Proceedings of the IASTED International Conference
Parallel and Distributed Computing and Networks (PDCN 2014)
February 17 - 19, 2014 Innsbruck, Austria

GPU ACCELERATED VESSEL SEGMENTATION USING LAPLACIAN EIGENMAPS

Lin Cheng
Department of Engineering
Trinity College
300 Summit Street
Hartford, CT, United States
email: lin.cheng@trincoll.edu

Hyunsu Cho and Peter A. Yoon
Department of Computer Science
Trinity College
300 Summit Street
Hartford, CT, United States
email: {Hyunsu.Cho, Peter.Yoon}@trincoll.edu

ABSTRACT
Laplacian eigenmap is one of the most widely used techniques to improve cluster-based segmentation of multivariate images. However, one problem with this approach is its excessive computational requirements, especially when processing large image datasets. In this paper, we aim to employ the emerging commodity graphics hardware of eigenmap-based segmentation. In particular, we present a highly parallel implementation for vessel segmentation using Nvidia's CUDA parallel computing platform. We demonstrate that segmentation steps such as computing the weight matrix can be implemented in a highly parallel fashion. In addition, our approach does not require the computation of the entire spectrum of eigenvalues, which is the most time-consuming step in eigenmap-based segmentation. Instead, we use the Lanczos method to calculate the extreme eigenvalues in parallel. Our experiments based on vessel images of various size achieve a speedup up to 14x over the conventional sequential implementations.

KEY WORDS
Image segmentation, Eigenmap, GPU computing.

1 Introduction

Pattern recognition techniques have been reported to segment a multivariate image into meaningful regions [1, 2]. In particular, Laplacian eigenmap is recently reported as a useful technique to improve cluster-based segmentation of multivariate images [3]. In this approach, the local image characteristics can be embedded in a high-dimensional feature space. A low-dimensional map is reconstructed, where the local image variations are presented in the context of global image variation. These non-linear projections serve as inputs to the fuzzy c-means algorithm. The final segmentation is produced by a labeling scheme that works pixel by pixel.

While experimental results using RGB images demonstrated the effectiveness and robustness to noise [3], one problem with these algorithms has been their high computational requirements for large image data. This presents a great challenge to image segmentation problems especially when processing large image datasets. For example, in medical image analysis, detecting and segmenting microvasculature usually utilizes a large set of images obtained from the medical imaging modality.

One method to address this problem is by parallel computing using a Graphics Processing Unit (GPU) [4, 5, 6]. GPU features substantial arithmetic and memory bandwidth capabilities, as well as user program interface for general-purpose computation on graphics hardware. In recent years, a number of non-graphics-oriented computationally expensive problems have been implemented on GPU, rather than on clusters of workstations owing to GPU's growing availability. Intrinsically, eigenmap-based segmentation algorithms features data-level parallelism with large computational requirements. This makes it very suitable to be implemented on the GPU.

This work concerns Laplacian eigenmap-based multivariate image segmentation based on data-parallel GPU programs. In particular, we aim to employ the emerging commodity graphics hardware to solve vessel segmentation problems using high parallel computing power. We implement eigenmap-based segmentation algorithms on Nvidia's Compute Unified Device Architecture (CUDA) platform for segmenting vessel image data sets. As we describe our CUDA implementation, we demonstrate that segmentation steps such as computing the weight matrix are highly parallel. As a means to evaluate the algorithms, we test input images of various size and observe a speedup factor of 14x for large images.

The paper is organized as follows. In Section II we briefly review the related theoretical concepts. In Section III we explain the parallel segmentation on GPU. Results from numerical experiments are presented in Section IV. In the last section, conclusions and further research are discussed.

2 Spectral Clustering and Eigenmap

The starting point of the theoretical algorithms is dimensionality reduction, especially for intrinsically low-dimensional data lying in a high-dimensional space.

Classical dimensionality reduction approach has origins in multidimensional scaling and principal components analysis (PCA) [7, 8]. Most of these methods do not explicitly incorporate the manifold structure the data may possibly

DOI: 10.2316/P.2014.811-012

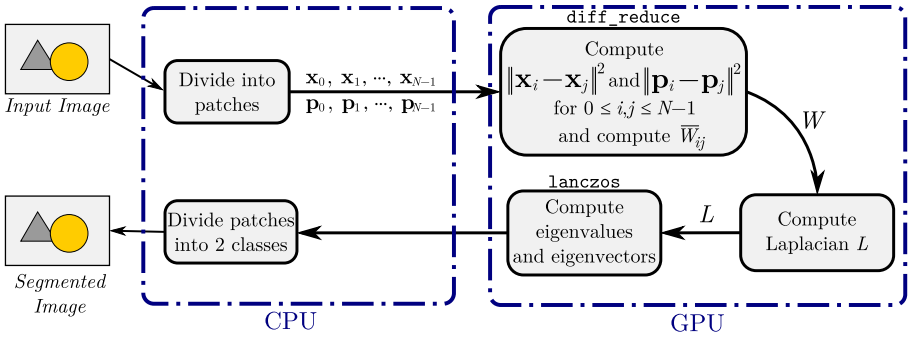


Figure 1: Schematic of the GPU segmentation process

lie in.

The graph Laplacian is a promising approach to compute a low-dimensional representation of the data to preserve local neighborhood information [3]. The manifold could be approximated by the adjacency graph computed from the data points [9]. The Laplace operator is approximated by the weighted Laplacian of the adjacency graph with appropriately assigned weights.

