Hybrid Multi-Core Algorithms for Regular Image Filtering Applications

Shrenik Lad*, Krishna Kumar Singh*, Kishore Kothapalli and P.J. Narayanan
Email: {shrenik.lad@students.,krishnakumar.singh@students., kkishore@,pjn@}iiit.ac.in
International Institute of Information Technology, Hyderabad, India
Gachibowli, Hyderabad 500 032, India.

***Abstract*—GPGPU has received lot of attention in the past few years mainly because of the performance gain GPUs offer at a low price. Recently, researchers have identified hybrid multi-core computing as a better solution compared to accelerator based computing for several problems. In this paper, we evaluate two regular problems in image processing, bilateral filtering and convolution, on a hybrid multi-core platform. We provide a detailed analysis of these algorithms by comparing their performance on three platforms 1)CPU+GPU hybrid, 2) pure GPU and 3)pure CPU. We show that clear performance gains can be obtained simply by using basic techniques like data decomposition and overlapped execution, when a hybrid computing model is used. Finally, we conclude by discussing some future prospects in the area of hybrid computing.**

I. Introduction

Accelerator based computing has been widely adopted with GPUs and other similar accelerators becoming more and more popular. GPUs are well known for the performance gains they offer at fairly low prices. Other available accelerators are IBM Cell, FPGAs, and ASICs. GPUs are good for data-parallel problems showing regular memory access patterns and high arithmetic intensities. GPU computing in image processing and graphics applications is very common mainly due to the inherent data parallelism present in them. Performance gains as high as 100x-1000x are obtained in such problems. GPUs, however do not perform well in case of irregular computations. In this case, speedup obtained compared to a CPU is much less (around 10x).

It is argued that after applying optimizations appropriate for both GPU and CPU, the performance gap between the two narrows down to 2.5x on average [1]. This implies that good performance gains can be achieved while using GPU and the CPU together for the computation. This leads us to the idea of hybrid multi-core computing, where we distribute our computation among the CPU and the accelerator (GPU in our case), so as to maximise the throughput while utilising all the available resources. However, it is challenging to decide the right workload for the right device.

Bilateral filtering and convolution are common filtering operations in image processing. A bilateral filter is an edge-preserving and noise removing filter, firstly presented by Tomasi and Manduchi in 1998 [2]. In this paper, we present hybrid multi-core algorithms for bilateral filter and convolution and compare the performance with pure-GPU and pure-CPU implementations of the same. To our knowledge, this is the first work on designing hybrid algorithms for these problems.

A. Related Work

A vast amount of literature can be found in the domain of GPGPU, ranging from accelerating scientific applications [4], [5] to essential primitives like Scan, Reduce, and Sort [3]. These works typically have the following structure: The host (CPU) transfers the required data to device (GPU). The kernel execution on the GPU takes place and the output is transferred back to the host. During the entire period, the CPU remains idle and does not contribute to the kernel computation. CPUs are evolving and matching the computational capabilities of GPUs. A hybrid computing model that uses both the CPU and GPU together for computing is a viable alternative.

Some of the earlier works in hybrid multi-core computing include hybrid solutions to solve dense linear algebra problems by Tomov, Dongarra, and Baboulin [6], hybrid approaches for QR factorization [7], Cholesky factorization [8]. Hybrid Computing works show several advantages while using CPU + GPU together, like domain decomposition, pipelined parallelism and more data parallelism. Recent work on hybrid list ranking

* Student Authors

and connected components [9] achieved around 25% speedups compared to their best known GPU implementations. These advantages have motivated us to think in the direction of hybrid computing instead of optimising the pure-GPU approaches.

There are few papers on image filtering using GPUs. Yuko in his paper [10] presents a detailed implementation of the Canny edge-detection algorithm on the GPU. Accelerating such operations makes real time video applications possible. Fialka and Cadik, in their work [11] evaluate two basic approaches (Fourier domain and Spatial domain) for image filtering on the GPUs. They conclude that spatial approach gives better performance than the Fourier approach in many situations.

II. Preliminaries

A. GPU Architecture

The GPU is a massively multi-threaded processor containing hundreds of processing elements or cores, called the Scalar Processors (SPs). The SPs are arranged in groups of eight, called the Streaming Multiprocessors (SMs). The Tesla C1060 has 30 such SMs, which makes for a total of 240 processing cores. These eight SPs execute in Single Instruction Multiple Thread (SIMT) fashion. Hence, all the SPs in an SM execute the same instruction at the same time.

