	3
kerOrder	order of the spatially-varying convolution kernel	2
ncomp2	((kerOrder + 1) * (kerOrder + 2)) / 2	6
nKSStamps	number of substamps in each stamp	3
statSig	threshold for sigma clipping	3

We use three real-world astronomical images at sizes 1601 x 1601, 2048 x 2048, and 3056 x 3056 respectively. These three input images are obtained from real observations. For each input image, a template image of the same size is synthesized from a series of images taken in the same condition. The input images are then registered, including removing interference from cosmic rays, aligning to the template, and stabilizing the offset and rotation based on the template through statistical interpolation.

Both the input images and the reference images are stored as FITS files. The images are single-precision; However, in order to improve accuracy, HOTPANTS uses double-precision convolution *kernel*. We follow HOTPANTS and adopt double-precision convolution kernels in our implementations.

We use $gettimeofday()$ to measure the program execution time on the CPU, and $cudaEventRecord()$ to measure the execution time of a GPU kernel program.

B. Experimental Results

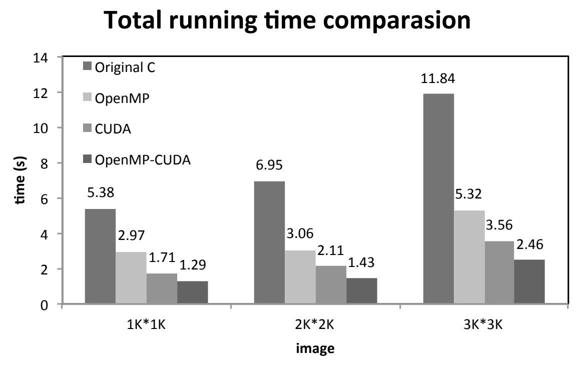

Fig. 3. Comparison of Overall Performance

First, we study the overall performance of P-HOTPANTS in comparison with the original HOTPANTS, CUDA-HOTPANTS, and OpenMP-HOTPANTS. As shown in Figure 3, on each pair of images, P-HOTPANTS achieves the best performance, followed by CUDA-HOTPANTS. Furthermore, with the image size increases, the speedup of P-HOTPANTS over the original increases as well. On the 3K x 3K images, P-HOTPANTS completes the task in 2.5 seconds whereas the original HOTPANTS takes nearly 12 seconds.

Second, we study the time breakdown by the five steps on the pair of 3K x 3K images. As shown in Table III, the speedup of P-HOTPANTS over the original is only 1.5-2.2 on the Initialization and Output steps, as these two steps involve reading or writing files, which are not parallelized. The speedup on the Checking step is also only 2.7 due to the branch divergence in this step. In contrast, the speedups on the Filling and Convolution steps are 7.6-9 times due to their data-parallelism and computation-intensiveness.

TABLE III
TIME ANALYSIS OF OPENMP-CUDA PROGRAM

Step	Init.	Fill	Check	Conv.	Output	Total
Original	0.90	1.52	0.38	7.98	1.05	11.84
(seconds)	7.5%	12.9%	3.2%	67.5%	8.8%	–
OpenMP	0.40	1.33	0.33	2.56	0.70	5.32
(seconds)	7.5%	25.0%	6.2%	48.1%	13.1%	–
Speedup	2.25	1.14	1.15	3.08	1.5	2.22
CUDA	0.90	0.17	0.14	1.36	0.98	3.56
(seconds)	25.5%	4.8%	3.9%	38.2%	27.5%	–
Speedup	1.00	8.94	2.71	5.80	1.07	3.32
P-HOTPANTS	0.41	0.17	0.14	1.04	0.70	2.46
(seconds)	16.7%	7.9%	5.7%)	42.3%	28.4%	–
Speedup	2.20	8.94	2.71	7.58	1.5	3.52

Last, we examine the performance of the Convolution step since it is the most time-consuming step in the overall performance.