Using the heat equation, the Laplace operator allows us to choose the weight decay function such that the embedding maps for the data approximate the eigenmaps of the Laplace operator [9]. [3] used an initial embedding of local image characteristics in a high-dimensional feature space. The Laplacian eigenmaps are used to describe, parameterize, and visualize the learned data-manifold.

3 Parallel Segmentation on GPU

In this section, we describe a GPU implementation of the algorithm to accelerate the multivariate image segmentation process. The implementation consists of several stages as illustrated in Fig. 1. Some parts have been implemented on the CPU because they are not particularly computationally intensive. In the following sections, we give a detailed description of each stage and our parallel implementation.

3.1 CUDA concepts

Let us first introduce key programming concepts in CUDA. CPUs and main memory are collectively referred to as the *host* whereas GPUs and their on-chip memory are referred to as the *device*. A *kernel* is a routine that executes on the device. Typically, a CPU core launches a kernel to be run on a selected GPU device. Since the host and the device communicate via the PCI Express channel, each kernel launch carries a significant amount of overhead. For the same reason, it is expensive to transfer data between the host and the device. Thus, it is imperative to minimize the number of kernel launches and the amount of data transfer.

To help programmers best utilize the computational power of GPU devices, CUDA offers two units of computation called *blocks* and *threads*. Threads constitute the smallest unit of execution path and keep distinct sets of local variables. Blocks are logical groups of threads that carry certain scheduling implications. For instance, threads that belong to the same block are guaranteed to be scheduled on one streaming multiprocessor (SM). Hence, it is comparatively inexpensive to synchronize threads within a block. In fact, we can issue a compiler intrinsic `__syncthreads()` within a kernel to synchronize threads. Each block is also assigned a special memory area called *shared memory* through which member threads can communicate. Shared memory has a shorter access time than global memory (i.e. rest of GPU memory). For more details in CUDA platform, see [10].

3.2 Generate patches

Given an image to be segmented, we first divide the image into blocks of equal dimensions (e.g. 4 by 4 pixels). We refer to such blocks as *patches*. Each patch is given an index between 0 and $N-1$, where N is the number of patches in the given image. Furthermore, each patch carries two distinct characteristics: a features vector and a position vector. A features vector holds the gray-scale values of the pixels (cf. Fig. 2). A position vector holds the position of the patch with respect to the patch grid. For the rest of the paper, we denote $\mathbf{x}_i$ and $\mathbf{p}_i$ to represent the features vector and the position vector of patch i, respectively. Note that this stage of generating patches will run serially on the CPU because it requires a direct input from an image file.

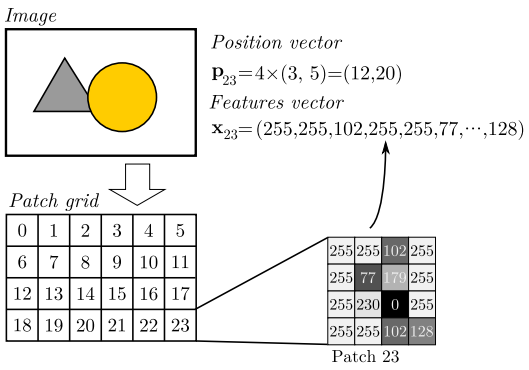


Figure 2: Schematic of feature extraction

3.3 Generate the adjacency graph

Next, we generate an adjacency graph G over the endpoint of the features vectors $\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_{N-1}$ with the following properties:

- Two endpoints that are close enough should be connected with an edge.
- An edge carries less weight as its endpoints are far apart.

To this end, we define the weight matrix W where W_{ij} represents the edge cost between $\mathbf{x}_i$ and $\mathbf{x}_j$:

$$W_{ij} = \begin{cases} \exp\left(-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{M\alpha^2}\right) & \text{if patches } i \text{ and } j \text{ connected} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

where M is the dimension of the patch and α indicates the relative importance of the feature differences in clustering decisions.

We probabilistically determine whether two nodes should be connected or not:

$$P(\text{patches } i \text{ and } j \text{ connected}) = \exp\left(-\frac{\|\mathbf{p}_i - \mathbf{p}_j\|^2}{2\beta^2}\right) \tag{2}$$

where β indicates how much position differences affect clustering decisions.

We use the expected value of weight W_{ij} given by

$$\overline{W}_{ij} = \exp\left(-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{M\alpha^2}\right) \exp\left(-\frac{\|\mathbf{p}_i - \mathbf{p}_j\|^2}{2\beta^2}\right) \tag{3}$$

The task of computing the weight matrix is embarrassingly parallel: there is no data dependency between any two entries of $\overline{W}$. For instance, as long as $\mathbf{x}_u$, $\mathbf{x}_v$, $\mathbf{x}_w$, and $\mathbf{x}_z$ are separately stored in the memory, $\|\mathbf{x}_u - \mathbf{x}_v\|^2$ and $\|\mathbf{x}_w - \mathbf{x}_z\|^2$ can be computed independently of each other.

