A Computationally Efficient Parallel Kernel Regression for Image Reconstruction

V.Sairam*, M.Srinivasa Rao*, G.Dada Khalandhar*, L.Srikanth, P.K.Baruah, R.R.Sarma
Sri Sathya Sai Institute of Higher Learning, Prashanthi Nilayam, India.
{v.sairam1,msrini.svn,dadakhalandhar, srikanthcl}@gmail.com, {pkbaruah,rraghunathasarma}@sssihl.edu.in

***Abstract*—Image reconstruction is a method by which the underlying images, hidden in blurry and noisy data, can be retrieved. This is used in applications such as computer tomography (CT), magnetic resonance and radio astronomy. In recent times, a non-parametric adaptive regression method called steering kernel regression was proposed and proved to be effective. This method involves computation of local gradient at each pixel thereby making it computationally intensive. The time consuming parts of the steering kernel regression can be optimized by off-loading them onto GPUs and multi-core processors. The parallel implementation of this algorithm improved the performance of image processing applications such as denoising, deblocking and upscaling. It has given an average speedup factor of 6 on multi-core and 21 on a GPU.**

***Index Terms*—Kernel Regression, Local Gradient, Multi-core, GPU.**

I. Introduction

In this aeon of multimedia, images and videos have become part and parcel of our lives. The need of the hour is to design faster and better algorithms for producing better quality images. The process of digital image acquisition is fraught with many problems such as noise and low sampling rate. Aliasing effects can be observed due to low spatial resolution of the digital imaging systems. Improving the density of CCD arrays is one possible solution, but this is costly. As a cost effective alternative software solution, image reconstruction has been used over many years to improve the quality of the images. Regression has proved to be an effective tool for improving the quality of images and is being used widely for image reconstruction[1]. Takeda et al.[2] have used this for image restoration and enhancement and proposed *steering kernel regression* for interpolation, denoising of sampled data .

With the advent of GPUs, the performance of many practical compute intensive applications are scaled to greater speeds. GPUs enable to satiate the hunger of performance of the application in proportional to the amount of the data parallelism involved in it. The algorithm considered here is computationally intensive and in-order to make it faster, we can exploit the inherent data parallelism in the algorithm. Our contribution in this paper is to make the algorithm computationally efficient by implementing the most time consuming steps on Graphical Processing Unit (GPU) and multi-core platforms. The results of both the approaches are compared.

The remainder of this paper is presented as follows. Section II reviews the theory related to data adaptive steering kernel regression. Section III discusses the methodology used to parallelize the algorithm and issues related to performance optimizations. Section IV presents the experimental results. Finally conclusion and future work are presented in Section V.

II. Data Adaptive Kernel Regression

Regression is a process of finding the underlying signal in a given data where the original signal is corrupted by noise. Many image regression techniques like edge-directed interpolation [3], and moving least squares [4] were proposed. Classical parametric regression methods assume that there is a specific model for underlying signal and estimate the parameters of this model. The parametric estimation has been used in major image processing techniques. The model generated out of the estimated parameters is given as the best possible estimate of the underlying signal.

In contrast, non parametric regression methods don't assume any underlying model, but depend on the data itself to arrive at the original signal. Regression function is the implicit model that has to be estimated. Takeda et al. [2] introduced steering kernel regression for image processing and reconstruction and have shown that it outperforms existing regression techniques. The measured data is given by $y_i = z(x_i) + \epsilon_i$, i=1,...,P, where $z(x_i)$ is

*student author

a regression function acting on pixel coordinates, ϵ_is are independent and identically distributed zero mean noise values, P indicates total number of pixels. Assuming that the regression function is smooth to a certain order N, the objective functional estimation of $z(x)$ can be deduced (detailed derivation is given in [2]) by minimization of the following functional:

$$\min_{\{\beta_n\}} \Sigma_{i=1}^{P}[y_i - \beta_0 - \beta_1^T(x_i - x) - \beta_2^T(x_i - x)^2 - \dots \beta_N^T(x_i - x)^N]^2 \frac{1}{h} K_H\left(\frac{x_i - x}{h}\right) \quad (1)$$

where β_i is i^{th} derivative of $z(x)$ and h is global smoothing parameter. $K_H(x_i - x)$ is the kernel weight assigned to the samples and is defined such that the nearby samples are given higher weight than the farther ones. The central notion of the *steering kernel regression* is to estimate the local gradients. The gradient information captures the features of the image. These in turn are used to find the weights to be assigned to the neighboring samples. Pixel near an edge will be influenced by the pixels lying on the same side of the edge. With this intuition in mind, the dominant orientation of the local gradients are measured. The kernel is then effectively steered locally based on this dominant orientation. An overview of the algorithm is given below.

