2011 Workshops of International Conference on Advanced Information Networking and Applications

GPU Accelerated Lanczos Algorithm With Applications

Kiran Kumar Matam [1], Kishore Kothapalli [2]

International Institute of Information Technology, Hyderabad
Gachibowli, Hyderabad, India 500 032

[1]kiranm@research.iiit.ac.in, [2]kkishore@iiit.ac.in

***Abstract*— Graphics Processing Units provide a large computational power at a very low price which position them as an ubiquitous accelerator. GPGPU is accelerating general purpose computations using GPU's. GPU's have been used to accelerate many Linear Algebra routines and Numerical Methods. Lanczos is an iterative method well suited for finding the extreme eigenvalues and the corresponding eigenvectors of large sparse symmetric matrices. In this paper, we present an implementation of Lanczos Algorithm on GPU using the CUDA programming model and apply it to two important problems : graph bisection using spectral methods, and image segmentation.**

Our GPU implementation of spectral bisection performs better when compared to both an Intel Math Kernel Library implementation and a Matlab implementation. Our GPU implementation shows a speedup up to 97.3 times over Matlab Implementation and 2.89 times over the Intel Math Kernel Library implementation on a Intel Core i7 920 Processor, which is a quad-core CPU. Similarly, our image segmentation implementation achieves a speed up of 3.27 compared to a multicore CPU based implementation using Intel Math Kernel Library and OpenMP. Through this work, we therefore wish to establish that the GPU may still be a better platform for also highly irregular and computationally intensive applications.

I. Introduction

The computational power of GPUs is increasing rapidly in the last few years. Coupled with the availability of programming environments such as CUDA, GPUs are being used for also general purpose computations. This trend is called GPGPU and presently several fundamental applications are available on GPUs. Examples include sparse matrix vector multiplication [1], SVD [7], and others.

There are many problems which can be modeled as graphs and can be solved by formulating them as discrete combinatorial optimization problems. Some of them include spectral graph partitioning, spectral image segmentation, spectral clustering, spectral graph layout and the like. Most of the aforementioned problems have a lot of practical applications, but posed as discrete optimization problems, these are hard to solve as they generally tend to be NP-complete problems. But real approximations to these problems can be solved using linear algebra methods like finding the spectrum of the Laplacian or adjacency matrix. These solutions require one to compute the extreme eigenvalues and corresponding eigenvectors of the Laplacian matrix of an underlying matrix. This approach of using the extreme eigenvalues, and the corresponding eigenvectors, also finds applications to many other problems from various settings such as computing page ranks [9], and latent semantic indexing. In general, the matrices involved in these computations are large, sparse, symmetric, and are real valued. Lanczos method is well suited for such problems. Lanczos method involves partial triadiagonalization on the given matrix, say A. Important information about the extremal eigenvalues of a matrix tends to emerge long before the tridiagonalization is complete. This makes the Lanczos algorithm particularly useful in cases where a few of the largest or smallest eigenvalues of A are desired. Further, the only large scale-operation involved is sparse matrix-vector multiplication which can be implemented as a black box. In this work, we implement Lanczos algorithm on a GPU and study two applications of the Lanczos algorithm : graph bisection with spectral methods, and image segmentation.

Given an undirected graph G, with vertex set $V(G)$ and edge set $E(G)$, and a positive integer k, the graph partitioning problem is to partition the graph G into k partitions. The partitions have to satisfy two conditions. Firstly, each partition has an equal number of vertices. Secondly, the total number of edges that have end points in two partitions is minimum. Graph partitioning is an important problem that has extensive applications in many areas, including scientific computing, VLSI design, and task scheduling. But graph partitioning is an NP-hard problem and mostly heuristics are employed in practice. Spectral methods are one such heuristics and are known to produce good partitions for a wide variety of graphs. Graph bisection is two-way graph partitioning. Graph bisection can also be posed as a discrete optimization problem which is still intractable. The continuous approximation of this problem can be solved using the eigenvector corresponding to second smallest eigenvalue of the Laplacian matrix of the graph. This eigenvector is referred to as the Fiedler vector. Graph bisection using spectral methods is called *spectral bisection*. Spectral bisection can be recursively called to get a k-way partitioning. Spectral bisection takes into account the global view or the total structure of the graph. Spectral bisection can be used to get an initial partitioning in multi-level schemes where global view of the partition is required [5]. Spectral bisection also is used in graph layout [3], genetic algorithms [8] and the like.