Let us define difference vector $\mathbf{d}_{ij}$, a $M \times 1$ vector whose k^{th} element is given by

$$\mathbf{d}_{ij}(k) = (\mathbf{x}_i(k) - \mathbf{x}_j(k))^2, \quad 1 \le i, j \le N-1. \tag{4}$$

Similarly, let $\mathbf{f}_{ij}$ be a 2×1 vector whose k^{th} element is given by

$$\mathbf{f}_{ij}(k) = (\mathbf{p}_i(k) - \mathbf{p}_j(k))^2, \quad 1 \le i, j \le N-1. \tag{5}$$

It is crucial to note that the entry sum of $\mathbf{d}_{ij}$ and $\mathbf{f}_{ij}$ should give $\|\mathbf{x}_i - \mathbf{x}_j\|^2$ and $\|\mathbf{p}_i - \mathbf{p}_j\|^2$ respectively.

Let us consider two implementations of computing entries of $\overline{W}$. The first approach explicitly stores the difference vectors and then employs a parallel primitive known as *binary reduction*. Each difference vector is divided into two groups of equal length, and each GPU thread increments one element in the first group by the corresponding element in the second. The first half is then recursively divided so that eventually the first entry will hold the entry sum (cf. Fig. 3). This primitive has $O(\log_2 n)$ step complexity where n is the number of elements being added. Furthermore, the task of computing the difference vectors is embarrassingly parallel and thus easily divided among GPU blocks.

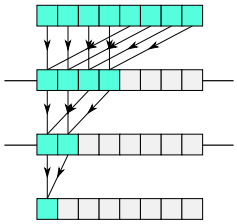
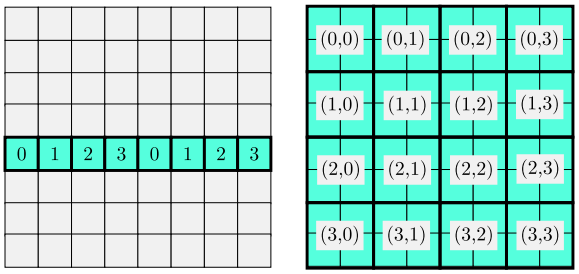


Figure 3: An illustration of binary reduction over 8 elements



(a) $\mathbf{d}_{ij}$ and $\mathbf{f}_{ij}$ are explicitly stored

(b) $\mathbf{d}_{ij}$ and $\mathbf{f}_{ij}$ are not explicitly stored

Figure 4: The portion of the weight matrix $\overline{W}$ that each kernel launch computes in two different scenarios. The boxes with heavy borders represent the way the work is allocated to the 2D grid of GPU blocks.

If we were to compute N^2 entries of $\overline{W}$ at once (i.e. using one kernel launch), we would have to store N^2 difference vectors, occupying M times the space of the weight matrix. Since a typical image often requires more than 10,000 patches, we can afford to store only N difference vectors at a time. This means that we have to make $O(N)$ kernel launches each of which computing one row of $\overline{W}$ (cf. Fig. 4a). At the same time, we would not be launching enough number of blocks to keep the GPU device busy. The cost is too high to offset the benefit of binary reduction. In fact, since M is typically 16 (in case of 4-by-4 patches) or smaller, we do not lose much even if we were to add up the elements sequentially.

On the other hand, the second approach does not store the difference vectors and lets each individual thread compute the entry sums sequentially. If we do not store the difference vectors in the memory, then we can compute the entirety of $\overline{W}$ in a single kernel launch. This approach will help reduce the number of kernel launches and spawn enough number of GPU blocks to keep the device busy.

We divide the weight matrix $\overline{W}$ into a 2D grid of blocks and threads where each thread computes one entry of the matrix (cf. Fig. 4b). Each thread that is in charge of $\overline{W}_{ij}$ computes the entry sums of $\mathbf{d}_{ij}$ and $\mathbf{f}_{ij}$ without actually storing either of the vectors. It does so by sequentially accumulating the partial sums in a per-thread local variable (cf. Fig. 5, Routine 1). It is true that we are forgoing the

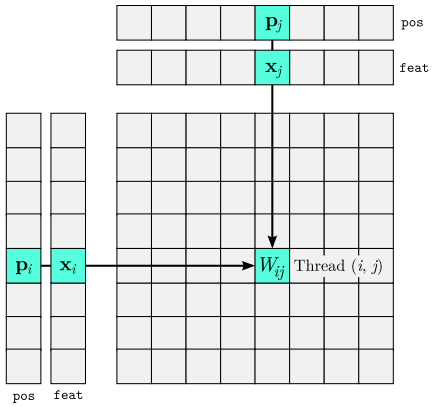


Figure 5: The data access pattern of each thread (i, j) when we do not store the difference vectors.