Adaptive Steering Kernel Regression

The Steering Kernel Regression Algorithm[5] consists of the following steps :

1) Given an image, apply classic kernel regression method to estimate the gradient of the image.
2) Apply singular value decomposition on the above obtained local gradients to compute scaling, rotation and elongation parameters. Determine steering matrix using these parameters.
3) Steering Kernel regression is applied on the original image to obtain new image gradients and the reconstructed image.
4) Iteratively repeat steps 2 and 3 till the noise is reduced in the image and a good quality output is obtained.

Number of iterations is an important factor in determining the quality of the resulting image. This is discussed in Section IV.

III. IMPLEMENTATION

This section presents the parallel implementation of the adaptive steering kernel regression method. The serial code was programmed in MATLAB and is available at http://www.soe.ucsc.edu/~htakeda/MATLABApp. The code was profiled using MATLAB Profiling Tool and it was observed that the steps 1 and 3 of steering kernel regression are the most time consuming parts. On careful analysis, we discovered that these steps have a lot of scope for parallelism. The serial implementation of step 3 consists the following ideas:

(a) To all pixels, in each iteration running till the square of the upscale factor, determine the feature matrix.
(b) For each pixel,
    - Obtain the weight matrix using the neighboring samples of covariance matrix.
    - Compute equivalent kernel which involves inverse of a matrix resulting from the product of the feature and weight matrices.
    - Estimate the pixel values and gradient structure values for the output image.

In Step 1, classic regression method computes feature matrix, weight matrix and equivalent kernel. The most time consuming operations involved in this method are to compute local gradient structure values along the axes directions and to estimate the target values.

Steps 1 and 3 of steering kernel regression involve data level parallelism and CUDA is used to parallelize these steps. We used MATLAB provided MEX interface feature to integrate CUDA and Matlab. We shall now discuss the parallel implementation of the step 3 :

- Copy the original image, covariance matrix, feature matrix from the CPU host memory to GPU global memory.
- Kernel is launched with the total number of threads equaling the total number of pixels in the source image.
- A thread is assigned to each pixel and it does the above mentioned steps of determining weight matrix and equivalent kernel.
- It contributes R^2 pixels in the estimated output image, where R is the upscaling factor.

Computing the steering matrix (step 2) is done on multicore for speed up of the whole process.

OPTIMIZATIONS: Since shared memory is on chip, accesses to it are significantly faster than accesses to global memory. Feature matrix is copied from host memory to shared memory of GPU. Only one thread in a block will get the data from global to shared memory. Only feature matrix could fit in the shared memory due to it's space limitations of 48K. For the remaining data, the global memory request for a warp is split into two memory requests, one for each half-warp, which are issued independently. GPU hardware can combine all memory requests to a single memory transaction if the threads in a warp access consecutive memory locations [6]. Our model is designed in such a way that the memory access pattern by threads is coalesced, thereby improving performance and ensuring low latency.

The multi-core code is implemented using the Parallel Computing Toolbox of MATLAB environment. The performance comparison of these two implementations is given in the following section.

IV. RESULTS

A. EXPERIMENTAL SETUP

For all experiments, the parallelized version of the steering kernel regression was run on Tesla T20 based "Fermi"GPU. The serial and multi-core implementations of the same were run in MATLAB R2012a on a node with Intel(R) Xeon(R) 2.13GHz processor and 24 GB RAM. CUDA timers were employed, which have resolution upto milliseconds to time the CUDA kernels and more importantly they are less subject to perturbations due to other events like page faults, interrupt from the disks. Also CudaEventRecord is asynchronous, there is less of Heisenberg effect when timing is short, exactly suitable to GPU-intensive operations as in these applications. For the multi-core code, timers provided by MATLAB were used.