The image segmentation problem is to distinguish objects from background. Image segmentation problem is an important problem in computer vision with a wide range of applications including face recognition, fingerprint recognition, and medical imaging. Given several important applications, the problem hence has natural research interest. There are several heuristics

978-0-7695-4338-3/11 $26.00 © 2011 IEEE
DOI 10.1109/WAINA.2011.70

to solve this problem and one of them is based on the graph partitioning based methods. As shown by Shi et al. [13], one can use the normalized cut criterion. The normalized cut image segmentation method technique introduced by Shi et al. [13] examines the affinities (similarities) between nearby pixels and tries to separate groups that are connected by weak affinities.

In this work, we first provide an efficient and highly optimized implementation on the GPU for the Lanczos method. We then use this implementation for solving two applications: graph bisection, and image segmentation.

Our GPU implementation of spectral bisection performs better when compared to both an Intel Math Kernel Library implementation and a Matlab implementation. Our GPU implementation shows a speedup up to 97.3 times over Matlab Implementation and 2.89 times over the Intel Math Kernel Library implementation on a Intel Core i7 920 Processor, which is a quad-core CPU. Similarly, our image segmentation implementation achieves a speed up of 3.27 compared to a multicore CPU based implementation using Intel Math Kernel Library and OpenMP.

Our image segmentation implementation achieves a speed up of 3.27 compared to a multicore CPU based implementation using Intel Math Kernel Library and OpenMP. Here too, we use standard data sets that are used for image segmentation.

A. Related Work

One can roughly divide the algorithms for eigenvalue problems into two groups. Direct methods which are intended to compute all eigenvalues and (optionally) eigenvectors. Some of the examples of the direct methods are QR Iteration with implicit shifts [4] Jacobi Method [4] etc. Direct methods are typically used on dense matrices and cost $O(n^3)$ operations and require $O(n^2)$ storage space to compute all eigenvalues and eigenvectors and this cost is relatively insensitive to the actual matrix entries. But for large matrices we require far less than $O(n^3)$ operations and $O(n^2)$ storage space. For this reason the implementations of SVD [7], QR Decomposition [11] are not suitable for our work. Iterative methods, e.g., Lanczos method, are usually applied to large sparse matrices or matrices for which matrix-vector multiplication is only convenient operation to perform. Iterative methods typically provide approximations only to a subset of the eigenvalues and eigenvectors. There are other works of Lanczos implementation on GPU. Cavanagh et al [2] and Zheng et al [18]implemented the Lanczos algorithm on dense matrices. They also used single precision floating point operations as the older GPU's did not have support for double precision floating point operations.

As graph partitioning is known to be an NP-Complete problem, it is not possible to compute optimal partitioning for graphs of interesting size in reasonable time. This fact, combined with the importance of the problem, has led to the development of numerous heuristic approaches. These can be classified as either geometric techniques, combinatorial techniques, spectral techniques, combinatorial optimization techniques, or multilevel methods. Spectral graph partitioning technique was introduced by Fiedler in the 1970s, but popularized in 1990 by Pothen et al [15]. These spectral graph partitioning techniques involve finding the partitioning of the graph based on the extreme eigenvectors of the Laplacian matrix of the graph. Second eigenvector corresponding to the second smallest eigenvalue of the Laplacian matrix can be used to partition the graph into 2 parts and this can be recursively applied to obtain a k-way partition of the graph.

Graph-based methods for image segmentation generally model the image as a graph assigning each pixel or a group of pixels to a vertex in the graph and edge weights define the similarity between the vertices. Then based on a criterion designed to model good clusters the graph is partitioned into sets of vertices. Each partition corresponds to an object segment in the image. Vibhav et al. [17] implemented image segmentation using graph-cuts on GPU which involves finding the mincut in the graph. There are no published results for the image segmentation using normalized cut on the GPU.