Routine 1 Computing the weight matrix $\overline{W}$

Given a set of features vectors $\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_{N-1}$ and position vectors $\mathbf{p}_0, \mathbf{p}_1, \cdots, \mathbf{p}_{N-1}$, this routine computes the weight matrix $\overline{W}$. The parameters α and β are given by `par0` and `par1` respectively. On exit, the `dev_w` array contains $\overline{W}$.

```
__global__ void diff_reduce(double *dev_w, double *feat,
   double *pos, int feat_dim, int pos_dim, int par0,
   int par1, int n_patch)
{
   int i = blockIdx.y * blockDim.y + threadIdx.y;
   int j = blockIdx.x * blockDim.x + threadIdx.x;
   double feat_dist = 0.0; // running entry sum of d_ij
   double pos_dist = 0.0;  // running entry sum of f_ij
   int feat_offi = i * feat_dim; // offset of x_i
   int feat_offj = j * feat_dim; // offset of x_j
   int pos_offi = i * pos_dim;   // offset of p_i
   int pos_offj = j * pos_dim;   // offset of p_j
   double feat_i, feat_j, pos_i, pos_j, k;
   // temporary local variables for entry sum calculation

   /* boundary check */
   if (i == j || i >= n_patch || j >= n_patch)
      return;
   /* thread (i, j) computes W_ij */
   // get the k-th element of difference vector d_ij
   // and add it to feat_dist
   for (k = 0; k < feat_dim; k++) {
      feat_i = feat[feat_offi + k];
      feat_j = feat[feat_offj + k];
      feat_dist += (feat_i - feat_j) * (feat_i - feat_j);
   }
   // get the k-th element of difference vector f_ij
   // and add it to pos_dist
   for (k = 0; k < pos_dim; k++) {
      pos_i = pos[pos_offi + k];
      pos_j = pos[pos_offj + k];
      pos_dist += (pos_i - pos_j) * (pos_i - pos_j);
   }
   dev_w[i + j * n_patch]
      = exp( -feat_dist / (feat_dim * par0 * par0))
        * exp( -pos_dist / (pos_dim * par1 * par1));
}
```

benefit of binary reduction, but the cost is not too high compared to the performance advantage we gain by reducing the number of kernel launches down to one. In fact, the second approach produces the weight matrix 10 times faster than the first approach.

3.4 Compute the Laplacian matrix

Our goal is to cluster patches according to their features. We define this problem in terms of a map [3]. We would like to map the nodes of G to points on the real number line so that connected points stay as close as possible. In other words, we seek to assign a scalar value y_i to each $\mathbf{x}_i$ such that the objective function

$$\sum_{ij}(y_i - y_j)^2 \overline{W}_{ij} \tag{6}$$

is minimized. Recall that $\overline{W}_{ij}$ decays exponentially as $\mathbf{x}_i$ and $\mathbf{x}_j$ grow farther apart. It turns out that finding a map $\mathbf{y} = (y_0, y_1, \cdots, y_{N-1})$ can be reduced to the eigenvalue problem $L\mathbf{y} = \lambda\mathbf{y}$ where L is the Laplacian matrix [3]. The Laplacian matrix L is given by

$$L = I - D^{-1/2}\overline{W}D^{-1/2} \tag{7}$$

where D is a diagonal matrix whose i^{th} diagonal entry is equal to the entry sum of the i^{th} column of $\overline{W}$. To simplify our discussion, let us denote the diagonal entries of D as $d_0, d_1, \cdots, d_{N-1}$.

It is straightforward to compute D. We use a binary reduction to compute the entry sum of each column of $\overline{W}$ and then compute $D^{-1/2}$ by taking the reciprocal of the square root of each diagonal entry.

We discuss two approaches to obtaining the Laplacian matrix L. If we store D as a dense matrix, then we can easily compute L by making two calls to `cublasDsymm` routine of cuBLAS library. The Level 3 routine performs the general symmetric matrix-matrix operation of the form $C = \alpha AB + \beta C$ where either A or B is a symmetric matrix [11].

This approach has a benefit of concise code, but it has two significant drawbacks. First, it is wasteful to store D as a dense matrix because all the non-diagonal entries are zero. GPU memory is still a relatively scarce resource, typically no more than a few gigabytes. Second, the Level 3 routine is $O(N^3)$ because it assumes all the entries to be nonzero.

Meanwhile, the second approach stores D as a series of row and column operations. The identity matrix I can be transformed into D by multiplying the first row by d_0, the second row by d_1 and so forth. Hence, the diagonal matrix is a superposition of N elementary matrices corresponding to row multiplications:

$$D = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & \ddots & \\ & & & d_{N-1} \end{bmatrix} \cdots \begin{bmatrix} 1 & & & \\ & d_1 & & \\ & & \ddots & \\ & & & 1 \end{bmatrix} \begin{bmatrix} d_0 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{bmatrix} I. \tag{8}$$

By a similar argument, D is also a superposition of N elementary matrices that correspond to column multiplications:

$$D = I \begin{bmatrix} d_0 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{bmatrix} \begin{bmatrix} 1 & & & \\ & d_1 & & \\ & & \ddots & \\ & & & 1 \end{bmatrix} \cdots \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & \ddots & \\ & & & d_{N-1} \end{bmatrix}. \tag{9}$$

It follows that multiplying each row i of $\overline{W}$ by $1/\sqrt{d_i}$ and then multiplying each column j of $\overline{W}$ by $1/\sqrt{d_j}$ should produce the desired product $D^{-1/2}\overline{W}D^{-1/2}$. We obtain the Laplacian matrix L by subtracting the product $D^{-1/2}\overline{W}D^{-1/2}$ from the identity matrix I. This formulation involves $2N$ vector multiplications and one matrix addition, bringing the total cost to only $O(N^2)$. It represents an $O(N)$ speedup over the previous approach. Furthermore, it costs only $O(N)$ in memory space to store the diagonal matrix D.

In practice, we compute L in-place and overwrite the content of the matrix $\overline{W}$. We do so to conserve scarce GPU memory. We first compute the product $D^{-1/2}\overline{W}D^{-1/2}$ in-place by applying a series of row and column multiplications to the matrix $\overline{W}$. Then we flip the sign of $\overline{W}$ and increment its diagonal entries by 1. See Routine 2 for details.