The CUDA API allows a user to create a large number of threads to execute code on the GPU. Threads are also grouped into blocks and blocks make up a grid. Blocks are serially assigned for execution on each SM. For more details, we refer the interested reader to [12]

B. Multi-Core Architecture

We use the Intel i7 980 CPU in our experiments. Each core runs at 3.4 GHz and with a thermal design power of 130 W. The i7-980X has six cores and with active SMT can handle twelve logical threads. The Intel i7 980 CPU has a peak double precision throughput of about 100 GFLOPS.

C. Hybrid Platform

Our hybrid (high-end) platform is a coupling of the two devices, the Intel i7 980 and the Nvidia Tesla C1060 GPU. The CPU and the GPU are connected via a PCI Express version 2.0 link. This link supports a data transfer bandwidth of 8 GB/s between the CPU and the GPU. We also use another low-end hybrid platform consisting of an Intel dual core and Nvidia GT520 GPU. To program the GPU we use the CUDA API Version 3.2 [13]. For programming the CPU, we use OpenMP specification 3.0 and ANSI C.

III. Bilateral Filter

A bilateral filter is an edge-preserving and noise reducing filter used in image processing. It is a non-linear filter in which the intensity value at any pixel is equal to weighted sum of intensities in the neighbourhood. The weight of each neighbour is dependent on its spatial distance as well as the intensity difference with the central pixel. The filter involves transcendental operations like computing exponentials, which can be very computationally expensive. Hence, it is a compute bound problem with regular memory access. Following is the equation for 2D Bilateral Filter.

$$O_{m,n} = \sum_{i=-\frac{k}{2}}^{\frac{k}{2}} \sum_{j=-\frac{k}{2}}^{\frac{k}{2}} I(m',n')\; e^{\frac{-(i^2+j^2)}{2\sigma_d^2}}\; e^{\frac{-(I(m',n')-I(m,n))^2}{2\sigma_r^2}}$$

$$where,\ m' = m - i,\ n' = n - j,\ and\ k * k\ filter$$

Fig. 1. Result of running bilateral filter on image (Taken from [2])

According to [1], bilateral filter computation is able to exploit all the available flops on both GPU and CPU architectures. Existing solutions for bilateral filtering do not exploit the computational power of CPUs and GPUs at the same time. In such a scenario, it makes sense to distribute the computations among the CPU and GPU, proportional to their processing capacity. We would like to do the load balancing in a way that minimises the idle time of both devices and hence maximises the throughput.

A. Proposed Solution

In this section, we describe our hybrid CPU+GPU algorithm for the bilateral filtering problem. The pixels can be divided among threads, where each thread computes the weighted sum for that pixel. The spatial filter can be computed once we know the filter size. For eg: for a 15x15 filter, the spatial x and y distances can vary only between 0 to 7 and hence the spatial filter can

be pre-computed. However, for the intensity (or range) filter, the computations depend on the intensity values of the neighbouring pixel and the central pixel. For a typical Image, it is interesting to note that the intensity difference between any two pixels ||I(i,j) - I(m,n)|| can vary only between 0 and 2^k, where k is the image color depth or bit depth (generally 8 or 16). Hence in our approach, we pre-compute all possible exponentials for intensity differences in (0, 255) and store them in a look-up table during the pre-processing stage. By doing this, we save a lot of computations which are typically done during kernel execution. Following are the equations for computing the *Spatial* and *Range* lookup tables.

$$Range(i) = e^{\frac{-i^2}{2\sigma_r^2}};\ Spatial(i,j) = e^{\frac{-(i^2+j^2)}{2\sigma_d^2}}$$

1) Pre - Processing: Since the computation of exponentials are more expensive on GPU, we create the look-up tables on the CPU. After the pre-processing stage, we divide the image into two parts, I_{GPU} and I_{CPU} and transfer I_{GPU} and the look-up tables to GPU device memory.