TABLE IV
PERFORMANCE OF CONVOLUTION

Image size	1K x 1K	2K x 2K	3K x 3K
Original (seconds)	2.01	3.76	7.77
CUDA (seconds)	0.505	0.821	1.832
OpenMP (seconds)	0.725	1.281	2.832
Estimated GPU-ratio	0.59	0.61	0.61
CUDA-OpenMP (seconds)	0.364	0.610	1.258
Speedup	5.52	6.16	6.18
Best GPU-ratio	0.70	0.75	0.70
CUDA-OpenMP (seconds)	0.292	0.469	0.988
Speedup	6.89	8.02	7.86

Table IV shows that the speedup of P-HOTPANTS over the original HOTPANTS is between 7 to 8, regardless of the size of the input image. The speedup of OpenMP with 8 threads is about 2.7 to 3.0, and that of CUDA between 4.0 and 4.5. Based on the performance ratio of OpenMP and CUDA, we estimate the GPU-ratio as the percentage of work to be allocated on the GPU in P-HOTPANTS: $GPU-ratio = Time_{OpenMP}/(Time_{OpenMP}+Time_{CUDA})$

The estimated ratios are all about 0.6, i.e., we estimate 60% of the work in the Convolution to be done on the GPU and the other 40% of work on the CPU. Interestingly, possibly due to the interference of other tasks on the CPU, the best P-HOTPANTS performance is achieved when the GPU-ratio is 0.70 or 0.75. Nevertheless, the performance difference between these two ratios is insignificant.

V. CONCLUSION

In this paper, we study the problem of accelerating astronmical image subtraction on systems with both multicore CPUs and GPUs. We take the widely used astronomical image subtraction tool HOTPANTS as example and analyze its workflow and algorithmic features. We find that the convolution and parameter filling steps are the most time consuming, taking around 80% of the total time. Moreover, these two steps are computation-intensive and data-parallel. As such, there is great potential to accelerate image subtraction on the multicore CPU and GPU. Therefore, we use the NVIDIA CUDA to parallelize HOTPANTS on the GPU and OpenMP on the CPU. Furthermore, we identify the performance ratio between the multicore CPU and the GPU, and distribute the work of convolution to both based on the performance ratio. Consequently, our P-HOTPANTS, employing both the CPU through OpenMP and the GPU through CUDA, achieves the highest performance among the other versions - the original HOTPANTS, the CUDA-HOTPANTS, and the OpenMP-HOTPANTS. The speedup of P-HOTPANTS over the original HOTPANTS is 4-5 times on real-world astronomical images of various sizes. In particular, on the 3Kx3K images, P-HOTPANTS finishes subtraction in 2.5 seconds whereas the original HOTPANTS needs 12 seconds. Our software will be used in the GWAC (Ground-based Wide-Angle Camera) project in China. We also plan to put it in the public domain for a wider user base.

ACKNOWLEDGMENT

This work was supported by grants 616012 and 617509 from the Hong Kong Research Grants Council and MRA11EG01 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] C. Alard, *Image Subtraction Using A Space-varying Kernel*, Astronomy & Astrophysics Supplement Series, JUNE 2000.
[2] C. Alard, RH. Lupton, *A Method for Optimal Image Subtraction*, The Astrophysical Journal, 1998 August 10.
[3] A. Becker, http://www.astro.washington.edu/users/becker/hotpants.html
[4] GB Berriman, SL Groom, *How will astronomy archives survive the data tsunami?*, Communications of the ACM , Volume 54 Issue 12, December 2011 .
[5] C. J. Fluke, DG. Barnes, BR. Barsdell, AH. Hassan, *Astrophysical Supercomputing with GPUs: Critical Decisions for Early Adopters*, Publications of the Astronomical Society of Australia, 2010.
[6] S. Hartung, H. Shukla, J. Patrick Miller and C. Pennypacker, *Gpu Acceleration Of Image Convolution Using Spatially-Varying Kernel*, IEEE-ICIP, 2012.
[7] A. Heller, *A Wide New Window on the Universe*, S&TR, November 2005.
[8] N. Kaiser, *Pan-STARRS: a wide-field optical survey telescope array*, Proc. SPIE, Ground-based Telescopes, 2004.
[9] V. Podlozhnyuk, *Image Convolution with CUDA*, NVIDIV Corporation.
[10] AK Qin, F Raimondo, F Forbes, YS Ong, *An Improved CUDA-Based Implementation of Differential Evolution on GPU*, ICGEC,2012.
[11] S. Ryoo, CI. Rodrigues, S. Baghsorkhi, SS. Stone, DB. Kirk, WW. Hwu, *Optimization principles and application performance evaluation of a multithreaded GPU using CUDA*, PPoPP, 2008.
[12] S. Philip, B. Summa, V. Pascucci, PT. Bremer, *Hybrid CPU-GPU Solver for Gradient Domain Processing of Massive Images*, Parallel and Distributed Systems (ICPADS), 2011.
[13] Y Yang, P Xiang, M Mantor, HY Zhou, *CPU-assisted GPGPU on fused CPU-GPU architectures*, High Performance Computer Architecture (HPCA), 2012.