The series of row and column multiplications would normally require $2N$ calls to a Level 1 routine such as `cublasDscal` that performs scalar-vector multiplication. However, for the particular series of multiplications induced by a diagonal matrix, the cuBLAS library offers a shorthand: the routine named `cublasDdgmm` performs multiplication by a diagonal matrix, either to the left or to the right [11]. The routine is called exactly once, so it has much less overhead. The only disadvantage is that the routine is not part of the standard BLAS interface.

3.5 Compute the eigenvalues and eigenvectors

It remains to solve the eigenvalue problem

$$L\mathbf{y} = \lambda\mathbf{y}. \tag{10}$$

In general, we solve a symmetric eigenvalue problem by reducing the system matrix into a tridiagonal form via a series of orthogonal similarity transformations. There are several well-known methods to extract eigenvalues of a symmetric tridiagonal matrix. In particular, the divide-and-conquer algorithm is numerically stable and efficient at computing the full spectrum of eigenvalues of a symmetric tridiagonal system [12]. This procedure for multiple CPU and GPU cores has been implemented in the MAGMA library [13]. As it turns out, however, we are mostly interested in the extreme eigenvalues; the reason is shown in the following section. A more efficient method exists to compute a few select extreme eigenvalues.

The *Lanczos method* is an iterative method that builds a symmetric tridiagonal matrix T whose spectrum of eigenvalues approximates that of a given symmetric matrix A (cf. Algorithm 1). The matrix T is the tridiagonal matrix with α_i on the main diagonal and β_i on the two subdiagonals. With sufficient number of iterations, the extreme eigenvalues of T approximates their counterpart in A [12]. To extract the corresponding eigenvectors of A, we first collect the orthonormal set of Ritz vectors $\mathbf{q}_i$ into an orthogonal

Routine 2 Computing the Laplacian matrix L

Given the weight matrix $\overline{W}$, this routine first computes the diagonal matrix D and then the Laplacian matrix L. On exit, the `dev_w` array contains L.

```
#define THREADS_PER_BLOCK 128
#define BLOCKS_PER_GRID 2048
void laplacian(double *dev_w, int n_patch)
{
   // declare and allocate necessary variables
      ...
   // Compute diagonal matrix D^(-1/2)
   diag<<<BLOCKS_PER_GRID, THREADS_PER_BLOCK>>>(dev_d,
      dev_w, n_patch);
   // W <- D^(-1/2) * W * D^(-1/2)
   cublasDdgmm(handle, CUBLAS_SIDE_LEFT, n_patch, n_patch,
      dev_w, n_patch, dev_d, 1, dev_w, n_patch);
   cublasDdgmm(handle, CUBLAS_SIDE_RIGHT, n_patch, n_patch,
      dev_w, n_patch, dev_d, 1, dev_w, n_patch);
   // L <- I - W
   compute_l<<<BLOCKS_PER_GRID, THREADS_PER_BLOCK>>>(dev_w,
      n_patch);
   // clean up
      ...
}
__global__ void diag(double *dev_d, double *dev_w,
   int n_patch)
{
   int b = blockIdx.x;
   int i, j, size;
   __shared__ double cache[THREADS_PER_BLOCK];
   /* binary reduction computes sum of b-th column */
   while (b < n_patch) {
      size = THREADS_PER_BLOCK / 2;
      i = threadIdx.x;
      j = i;
      cache[i] = 0.0;
      while (j < n_patch) {
         // load partial sums into shared memory
         cache[i] += dev_w[b * n_patch + j];
         j += THREADS_PER_BLOCK;
      }
      __syncthreads();
      // reduce the shared array into one output
      while(size != 0) {
         if (i < size)
            cache[i] += cache[i+size];
         __syncthreads();
         size /= 2;
      }
      if (i == 0)
         dev_d[b] = 1/sqrt(cache[0]);
      __syncthreads();
      b += BLOCKS_PER_GRID;
   }
}
__global__ void compute_l(double *dev_w, int n_patch)
{
   int tid = threadIdx.x + blockIdx.x * blockDim.x;
   int N = n_patch * n_patch;
   while (tid < N) {
      dev_w[tid] =
          ((tid % (n_patch + 1) == 0)? 1 : 0) - dev_w[tid];
      tid += blockDim.x * gridDim.x;
   }
}
```

matrix Q_k as follows:

$$Q_k = [\mathbf{q}_1 \quad \mathbf{q}_2 \quad \cdots \quad \mathbf{q}_k] \tag{11}$$

Then we multiply each eigenvector $\mathbf{u}_i$ of T by Q_k to the left to get the corresponding eigenvector $\mathbf{v}_i$ of A. In practice, we compute all of the k eigenvectors simultaneously as follows:

$$[\mathbf{v}_1 \quad \mathbf{v}_2 \quad \cdots \quad \mathbf{v}_k] \leftarrow Q_k[\mathbf{u}_1 \quad \mathbf{u}_2 \quad \cdots \quad \mathbf{u}_k] \tag{12}$$

For many classes of matrices, round-off errors cause

Algorithm 1 The Lanczos method [12]

Given a symmetric matrix A and an iteration count k, this algorithm constructs the entries α_i, β_i of the symmetric tridiagonal T whose extreme eigenvalues approximate those of A. The approximation is accurate if k is large compared to the number of extreme eigenvalues sought.