B. Organization of the Paper

The rest of the paper is organized as follows. In Section II, we briefly describe the architectural and the programming model of the NVidia GPUs. Section III describe the algorithms that are used in the paper. Section IV describes some implementation details and application to the graph bisection problem. Section V describes the application to image segmentation. The paper ends with some concluding remarks in Section VI.

II. GPU Introduction

Nvidia's unified architecture for its current line of GPUs supports both graphics and general computing. In general purpose computing, the GPU is viewed as a massively multithreaded architecture containing hundreds of processing elements(cores). Each core comes with a four stage pipeline. Eight cores, also known as Symmetric Processors (SPs) are grouped in an SIMD fashion into a Symmetric Multiprocessor (SM), so that each core in an SM executes the same instruction. The GTX280 has 30 such SMs, which makes for a total of 240 processing cores. Each core can store a number of thread contexts. Data fetch latencies are tolerated by switching between threads. Nvidia features a zero-overhead scheduling system by quick switching of thread contexts in the hardware. The CUDA API allows a user to create 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. The blocks themselves are divided into SIMD groups called warps, each containing 32 threads on current hardware. An SM executes one warp at a time. CUDA has a zero overhead scheduling which enables warps that are stalled on a memory fetch to be swapped for another warp.

The GPU also has various memory types at each level. A set of 32-bit registers is evenly divided among the threads in each SM. 16 Kilobyte of shared memory per SM acts as a user-managed cache and is available for all the threads in a Block. The GTX 280 is equipped with 1 GB of off-chip global memory which can be accessed by all the threads in the grid, but may incur hundreds of cycles of latency for each fetch/store. Global memory can also be accessed through two

read-only caches known as the constant memory and texture memory for efficient access for each thread of a warp.

Computations that are to be performed on the GPU are specified in the code as explicit kernels. Prior to launching a kernel, all the data required for the computation must be transferred from the host (CPU) memory to the GPU global memory. A kernel invocation will hand over the control to the GPU, and the specified GPU code will be executed on this data. Barrier synchronization for all the threads in a block can be defined by the user in the kernel code. Apart from this, all the threads launched in a grid are independent and their execution or ordering cannot be controlled by the user. Global synchronization of all threads can only be performed across separate kernel launches. For more details, we refer the interested reader to [14].

III. The Algorithms

A. The Lanczos Algorithm

Let A be an $n \times n$ real symmetric matrix and x be a nonzero vector. The Rayleigh quotient, denoted $\rho(x; A)$, is $\rho(x; A) := \frac{x^T Ax}{x^T x}$. The minimum and maximum values of the Rayleigh quotient on A equals to λ_1 and λ_n, where λ_1 ... λ_n are eigenvalues of A and $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_N$. Any subspace iterative method that finds the eigenvalue approximations of A builds the subspace in the direction of the gradient of Rayleigh quotient, so that the eigenvalues of the projection of A on the subspace are increasingly better approximates of the eigenvalues of A. The subspaces formed by the Krylov sequence$(q_1, Aq_1, A^2q_1, ..., A^{n-1}q_1)$ are in the direction of the gradient of the Rayleigh quotient. By the properties of the Krylov subspace$(K(A, q_1, n) = [q_1, Aq_1, A^2q_1, ..., A^{n-1}q_1])$ the projection of A on the orthonormal basis of Krylov subspace is a symmetric tridiagonal matrix. If $Q^TAQ = T$ is tridiagonal with $Qe_1 = q_1$, then $K(A, q_1, n) = Q[e_1, Te_1, T^2e_1, ..., T^{n-1}e_1]$ is the QR factorization of $K(A, q_1, n)$ where $e_1 = I_n(:, 1)$. Thus q_k can effectively be generated by tridiagonalizing A with an orthogonal matrix whose first column is q_1. So by setting $Q = [q_1, q_2, ..., q_n]$ and T as $n \times n$ symmetric tridiagonal matrix with diagonal elements as $\alpha_1, \alpha_2, ..., \alpha_n$ and subdiagonal elements as $\beta_1, \beta_2, ..., \beta_{n-1}$ and equating columns in $AQ = QT$, we find:

$$\begin{aligned} Aq_k &= \beta_{k-1}q_{k-1} + \alpha_k q_k + \beta_k q_{k+1}, \text{ for } k = 1 \text{ to } n-1. \\ \beta_0 q_0 &\equiv 0 \end{aligned}$$