B. EXPERIMENTS

Experiments and the performance results of GPU, multi-core implementations on simulated and real data are presented. These experiments were conducted on diverse applications and attest the claims made in previous sections. In all experiments, we considered regularly sampled data.
An important note to consider is the determination of the maximum speed up achieved by an application. Understanding the type of scaling that is applicable in any application is vital in estimating the speed up. All the applications that are mentioned here exhibit good scaling. An instance of this phenomenon is shown in Figure 1. One can observe that for a fixed problem size, increase in the number of processing elements, the time of execution decreases.

Quality Test: We performed the quality test for the image by visual inspection and RMSE (root mean square error) values obtained from all the implementations. Figure 6 displays the results of the serial code and CUDA when applied on a CT scan image with Gaussian

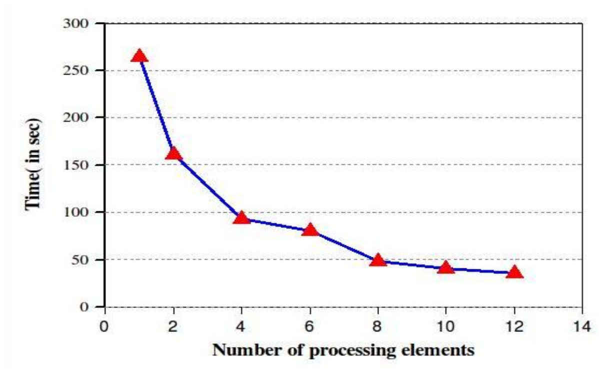

Fig. 1: Plot of execution time against number of processing elements.

noise and standard deviation $\sigma = 35$. The RMSE values of the images resulted from multi-core and GPU implementations are 13.2, which is closer to serial result. We observed that quality was maintained with out any visually plausible differences.

For all the experiments, the initial estimates are given by the classical kernel regression method. Different experiments that were performed are:

1) Denoising Experiment: Well known Lena image of 512×512 size and a picture of a pirate with 1024×1024 size were consideration for testing. Controlled simulated experiment was set up by adding white Gaussian noise with standard deviation of $\sigma = 25$ to both the images. The global smoothing parameter h was set 2.4. The

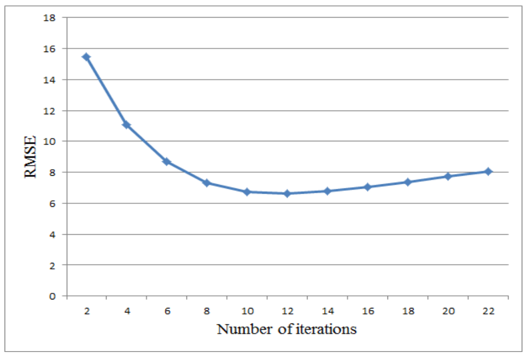

Fig. 2: RMSE(root mean square error) of the resulted image with respect to original image against Number of iterations

choice of number of iterations was set after careful analysis of the behavior of the application. The graph in the Figure 2, indicates that RMSE (root mean square error) values of the resulted image drop till a point and then it raises. This point was observed to be 12 for this application. In this way, a limit for the number of iterations is deduced for all the mentioned images. The RMSE values of the resulted images are 6.6426 and

9.2299.

Clearly, the dominance of the GPU performance over multi-core can be evidently seen in Figure 3. The multi-core code was run with 12 MATLAB workers and as expected a near 10x performance is achieved for image size 1024×1024. The slack is possibly due to the communication overhead of the MATLAB worker threads.

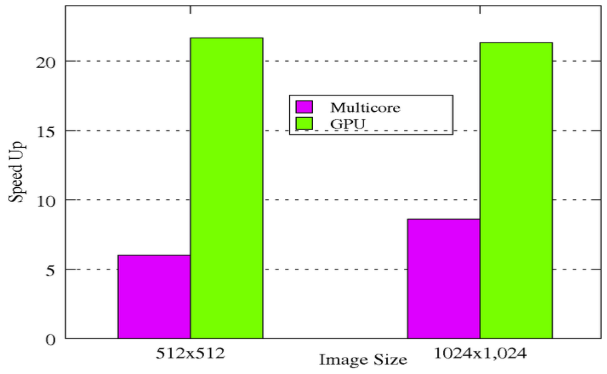

Fig. 3: Speedup factors against image sizes for the denoise application.