1: $\beta_0 \leftarrow 0$, $\mathbf{q}_0 \leftarrow \mathbf{0}$, $\mathbf{q}_1 \leftarrow$ a random unit vector
2: **for** $i \leftarrow 1$ to k **do**
3: $\quad \mathbf{z} \leftarrow A\mathbf{q}_i$
4: $\quad \alpha_i \leftarrow \mathbf{q}_i^T \mathbf{z}$
5: $\quad \mathbf{z} \leftarrow \mathbf{z} - \alpha_i \mathbf{q}_i - \beta_{i-1}\mathbf{q}_{i-1}$
6: $\quad \mathbf{z} \leftarrow \mathbf{z} - \sum_{j=1}^{i-1}(\mathbf{z}^T\mathbf{q}_j)\mathbf{q}_j, \mathbf{z} \leftarrow \mathbf{z} - \sum_{j=1}^{i-1}(\mathbf{z}^T\mathbf{q}_j)\mathbf{q}_j$
7: $\quad \beta_i \leftarrow \|\mathbf{z}\|_2$
8: $\quad \mathbf{q}_{i+1} \leftarrow \mathbf{z}/\beta_i$
9: **end for**

the set of Ritz vectors $\mathbf{q}_i$ to lose its orthogonality [12, 14]. One effective remedy is to reorthogonalize the vectors by applying a step of the Gram-Schmidt process shown in Line 6 of Algorithm 1, which eliminates all the components of $\mathbf{z}$ that are parallel to any of $\mathbf{q}_1, \mathbf{q}_2, \cdots, \mathbf{q}_{i-1}$. It turns out that twice is enough to guarantee the orthogonality of $\mathbf{z}$ to $\mathbf{q}_1$ through $\mathbf{q}_{i-1}$ [15].

To best utilize BLAS, we express the Gram-Schmidt step in a matrix form:

$$\mathbf{z} \leftarrow \mathbf{z} - Q_{i-1}Q_{i-1}^T\mathbf{z} \tag{13}$$

where

$$Q_{i-1} = [\mathbf{q}_1 \quad \mathbf{q}_2 \quad \cdots \quad \mathbf{q}_{i-1}] \tag{14}$$

This formulation neatly translates into two calls to a BLAS 2 routine.

3.6 Compute clustering

The solutions to the eigenvalue problem $L\mathbf{y} = \lambda\mathbf{y}$ represent a map $\mathbf{y} = (y_0, y_1, \cdots, y_{N-1})$ that projects the vectors $\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_{N-1}$ to points on a Euclidean space. Specifically, the number of extreme eigenvectors we use determines the dimension of the Euclidean space. For now, let us pick the second-smallest eigenvalue and the corresponding eigenvector so that the features vectors are projected onto the one-dimensional real number line. This allows us to pick a simple clustering algorithm. Since we are segmenting the given image into two monochrome classes, thresholding is a good choice. One of the classes will be plotted in black (to indicate the segmented parts) and others in white.

4 Numerical Experiment

Our testing platform comprised dual 2.0 GHz Intel Xeon® CPUs with a total of 64 GB of main memory, one Tesla K20c graphics card with 5 GB of memory, and CUDA 5.0 runtime running on the Linux operating system. Our experiments used double-precision floating point arithmetic.

Table 1: Performance for various inputs

Cases	Fig. 9a	Fig. 9b	Fig. 9c
Image size (pixels)	256×96	284×284	360×300
Patch size (pixels)	4×4	4×4	4×4
Number of patches	1536	5041	6750
Number of Lanczos iterations	12	48	50
Single-thread CPU (sec)	0.71	7.84	13.90
GPU (sec)	1.45	1.67	1.84
Speedup	0.49×	4.69×	7.55×

Cases	Fig. 9d	Fig. 9e	Fig. 9f
Image size (pixels)	448×352	612×372	784×480
Patch size (pixels)	4×4	4×4	4×4
Number of patches	9856	14229	23520
Number of Lanczos iterations	199	250	250
Single-thread CPU (sec)	42.45	92.95	265.31
GPU (sec)	3.92	8.00	18.87
Speedup	10.83×	11.62×	14.06×

In addition, all the implementations under testing have been compiled from the sources on our testing platform. We used ATLAS (Automatically Tuned Linear Algebra Software) as the reference BLAS implementation.

The GPU implementation delivers larger performance boosts to larger images (cf. Table 1). Furthermore, the results (cf. Fig. 9) show accurate segmentation compared to the original, with the underlying vessel structures well preserved.

Let us see which stage benefited the most. For the largest input (Fig. 9f), the first two stages experienced a speedup in the order of 100 whereas the eigensolver stage experienced a speedup of only 5 (cf. Table 2). As a result, the eigensolver stage occupied 84% of compute time (cf. Fig 6). This stage consists of many BLAS 1 and BLAS 2 operations, which do not scale up quite readily.

It should be also noted that memory allocation took a substantially longer time on the GPU. Unlike on the host, where memory allocation could take anywhere between 0.01 seconds and 3.17 seconds, memory allocation took a relatively constant time (about 1.43 seconds) on the GPU. We suspect that the need for communicating over the PCI Express channel puts a lower bound on the overhead.