Pseudo-code for the Lanczos algorithm is shown in Algorithm 1. By properly sequencing the above formulae we obtain the Lanczos Iterations shown in Table 1 for over m iterations. We run the Lanczos iterations until we get the desired eigenvalues to a required precision. After the Lanczos iterations we get the Lanczos vectors$(Q_m = [q_1, q_2, ..., q_m])$. The eigenvalues of the T are the approximate eigenvalues of A. We solve for the eigenvalues of T by bisection method and sturm sequence property and then get the corresponding eigenvector of T by inverse iteration method. Then we get the required eigenvectors of A by multiplying the Lanczos vectors with the corresponding eigenvectors of T.

Algorithm 1 Lanczos Algorithm

$q_1 \leftarrow$ Random vector with norm 1
$q_0 \leftarrow 0$
$\beta_1 = 0$
for $j = 1, 2, ..., m$ **do**
  $w_j \leftarrow Aq_j - \beta_j q_{j-1}$
  $\alpha_j \leftarrow w_j \cdot q_j$
  $w_j \leftarrow w_j - \alpha_j q_j$
  $\beta_{j+1} \leftarrow \|w_j\|$
  $q_{j+1} \leftarrow w_j/\beta_{j+1}$
  Compute eigenvalues, eigenvectors and error bounds of T_j
end for

B. Algorithm for Graph Bisection

Let $G = (V, E)$ be the graph for which we need to find the bisection. The Laplacian matrix of a graph G, denoted $L(G)$, that is obtained as follows. Let $n = |V(G)|$, $L(G)$ is an $n \times n$ matrix where $L(G)[i, j]$ is the degree of the vertex i if $i = j$ and $L(G)[i, j] = -1$ if $i, j \in E(G)$ and $L(G)[i, j] = 0$ otherwise. Let s be the index vector of size $|V|$ with entries as $s[i] = -1$ if vertex i is in first partition(V-) and $s[i] = 1$ if vertex i is in second partition(V+), then $e^T S = 0$, where $e = [1, 1, ..1]^T$ as the partition needs to be balanced. The number of edges with end points in different partitions (edgecut,R) is equal to $R = \frac{s^T L(G) s}{4}$. Minimizing R is a discrete optimization problem and is NP-hard. But by taking the continuous approximation to x (values are real) and $\|s\|_2^2 = |V|$, R becomes $R = \|V(G))\| \cdot \frac{s^T L(G) s}{s^T s}$. Minimizing R such that s is not 0 and $e^T s = 0$ is a special case of the Courant Fischer Minimax Theorem. The solution is the eigenvector corresponding to the second smallest eigenvalue of $L(G)$. The continuous approximation solution is then mapped to the discrete optimization problem. We find the median of the second eigenvector and assign -1 to values that are less than or equal to median and 1 to values that are greater than median. The obtained partition may not be optimal but gives a good edgecut quality. Table 2 shows the spectral bisection algorithm.

Algorithm 2 Spectral Bisection Algorithm

Compute the eigenvector V_2 corresponding to λ_2 of L(G)
for each node n of G **do**
  if $V_2(n) \leq Median(V_2)$ **then**
    put node n in partition $V-$
  else
    put node n in partition $V+$
  end if
end for