X Dist / Y Dist	0	1	2	...	14	15
0	$E_{0,0}$	$E_{0,1}$	$E_{0,2}$		$E_{0,14}$	$E_{0,15}$
1	$E_{1,0}$	$E_{1,1}$	$E_{1,2}$		$E_{1,14}$	$E_{1,15}$
2	$E_{2,0}$	$E_{2,1}$	$E_{2,2}$		$E_{2,14}$	$E_{2,15}$
...						
14	$E_{14,0}$	$E_{14,1}$	$E_{14,2}$		$E_{14,14}$	$E_{14,15}$
15	$E_{15,0}$	$E_{15,1}$	$E_{15,2}$		$E_{15,14}$	$E_{15,15}$

Intensity Diff	Exponential Term
0	I_0
1	I_1
2	I_2
3	I_3
...	...
254	I_{254}
255	I_{255}

Fig. 2. Spatial And Intensity Lookup Table

2) CPU Computation: We use 12 OpenMP threads on an Intel Core i7 for processing the computation on I_{CPU} part of the input image. Each thread takes a set of pixels and computes filtered output for them by using the pre-computed look up tables.

3) GPU Computation: On the GPU, we first load the image from global memory into shared memory. This is necessary because each image pixel will be accessed by multiple threads in its neighbourhood. Around the image block within a thread block, there is an apron of pixels of the width of the kernel radius that is required in order to filter the image block (Refer Fig.3.). Thus, each thread block must load into shared memory the pixels to be filtered and the pixels in the apron. If each thread loads one pixel into shared memory, then the threads which load the apron pixels remain idle during filtering. The number of idle threads can be quite large in case of large filter sizes. In our algorithm, we make each thread in the block load multiple pixels sequentially into shared memory. A linear mapping on thread (tx, ty) gives the region of pixels to be loaded. On an average, each thread loads 4-5 pixels for normal filter sizes. Our approach of loading pixels does not create any idle threads. Once the image is loaded into shared memory, filtering is done using the look-up tables and the filtered image is transferred back to the host.

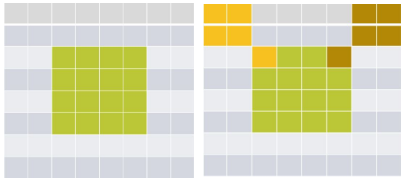

Fig. 3. Linear Mapping for copying pixels from global memory to shared memory. Left portion shows a thread block of 16 threads marked in green color along with the apron in background. Yellow colored thread (right image) in the block copies yellow colored pixels from apron to shared memory. Similarly for others.

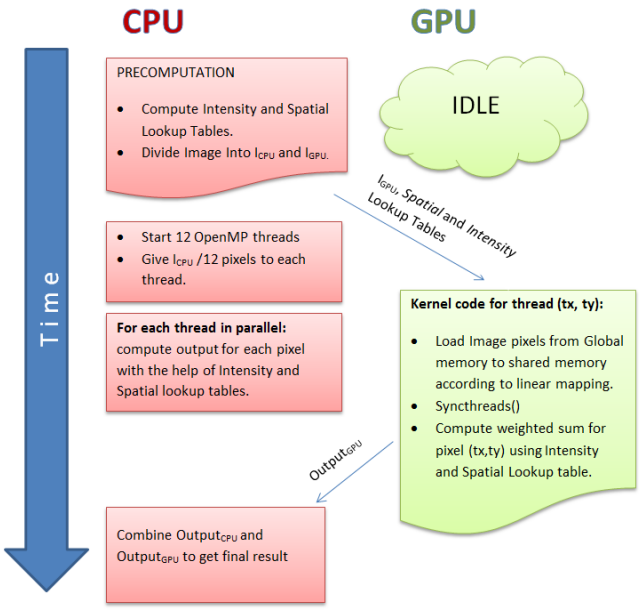

Fig. 4. flow of the hybrid algorithm with time

B. Results

The fastest known implementation of bilateral filter is reported in [1]. When we compare our algorithm's performance with it, we find that our approach is more than two times faster than the best GPU implementation available. This is mainly because of the improvements we have made in the basic implementation. For each

filter size, we vary the I_{GPU} to I_{CPU} ratio (threshold) and report the performance for best threshold. On an average, the threshold for all filter sizes is around 9:1 on the architectures described in Section 2C. This ensures that the idle time for both devices is close to zero.