Figure 7 shows the results of denoising experiment when applied on a color image of John.F.Kennedy.

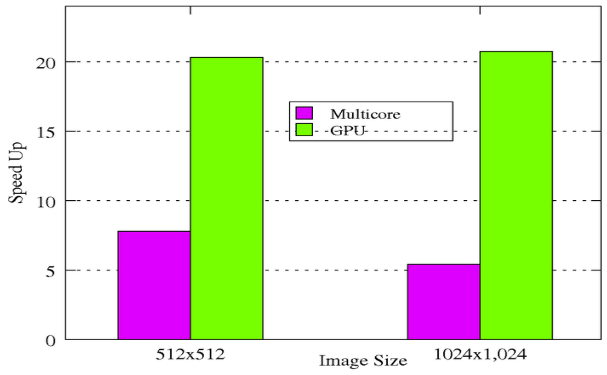

Fig. 4: Speedup factors against image sizes for the compression artifact removal.

2) Compression artifact removal: Pepper image was considered for this experiment and compressed it by MATLAB JPEG routine with a quality parameter 10 and RMSE of 9.76. The number of iterations and h were set to 10 and 2.4. RMSE values for single core version is 8.5765, multi-core version is 8.5759 and that of GPU is 8.5762. The performance is shown in Figure 4. The multi-core code has a average speed up factor of 6 over serial implementation whereas GPU has speed up of 20.

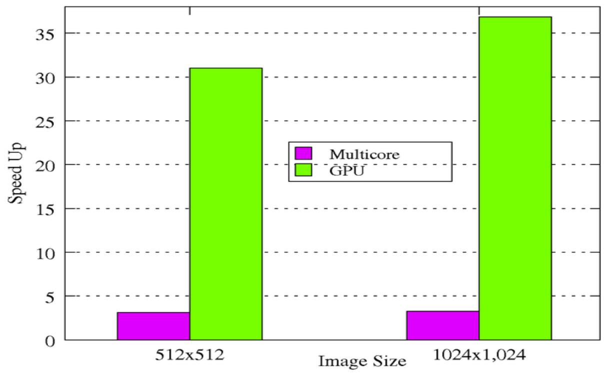

Fig. 5: Speedup factors against image sizes for the upscaling.

3) Upscaling: We performed an upscale operation on the lena image and on the pirate's picture. The performance is noted in the Figure 5. In this case, the GPU performance is achieved to be significantly higher than the multi-core. The arithmetic intensity is high for this application as each thread needs to estimate more number of pixels in the target image, precisely 4 (the upsampling factor).

Kernel Timing Results

Table I presents the runtime measurements of the kernels for single core, multi-core and GPU implementations for different image sizes. The timings given for GPU code also include the data transfer time from host to GPU memory and vice-versa. The GPU code maintains the efficiency even with the increase in the image size. The GPU version of classic kernel has achieved a factor of 175x over the serial implementation. An improvement factor of 75x was observed for steering kernel.

TABLE I: Kernel execution timings(in seconds)

Image size	Regression method	Single core	multi-core	GPU
512 x 512	classic kernel	11.765	1.765	0.067
512 x 512	steering kernel	105.140	18.200	1.470
1024 x 1024	classic kernel	47.741	6.930	0.270
1024 x 1024	steering kernel	427.701	102.234	5.819

V. Conclusions And Future Work

Steering kernel regression is indeed an innovative work in image reconstruction. This algorithm was successfully parallelized on both multi-core, GPU platforms using MATLAB and CUDA. We evaluated our implementations for different applications with varying parameters. From the observations, it is clearly evident