C. Image Segmentation via Spectral Methods

Image segmentation is the task of finding groups of pixels that share some visual characteristics. Shi et al. [13] proposed a new graph theoretic criterion called normalized-cut for image

segmentation. Let $G(V, E)$ be the graph and A, B be the two disjoint partitions of G. Then normalized-cut is defined as

$$Ncut(A, B) = \frac{Cut(A,B)}{Assoc(A,V)} + \frac{Cut(A,B)}{Assoc(B,V)}$$

where $Cut(A, B) = \sum_{u \in A, v \in B} w(u, v)$ and $Assoc(A, V) = \sum_{u \in A, t \in V} w(u, t)$.The normalized cut criterion measures both the total dissimilarity between the different groups as well as the total similarity within the groups. Minimizing the normalized cut gives a good segmentation of the image. Minimizing normalized cut is a discrete combinatorial problem which is NP-complete and the continuous approximation to it is a generalized eigenvalue problem and can be solved. Let W be the adjacency matrix of the graph $G = (V, E)$ constructed by taking each pixel as node and the edge weight w_{ij} between the node i and j is the product of a feature similarity term and a spatial proximity term:

$$w_{ij} = e^{\frac{-\|F(i)-F(j)\|_2^2}{\sigma_I^2}} \cdot \begin{cases} e^{\frac{-\|X(i)-X(j)\|_2^2}{\sigma_X^2}} & \text{if } \|X(i) - X(j)\|_2 < r \\ 0 & otherwise \end{cases}$$

where $X(i)$ is the spatial location of node i, i.e., the coordinates in the original image I, and $F(i)$ is a feature vector defined as $F(i) = I(i)$, the intensity value, for segmenting brightness images. Algorithm 3 shows the normalized cut image segmentation algorithm.

Algorithm 3 Normalized Cut Algorithm

1. Given a set of features, set up a weighted graph $G = (V, E)$ compute the weight on each edge, and summarize the information into W and D.
2. Solve $(D - W)x = \lambda Dx$ for eigenvectors with the smallest eigenvalues.
3. Use the eigenvector with the second smallest eigenvalue to bipartition the graph by finding the splitting point such that $Ncut$ is minimized.
4. Decide if the current partition should be subdivided by checking the stability of the cut, and make sure $Ncut$ is below the specified value.
5. Recursively repartition the segmented parts if necessary.

IV. Application I : Graph Bisection

In this section, we describe some implementation details when using the Lanczos method for graph bisection.

A. Implementation Details

We directly take the Laplacian Matrix of the Graph as input and store it as Compressed Row Storage (CRS) sparse matrix representation as the given graphs are sparse. We implement the dot product, norm using CUBLAS routines. We implemented the sparse matrix vector multiplication with the matrix stored in CRS format as described by Bell et al. [1]. This format works well also as the underlying graphs are sparse and do not lend efficiently to other sparse matrix representations. We introduced a few changes to the method described in [1] given that our matrices are symmetric and integer valued.

Since our matrices are integer valued, all the entries in the Laplacian matrix are integer valued. The Lanczos vectors may however contain double precision floating point numbers. Notice that the GPU is not a very versatile architecture. On the GPU, it is efficient to multiplying an integer with a double precision floating point number compared to multiplying two double precision floating point numbers. Fortunately, in the Lanczos algorithm, we require operations of the first kind only. So, we convert the spmv routine [1] correspondingly.

We also dynamically check the correct version, between CSR (scalar) and CSR (vector), the spmv routines described by Bell et al [1] to use depending on the density of a row. If the average number of entries per row is less than 7 we used the CSR (scalar) spmv routine where one thread is assigned to each row. Otherwise we used CSR (vector) where one warp is assigned to each row. Experimentally, it was observed that when a row has less than 7 entries, then it is efficient to use the CSR (scalar) routine.

We implemented finding the median of a vector using primitives reduce, scan from CUDPP. When implementing the Lanczos Algorithm using the single precision floating point operations, it is likely that Lanczos vectors may suffer loss of orthogonality. This can give incorrect results when using the Lanczos algorithm for graph bisection. In this case, it is recommended that double precision floating point operations be used for correct results. However, the GPU is not a very versatile architecture and cannot support double precision operations efficiently. This results in a significant reduction in the performance.

We need to store the Lanczos vectors obtained during iterations to get the second eigenvector. We see that for large graphs (from road networks [6]) there is not enough space to store the Lanczos vectors. For such large instances, we recommend that other advanced Lanczos implementations like implicitly restarted Lanczos method be used. This however may increase the overall time taken while being also quite involved to implement efficiently. The basic idea is to restart the Lanczos method after every few iterations with a better initial estimate for the vector q_1 from Algorithm 1.