To put performance figures in context, we created a multithreaded CPU implementaion of eigenmap. We focus on the two stages that received a substantial boost on the GPU. Certainly, the stages exhibit a strong scaling with respect to the number of CPU cores, experiencing a speedup of 10 on top of 12 physical cores (cf. Fig. 7 and 8). Unfortunately, the CUDA API does not offer a mechanism to limit the number of CUDA cores to which work is assigned. The 10x speedup over the multithreaded CPU implementation looks good enough, however, given that cuBLAS delivers about 5x speedup over Intel's Math Kernel Library (MKL) for ZGEMM [16].

We repeated the experiment using a general symmetric eigenvalue solver instead of the Lanczos method. Other parts of the implementation were kept the same. Our GPU

Table 2: Performance by stages, Fig. 9f

Stages	pairweight	laplacian	eigenvalue	malloc
Single-thread CPU (sec)	146.95	24.82	90.01	3.17
GPU (sec)	1.29	0.22	15.85	1.43
Speedup	113.91×	112.82×	5.68×	2.22×

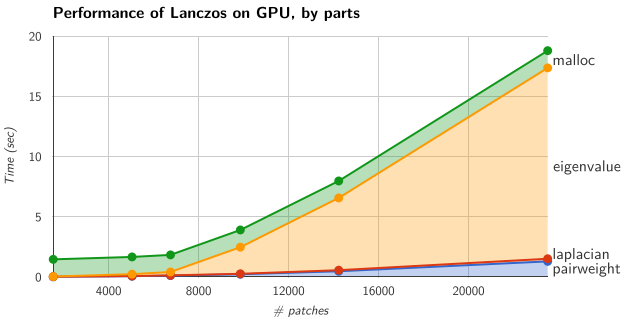


Figure 6: Performance by stages of eigenmap segmentation on GPU

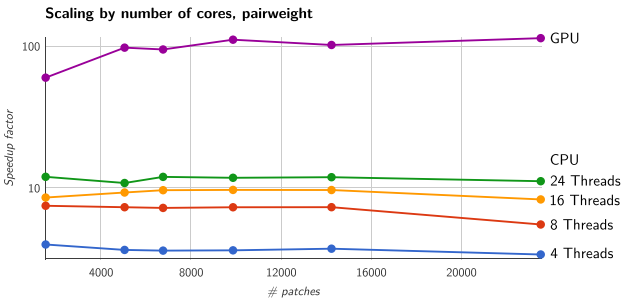


Figure 7: Performance of pairweight stage on multithreaded CPU and GPU

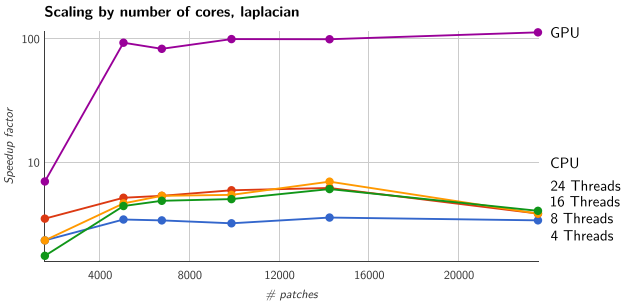
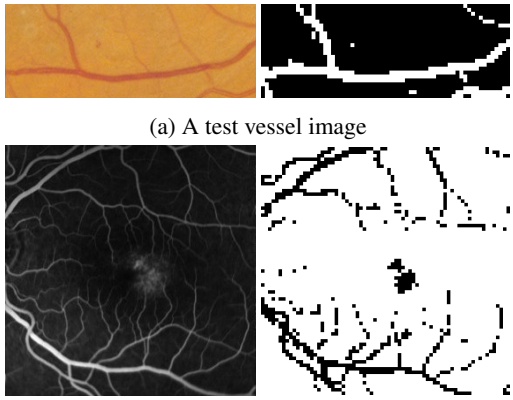


Figure 8: Performance of laplacian stage on multithreaded CPU and GPU

implementation of the Lanczos method performed up to 28 times faster than the routine `magma_dsyevdx` of the MAGMA library. This translated into a speedup of 21 over the total running time.

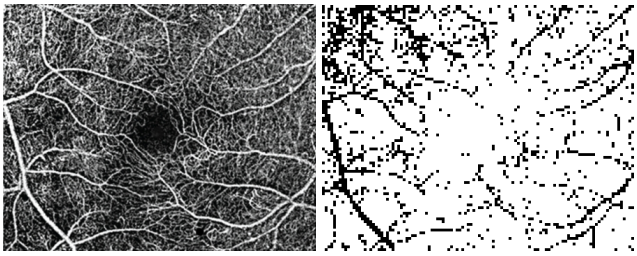
We also repeated the experiment without reorthogonalizing the Ritz vectors $\mathbf{q}_i$. The reorthogonalization step imposed only a modest 8% penalty on performance.



