Tuning Principal Component Analysis for GRASS GIS on Multi-core and GPU Architectures

Peng Du[1], Matthew Parsons[2], Erika Fuentes[3], Shih-Lung Shaw[4], Jack Dongarra[5]

[1]Department of Electical Engineering and Computer Science, du@cs.utk.edu
[2]Cascadia Community College, mparsons@cascadia.edu
[3]Microsoft, efuente@microsoft.com
[4]Department of Geography, sshaw@utk.edu
[5]Department of Electical Engineering and Computer Science, dongarra@eecs.utk.edu

Abstract

This paper presents optimizations to Principal Component Analysis (PCA) in GRASS GIS. The current implementation of PCA in GRASS is based on eigenvalue decomposition, which does not have high memory requirements but it can suffer from low runtime performance. In modern computers, significant performance improvements can be achieved by appropriately taking advantage of the memory configuration (hierarchy). A common way of doing this is data reuse for frequent operations. The GRASS PCA is only reusing data at a very high level, i.e., at main memory and I/O level. We can improve the implementation of the PCA methodology by re-arranging the computations of matrix operations, and using available high performance packages that use block algorithms to optimize data reuse. We can achieve further optimizations by taking advantage of the now popular multi-core architectures and the use of Graphic Processing Units (GPU). By taking into account all these key supercomputing components, GRASS GIS can leverage massively parallel computing without requiring a supercomputer. The speed-up that can be achieved by using multi-core CPU and GPU architectures will greatly improve the efficiency of GRASS GIS for its users. We use imaging spectrometer data to demonstrate the performance improvements attained by our implementation, which reduced runtime by nearly 99% using only multi-core related optimizations and an additional 50% reduction using GPU related optimizations.

1. Introduction

In the past, much of the innovation in CPUs had previously focused on increasing the number of clock cycles in a single core. Nowadays most computers are equipped with multi-core CPUs, which are more power efficient than machines with multiple CPUs. Software applications –like GRASS GIS– have to continuously redesign its algorithms to benefit from the increasing computing capabilities of the hardware. Additionally, from a software perspective, it is relatively easy to take advantage of existing high performance math libraries to offload the bulk of the computations to these highly specialized and optimized routines.

Another important hardware component to consider is the GPU, which until recently, had traditionally been limited to perform graphical operations. GPU hardware improvements and

increased accessibility to software packages such as CUDA (Compute Unified Device Architecture) has generated tremendous interest in using the GPU for scientific computing. A good example is the NVIDIA GPU, which is composed of groups of “stream processors”, each of which contains an array of cores and performs the same type of operations concurrently while sharing resources like registers and shared memory (this describes Single Instruction Multiple Data model, a well-known concept in the scientific computing community). Commonly used algorithms like PCA, can be easily optimized to exploit these characteristics of the GPU architecture. Additional improvements can be made by making use of the “offsite memory” on the GPU, mitigating the increased memory requirements frequently caused by doing CPU optimizations.

In our work, we redesign the GRASS GIS PCA methodology to take advantage of these concepts and the associated potential space for improvements. Our experiments focus on imaging spectrometer data, commonly called hyperspectral data, which is a remote sensing dataset that covers a wide range of the electromagnetic spectrum. In particular, we use data obtained by AVIRIS (Airborne Visible/Infrared Imaging Spectrometer). The AVIRIS three-dimensional dataset, or image cube, provides images with wavelengths from 400 to 2500 nanometers in 224 contiguous spectral channels (or bands). Two dimensions define the area of coverage and the 3rd dimension corresponds to a range of the electromagnetic spectrum and is referred to as the spectral band. Imaging spectrometer data is widely used in GIS to provide spectral information which for instance helps generate finer classification results (Zhang et al 2006). GRASS GIS has incorporated functions (Neteler, 2004) for imaging spectroscopy processing. PCA is an important statistical methodology that is commonly used to reduce the dimensionality of datasets; GRASS GIS can take advantage of this for false-color viewing or classification.