B. Experimental Results

In this section, we report the results of our experiments. The experiments were run on the following systems:

- **CPU:** An Intel Core i7 920, with 8 MB cache, 4 GB RAM and a 4.8 GT/s Quick path interface, with maximum memory bandwidth of 25 GB/s.
- **GPU:** A Tesla C1060 which is one quarter of a Tesla S1070 computing system with 4 GB memory and 102 GB/s memory bandwidth. It is attached to a Intel Core i7 CPU, running CUDA Toolkit/SDK version 2.2. [14].

Our results on the Matlab are performed an Matlab installation on the CPU mentioned above.

For the experiments on the graph bisection using spectral methods we took the graphs from the Walshaw benchmark which is a popular graph partitioning archive. Figure 1 shows the graphs instances along with their sizes and the number of edges between the partitions. The reported edge cut values

Instance#	Instance Name	$\|V\|$	$\|E\|$	Edge cut
1	add20.graph	2395	7462	871
2	uk.graph	4824	6837	37
3	crack.graph	10240	30380	231
4	fe_sphere.graph	16386	49152	462
5	bcsstk30.graph	28924	1007284	6536
6	wing.graph	62032	121544	1349
7	finan512.graph	74752	261120	978
8	fe_rotor.graph	99617	662431	3166
9	144.graph	144649	1074393	7236
10	m14b.graph	214765	1679018	4057
11	auto.graph	448695	3314611	12831

Fig. 1. Table showing a few instances along with their size and the number of edges between the partitions.

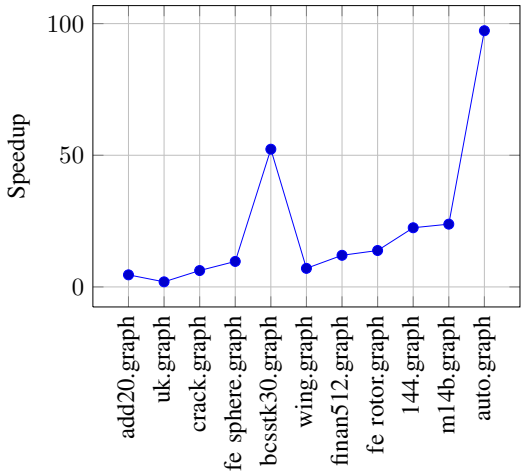

Fig. IV-A Comparison with Matlab

are the average obtained over multiple runs. Figure 1 lists only a representative set of instances with varying sizes from 2K to 500K. Results on the entire data set from Walshaw's benchmark are available in [12]. Figure IV-B shows the overall speedup when compared to Matlab implementation and Figure IV-B shows the speedup when compared to an Intel MKL implementation.

Observations: Most of the graph partitioning time is taken by the Lanczos method for finding the second eigenvector of the Laplacian matrix. For small graphs we observe that the speedup when compared to multi-core implementation is less than 1 because as there is not enough work to keep all the cores in the GPU busy. For the large graph finan512.graph we see that the time taken by GPU implementation is more when compared to multi-core implementation because the sparse matrix vector multiplication(spmv) routine in Lanczos iteration takes more time on GPU than on multi-core for that graph. In GPU access to global memory is slower and coalesced reads to global memory are much faster than the random ones. In the spmv routine the global memory access depends on the column indices of the elements in the row, and hence the runtime of the spmv routine depends on the graph. Further, load imbalance induced by the fact that the number of non-zero elements vary across rows causes performance degradation.