We move all the work to only GPU and only CPU respectively and call it Pure-GPU and Pure-CPU performance. The average speed-up achieved by our hybrid algorithm over the Pure-GPU code is 10% for filter sizes from 3x3 to 17x17. Figure 6 shows the absolute speed of Hybrid, Pure-GPU, Pure-CPU approaches on both architectures - Hybrid (high end) and Hybrid (low end). Figures 5(left) plots the relative speeds of Pure-GPU and Pure-CPU approaches with respect to the hybrid algorithm's speed on high end architecture. Figure 5(right) plots the same for low-end architecture.

$$Relative\ Speed(GPU) = \frac{Pure\ GPU\ speed}{Hybrid\ Algorithm\ speed}$$

$$Relative\ Speed(CPU) = \frac{Pure\ CPU\ speed}{Hybrid\ Algorithm\ speed}$$

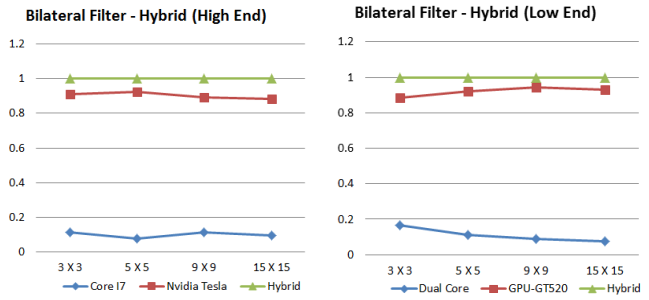

Fig. 5. GPU and CPU relative speed vs filter size on high-end(left) and low-end(right) hybrid platforms

	Core I7	GPU Tesla	Hybrid-High	Dual Core	GPU-GT520	Hybrid-Low
3 X 3	120	955	1050	27	143	161.7
5 X 5	52	619	670	12	100	108.5
9 X 9	18	159	178	4.1	43.9	46.5
15 X 15	6.6	62	70	1.5	18	19.3

Fig. 6. Table showing speed of filtering in Million pixels per second (Mpix/sec) for various filter sizes on high-end and low-end Hybrid platforms

IV. Convolution

Convolution is a common operation used in image processing for effects such as blur, emboss and sharpen. Given the image signal and the filter, the output at each pixel is equal to the weighted sum of its neighbours. Thus, the only arithmetic computations are simple multiply-add operations and its memory accesses are quite regular in a small neighbourhood. Since each pixel can be computed independently by a thread, there is ample parallelism available. The computations increase with the size of filter and it exhibits high compute to memory ratio.

The computation involved is able to exploit all the available flops on both the GPU and the CPU architectures. Since the problem is compute bound and the operations are highly data-parallel, we want the GPU and CPU together to do the filtering on separate image parts. Again, we would like to do the load balancing in a way that maximises performance and minimises the idle time of both devices. Our hybrid algorithm for convolution is similar to bilateral filtering, except that no look-up tables are required here. Only the filter and the image signal are required to compute the filtering result. It must be noted that our work includes only unseparable convolution, and we analyse only small filter sizes - upto 17x17.

There are two approaches of doing convolution - spatial and the Fourier approach, both the approaches having their advantages and disadvantages. [11] does a rigorous evaluation of the two approaches on GPU and identifies that the spatial domain approach outperforms the Fourier approach for simple and small filters. Hence, we chose to use the normal convolution technique on GPU. However on the CPU, we observe that the Fourier approach always outperforms the spatial approach. While using the Fourier approach, the filtering speed does not vary with filter size because convolution translates to a simple multiplication in the Fourier domain. The CPU filtering speed therefore, remains constant across all filter sizes whereas GPU speed decreases. Hence, contribution of CPU to the overall computation increases.

A. Proposed Solution

1) Initialization: Divide the image into two parts I_{GPU} and I_{CPU} and transfer I_{GPU} to the global memory of GPU.

2) GPU computation: Since each pixel is accessed by multiple threads in the same block, we load the image into shared memory by the method described in Section III-A-3. The thread with global id (tx, ty) computes the weighted sum for the pixel (tx, ty) by sequentially going through the neighbours. The filtered image is transferred to the CPU end after completion.

3) CPU computation: On the CPU, we use multiple threads which perform the filtering on I_{CPU}. For very small filters (upto 5x5) we create OpenMP threads which use the spatial approach for filtering, whereas for filters

greater than 5x5, we use MKLs implementation of convolution (Fourier approach).

B. Results

We move all the work to only GPU and only CPU respectively and call it Pure-GPU and Pure-CPU performance. Performance gains upto 15-20% on high-end platform and 30-40% on low-end platforms were achieved. Average speedup considering all filter sizes and both platforms is 18%. The idle time of both devices is close to zero. Figure 7 shows the absolute speed of Hybrid, Pure-GPU, Pure-CPU approaches on both architectures. Figures 8(left) and 8(right) plot the relative speeds of Pure-GPU and Pure-CPU approaches on high-end and low-end platforms respectively.