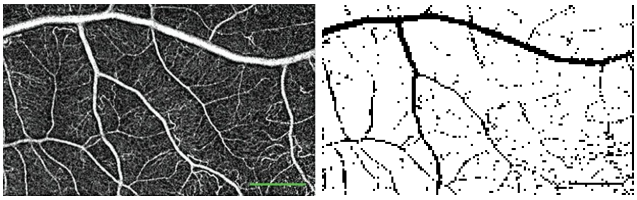
(a) A test vessel image

(b) A noninvasive capillary-perfusion map (nCPM) of retinal vessels [17]

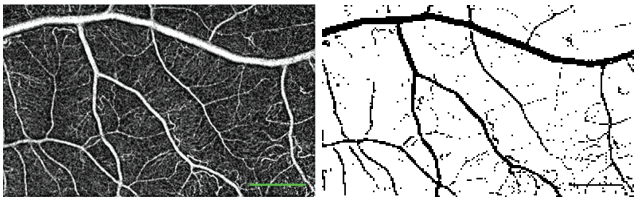
(c) A vascular pattern of skeletal muscle in mammals [18]



(d) Another nCPM of retinal vessels [19]



(e) Yet another nCPM of retinal vessels [19]



(f) A high-resolution version of Fig. 9e [19]

Figure 9: Segmented images of various size

5 Conclusion

In this paper we presented an efficient parallel implementation of Laplacian eigenmap-based vessel segmentation on a GPU. Our algorithm was also designed to exploit the massive data parallelism of the underlying hardware when map-

ping datasets onto logical block and thread structures. We demonstrated that virtually all of the segmentation steps can be done in a highly parallel fashion. These include weight matrix generation, computing the Laplacian matrix, solving the associated eigenvalue problem, and clustering. Our approach also avoids computing the entire spectrum of eigenpairs by determining the dimension of the projected Euclidean space using only extreme eigenvectors. Our numerical expriments have shown that the performance of our GPU-based vessel segmentation procedure outperformed that of a CPU-based approach by a factor of 14 for large image datasets. This provides a solid case for GPU-based image segmentation strategies to overcome high computational requirements for large multivariate image datasets.

Acknowledgements

The research of Hyunsu Cho was supported by Trinity College under the Student Research Program. The authors would like to thank Nvidia Corporation for providing GPUs under CUDA Teaching Center Program. The authors would also like to thank Jiajia Zhao for her assistance with the development of earlier versions of software.

References

[1] J.C. Bezdek, L.O. Hall and L.P. Clarke, Review of MR image segmentation techniques using pattern recognition, *Medical Physics*, 20(4), 1993, 1033-1048.

[2] E. Gokcay, *A New Clustering Algorithm for Segmentation of Magnetic Resonance Images* (Gainesville, FL: University of Florida, 2000). Ph.D. Thesis.

[3] I. Tziakos, N. Laskaris and S. Fotopoulos, Multivariate Image Segmentation Using Laplacian Eigenmaps, *European Signal Processing Conference*, Vienna, Austria, 2004, 945-948.

[4] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A.E. Lefohn and T.J. Purcell, A Survey of general-purpose computation on graphics hardware, *Computer Graphics Forum*, 26(1), 2007, 80-113.

[5] S.S. Samant, J. Xia, P. Muyan-Özçelik and J.D. Owens, High performance computing for deformable image registration: Towards a new paradigm in adaptive radiotherapy, *Medical Physics*, 35(8), 2008, 3546-3553.

[6] P. Harish and P.J. Narayanan, Accelerating large graph algorithms on the GPU using CUDA, *International Conference on High Performance Computing*, Goa, India, 2007, 197-208.

[7] P. Indyk, Dimensionality reduction techniques for proximity problems, *Annual ACM-SIAM Symposium on Discrete Algorithms*, San Francisco, USA, 2000, 371-378.

[8] P. Indyk, Better algorithms for high-dimensional proximity problems via asymmetric embeddings, *Annual ACM-SIAM Symposium on Discrete Algorithms*, San Francisco, USA, 2003, 539-545.

[9] M. Bernstein, V.D. Silva, J.C. Langford and J.B. Tenenbaum, Graph approximations to geodesics on embedded manifolds, Department of Psychology, Stanford University, 2000. Technical Report.

[10] Nvidia, CUDA C Programming Guide.

[11] Nvidia, cuBLAS Library User Guide.

[12] D. S. Watkins, *Fundamentals of Matrix Computations* (New York: John Wiley and Sons Incs., 1991).

[13] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek and S. Tomov, Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects, *Journal of Physics: Conference Series*, 180(1), 2009, 12-37.

[14] J. Demmel, *Applied Numerical Linear Algebra* (Philadelphia: SIAM, 1997).

[15] B. Parlett, *The Symmetric Eigenvalue Problem* (Englewood Cliffs, NJ: Prentice-Hall, 1980).

[16] Nvidia, CUDA Toolkit 5.0 Performance Report.

[17] A.J. Witkin, R.A. Alshareef, S.S. Rezeq, K.M. Sampat, J. Chhablani, D.U. Bartsch, W.R. Freeman, J.A. Haller and S.J. Garg, Comparative analysis of the retinal microvasculature visualized with fluorescein angiography and the retinal function imager, *American Journal of Ophthalmology*, 154(5), 901-907.e2.

[18] S. Selinger, The Krogh Cylinder, Rice University, 2004. Student Project. http://www.owlnet.rice.edu/~ceng402/proj04/seli/CENG402_Project_Files/Blank Page 2.htm

[19] D.A. Nelson, Z. Burgansky-Eliash, H. Barash, A. Loewenstein, A. Barak, E. Bartov, T. Rock and A. Grinvald, High-resolution wide-field imaging of perfused capillaries without the use of contrast agent, *Clinical Ophthalmology*, 5, 2011, 1095-1106.