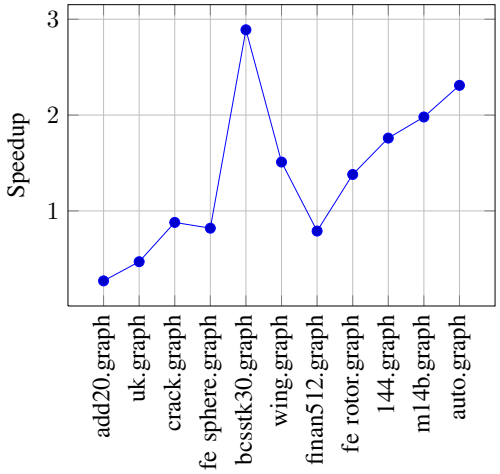

Fig. IV-B Comparison with Intel MKL

As multi-core CPU's are good at irregular memory reads the speedup of GPU implementation when compared to multi-core implementation depends on the spmv routine. SPMV routine takes most of the time in Lanczos iterations and for the graph finan512.graph, spmv routine takes more time on GPU when compared to multi-core implementation. Therefore the speedup in this case is less than 1. As the size of the graphs increases there are variations in the speedup because Lanczos takes more iterations to converge for the graphs. The speed of convergence depends on the eigengap between λ_2 and λ_1 of the Laplacian matrix, i.e on λ_2 - λ_1.

V. Application II : Image Segmentation

A. Implementation Details

The algorithm that we followed in our implementation is shown in Algorithm 3. Each step in Algorithm 3 is mapped to different kernels on GPU. Where ever possible, we make use of CUDPP and CUBLAS primitives. We first construct the similarity graph on GPU and store it in CRS format. We compute the Laplacian matrix from the similarity graph and then find the eigenvector e corresponding to the second smallest eigenvalue. We then split the vertices into partitions based on the eigenvector e. We take ℓ evenly spaced values, $e[i \cdot n/\ell]$ for $i = 1, 2, \cdots, \ell$, in the eigenvector e, and consider ℓ partitions as follows. The partition P_i is obtained by partitioning the vertex set based on $e[in/\ell]$. For each of the partitions P_i, where $i = 1, 2, \cdots, \ell$, we calculate the normalized cut value n_i. Finally, the vertex set is partitioned according to $e[i^* n/\ell]$ where $i^* = \text{argmin}_i n_i$. The process is terminated if $e[i^* n/\ell]$ is less than a threshold value. Otherwise, we continue in a recursive fashion.

For all the kernels we kept the blocksize as 512 for the better occupancy of the SM. For constructing the graph we assigned each pixel to a thread and launched threads equals to the number of pixels. Each thread calculates its neighbours among the possible r^2 neighbours. We then perform a scan and kernel launch to construct the similarity matrix stored in CRS format. Then we recursively partition the graphs

Fig. 2. Shows the images of Sponge Person and Flower.

Fig. 3. Shows the segmented images of the Sponge, Person and Flower with ncut values less than 0.0075,0.0065 and 0.0275 correspondingly. Parameter settings are $\sigma_X = 0.05$, $\sigma_I = 4, r = 5$

into subgraphs. From the similarity graph we construct the required Laplacian matrix on the GPU. We then calculate the eigenvector corresponding to the second eigenvalue using our Lanczos method implemented on GPU. We then select l equally spaced splitting points and for each splitting point we calculate the normalized cut for the partitions produced by the splitting point. For calculating the normalized cut we first calculate the edge cut and then the association of each partition i.e the sum of all edge weights of vertices in that partition. We see that the number of neighbours of each pixel is greater than 32. We therefore assign one warp to each row(or vertex) to read the column indices and the values, so that we can make the memory reads of the threads coalesced and gain better performance. After calculating the edge cut of each pixel we then calculate the total edge cut using the cublas sum function. If the minimum normalized cut is less than a threshold we stop the recursion. Otherwise we segment the partition into subpartitions and construct the subgraphs for that partitions. We assign each row to a thread and it calculates the vertices in its partition and then get the rowindices array of the subgraph. Then we again launch a kernel to get the column indices and values array of the subgraph.

B. Experimental Results

We present experiments on the figures sponge, flower, and person. We compare our GPU implementation of image segmentation with the multi-core implementation. We implemented the image segmentation on multi-core using Intel MKL libraries and Openmp. The sizes of the images shown in 2 Flower, Person, and Sponge respectively are $100 \times 80, 480 \times 320$, and 600×450. Figure 3 shows the output of image segmentation on the respective figures. The speedup of our implementation on the GPU can be seen in Figure 8.

We see that most of the image segmentation time is taken for calculating the second eigenvector by Lanczos method. For the sponge image and person image the GPU implementation performs better. For the sponge image and flower image we see that the algorithm is able to identify the major components but for the person image we see some discrepancies.