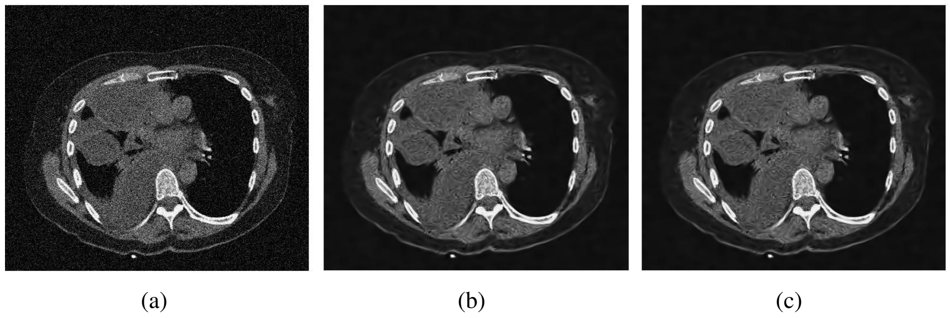

Fig. 6: (a) Noisy image (b) Serial Result (c) Parallel Result.

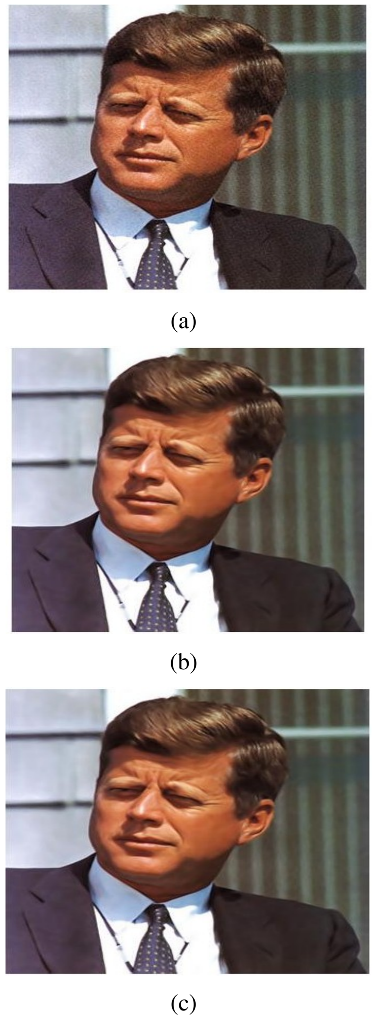

Fig. 7: (a) Noisy Image (b) Serial Result (c) Parallel Result.

that the time involved in computing steering kernel was minimized. On an average, a speedup of 6 and 21 was achieved on the multi-core and GPU.

A direction to the future work is to extend the parallel implementation even to the videos where a plethora of applications like deblurring, super-resolution use this algorithm. All the image datasets considered here fit within the limits of GPU memory. Computing Steering Kernel regression on single GPU may not be scalable to larger images as the on-board memory of GPU is a major constraint. Thus, there is a possibility of using multiple GPUs.

ACKNOWLEDGMENT

We would like to dedicate our efforts to our Divine Founder Chancellor Bhagawan Sri Sathya Sai Baba, without whom this work wouldn't have been possible. We would like to acknowledge Takeda et.al for making the Kernel Regression Tool-box software available on-line. This work was partially supported by nVIDIA, Pune, grant under Professor partnership program and DRDO grant under Extramural Research and Intellectual Property rights.

REFERENCES

[1] M. Unser. Splines: A perfect fit for signal and image processing. *IEEE Signal Processing Magazine*, 16(6):22–38, November 1999. IEEE Signal Processing Society's 2000 magazine award.

[2] Hiroyuki Takeda, Student Member, Sina Farsiu, Peyman Milanfar, and Senior Member. Kernel regression for image processing and reconstruction. *IEEE Transactions on Image Processing*, 16:349–366, 2007.

[3] X. Li and M. T. Orchard. New edge-directed interpolation. *IEEE Trans.*, 10:1521–1527, 2001.

[4] N. K. Bose and N. A. Ahuja. Superresolution and noise filtering using moving least squares. *Trans. Img. Proc.*, 15(8):2239–2248, August 2006.

[5] Peyman Milanfar, Hiroyuki Takeda, and Sina Farslu. Kernel regression for image processing and reconstruction. patent, Aug 30, 2006.

[6] *NVIDIA CUDA Programming Guide*. [Online], Available at:http://developer.download.nvidia.com/, [Accessed Aug 15,2012].