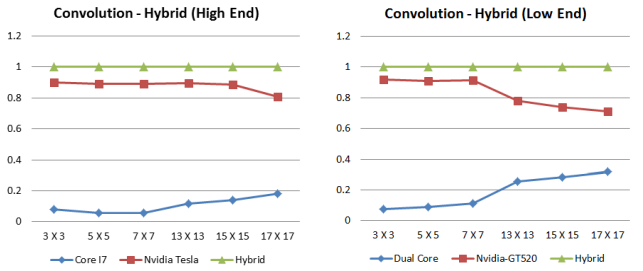

Fig. 7. GPU and CPU relative speed vs filter size on high-end(left) and low-end(right) hybrid platforms

	Core I7	GPU Tesla	Hybrid-High	Dual Core	GPU-GT520	Hybrid-Low
3 X 3	137.8	1553	1728	17.5	216	235
5 X 5	73.13	1103	1234	17.5	177	195
7 X 7	57.6	870	974	17.5	140	153
13 X 13	57.6	441	492	17.5	53	68
15 X 15	57.6	360	405	17.5	46	62
17 X 17	57.6	255	316	17.5	39	55

Fig. 8. Table showing speed of convolution in million pixels per second (Mpix/sec) for various filter sizes on high-end and low-end hybrid platforms

V. Conclusions and Future Work

We have presented hybrid algorithms for two filtering applications used in Image Processing. Good speedups were achieved as compared to the best GPU implementation and our Pure-GPU implementation. Our algorithms are designed to scale as the underlying hybrid architecture changes. They are well suited for current desktops which come with dual-core or quad-core CPU along with a low-end GPU.

Our current work analyses only small filters and does not handle separable convolution. In the future, we plan to do more rigorous study of convolution by designing a hybrid algorithm for all possible scenarios. Presently, GPU libraries are available which perform filtering using only the GPU. A library for filtering algorithms on hybrid architectures will save a programmer from the efforts needed to write a hybrid code. It will also promote hybrid computing since better performance will be available at the same cost.

We also plan to work on other common primitives used in image processing and computer vision. As more hybrid algorithms are being designed and benefits are realised, it becomes imperative for HPC researchers to come up with programming mechanisms that ease their implementation.

References

[1] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen,N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU," in *Proc. ISCA*, 2010.

[2] C. Tomasi and R. Manduchi, "Bilateral Filtering for Gray and Color Images", *Proceedings of the 1998 IEEE International Conference on Computer Vision*, Bombay, India.

[3] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for GPU computing," in *Proc. ACM Symp. GH*, 2007.

[4] N. Govindaraju and D. Manocha, "Cache-efficient Numerical Algorithms using Graphics Hardware," *Parallel Computing*, vol. 33, no. 10-11, pp. 663684, 2007.

[5] R. Cole and U. Vishkin, "Faster Optimal Parallel Prefix Sums and List Ranking," *Info. and Comput.*, vol. 81, no. 3, pp. 334352, 1989.

[6] S. Tomov, J. Dongarra, and M. Baboulin, "Towards Dense Liner Algebrafor Hybrid GPU accelerated manycore systems," *Parallel Computing*, vol. 12, pp. 1016, Dec. 2009.

[7] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov, "QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators," University of Tennessee, Tech. Rep.Computer Science Technical Report, ICL-UT-10-04, 2010.

[8] H. Ltaief, S. Tomov, R. Nath, P. Du, and J. Dongarra, "A Scalable High Performant Cholesky Factorization for Multicore with GPU Accelerators," in *Proc. of VECPAR10*, 2010.

[9] Dip Sankar Banerjee and Kishore Kothapalli, "Hybrid Algorithms for List Ranking and Graph Connected Components," in *Proc. of HiPC*, 2011

[10] Yuko Roodt, Willem Visser and Willem A. Clarke, "Image Processing on the GPU: Implementing the Canny Edge Detection Algorithm," in *Proc. of PRASA*, 2007.

[11] FIALKA O, Cadk M, "FFT and Convolution Performance in Image Filtering on GPU," in *Proceedings of the conference on Information Visualization*, IEEE Computer Society, Washington, DC, USA, 2006, pp. 609-614.

[12] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," *ACM Queue*, vol. 6, no. 2, pp. 4053, 2008.

[13] Nvidia Corporation, "CUDA: Compute Unified Device Architecture programming guide," Technical report, Nvidia, Tech. Rep., 2007.