VI. Conclusions

In this work, we have shown that GPUs can be used to even difficult numerical optimization problems such as finding the extreme eigenvalues of large sparse matrices. We have applied our implementation on two problems of practical interest. In future, we wish to study further applications of the present implementation and also study other numerical optimization problems that can benefit from a GPGPU point of view.

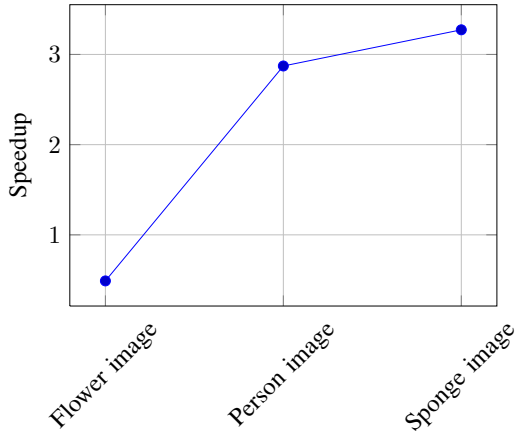

Fig. 8. Comparison with multi-core Implementation

References

[1] N. Bell, M. Garland Efficient Sparse Matrix-Vector Multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, December 2008.

[2] Joseph M. Cavanagh, Thomas E. Potok, and Xiaohui Cui. Parallel latent semantic analysis using a graphics processing unit. In Proc. GECCO 09, pages 2505–2510, 2009.

[3] J. Daz, J. Petit, and M. Serna, A survey of graph layout problems, ACM Computing Surveys Journal, pp. 313356, 2002.

[4] Demmel, J. W., Applied Numerical Linear Algebra. Society for Industrial and Applied Mathematics, 1997.

[5] Hendrickson, B. and Leland, R. 1995. A multilevel algorithm for partitioning graphs. In Proc. SC 1995.

[6] Jure Leskovec, Stanford large network dataset collection, Available at http://snap.stanford.edu/data/#road

[7] Sheetal Lahabar, P. J. Narayanan, "Singular value decomposition on GPU using CUDA," in Proc. IPDPS, pp.1-10, 2009.

[8] Martin, J. G. Spectral techniques for graph bisection in genetic algorithms. In Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation, pp. 1249-1256, 2006.

[9] Andrew Y. Ng, Alice X. Zheng, and Michael I. Jordan. Link analysis, eigenvectors and stability. In Proc. IJCAI, 2001.

[10] Von Luxburg U.: A Tutorial on Spectral Clustering.Tech. Rep. TR-149, Max Plank Institute for Biological Cybernetics, August 2006.

[11] Kerr, A., Campbell, D., and Richards, M. QR decomposition on GPUs. In Proc.c of GPGPU-2, pp. 71-78, 2009.

[12] Kiran Kumar M., Results of spectral graph partitioning using the Walshaw benchmark archive. Available at http://researchweb.iiit.ac.in/∼kiranm/results.xls.

[13] Jianbo Shi; Malik, J.; , "Normalized cuts and image segmentation," IEEE T. PAMI, vol.22, no.8, pp.888-905, 2000.

[14] NVIDIA Corporation. CUDA: Compute unified device architecture programming guide. Technical report, NVIDIA, 2007.

[15] Alex Pothen, Horst D. Simon, Kan-Pu Liou, Partitioning sparse matrices with eigenvectors of graphs, SIAM J. Mat. Anal. Appl., 11(3), pp. 430-452.

[16] D.A. Spielmat, Shang-Hua Teng, "Spectral partitioning works: planar graphs and finite element meshes," focs, in Proc. FOCS, pp. 96, 1996.

[17] Vibhav Vineet, P. J. Narayanan, CUDA cuts: Fast graph cuts on the GPU, In CVPR Workshops, 2008, pp. 1-8.

[18] Jing Zheng; Wenguang Chen; Yurong Chen; Yimin Zhang; Ying Zhao; Weimin Zheng; , "Parallelization of spectral clustering algorithm on multi-core processors and GPGPU," in Proc. ACSAC 2008, pp.1-8.