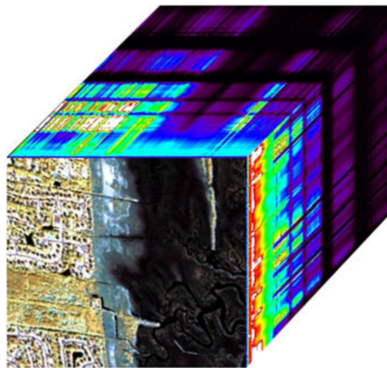

Figure 1: AVIRIS image cube representing an image (RGB = 42, 90, 140) of a coastal area in Charleston, SC (http://www.csc.noaa.gov/crs/rs_apps/sensors/hyperspec.htm)

The current implementation of PCA in GRASS GIS uses an EVD (eigenvalue decomposition) based algorithm (Richards et al 2006). This algorithm features covariance matrix computation and matrix multiplication based data re-projection whose performance degradation becomes prohibitive for large data sets. Previous work discussed in (Andrecut 2009) has addressed a PCA implementation that avoids the time consuming computations in the EVD algorithm by generating only the first few dimensions (or principal components) in the PCA by using an iterative method (Wold et al 1987)(Kramer 1998). EVD based PCA has been evaluated (Castaño-Díez et al 2008)

to use GPU but the speed-up is limited by using only CUBLAS (CUDA CUBLAS LIBRARY 2010) libraries on older generation GPUs. To the best knowledge of the authors, no work has been presented related to an EVD-based GPU implementation yet, however a request has been made for an EVD-based GPU implementation (http://forums.nvidia.com/index.php?showtopic=157531).

The rest of this document analyzes the performance of the current implementation and makes optimizations targeting multi-core CPU and GPU, and it is organized as follows: Section 2 explains the EVD algorithm and implementation in GRASS GIS and decomposes the performance to locate bottlenecks. Section 3 and 4 addresses the bottlenecks exploiting the features of multi-core CPU and GPU. Section 5 presents the experiment results. Throughout the work we use the GRASS 7 code from SVN repository (http://trac.osgeo.org/grass/wiki/DownloadSource)

2. Principal Component Analysis in GRASS GIS

2.1 Algorithm

PCA is a linear projection method used for linear dimensionality reduction. It identifies the orthogonal directions of maximum variance in an original set of vectors, and projects these onto a lower dimension space formed by a subset of the highest variance components (Jackson 2003). Each new dimension is called a principal component. The magnitude of variance is sorted from large to small along the sequence of the dimension, i.e., the first PC has the largest variance. PCA can produce an optimal transformation in terms of least mean square error.

In the context of our work, each image pixel is part of the input dataset of PCA. Each of them has n spectral bands and is regarded as a random vector $\vec{x} = (x_1, x_2, \cdots, x_n)^T$. The AVIRIS image that is used in our work has $n = 224$. The objective of PCA is to look for a linear transformation G such that the covariance matrix of $\vec{y} = G\vec{x}$ is diagonal (see Equation [1]).

$$\textstyle\sum_{\vec{y}} = \frac{1}{n}\sum_{i=1}^{n}(G\vec{x} - G\bar{x})\,(G\vec{x} - G\bar{x})^T$$

$$= G\left(\frac{1}{n}\sum_{i=1}^{n}(\vec{x} - \bar{x})\,(\vec{x} - \bar{x})^T\right)G^T = G\textstyle\sum_{\vec{x}}G^T \qquad [1]$$

$\sum_{\vec{y}}$ and $\sum_{\vec{x}}$ are the covariance of $\vec{y}$ and $\vec{x}$ respectively, and n is the total number of samples.

Since $\sum_{\vec{y}}$ has to be diagonal and G is an orthogonal matrix, Equation [1] can be rewritten as:

$$\boldsymbol{C} = \left(\frac{1}{n}\sum_{i=1}^{n}(\vec{x} - \bar{x})\,(\vec{x} - \bar{x})^T\right) = G^T\textstyle\sum_{\vec{y}}G \qquad [2]$$

$\boldsymbol{C}$ is now the covariance matrix.

Equation [2] is actually the eigenvalue decomposition of the covariance matrix of the input dataset, making this PCA algorithm EVD-based (Algorithm 1 outlines its main steps).

Algorithm 1: EVD-based PCA

I. Creating mean values vector: $\bar{x}$
II. Mean centering: $\vec{x} - \bar{x}$
III. Create covariance matrix: $\boldsymbol{C} = \frac{1}{n}\sum_{i=1}^{n}(\vec{x} - \bar{x})\,(\vec{x} - \bar{x})^T$
IV. Eigenvalue decomposition: $\mathbf{C} = G^T \sum_{\vec{y}} G$
V. Transpose of eigenvector matrix G^T
VI. Projection $\vec{x}$ to new space $\vec{y} = G^T\vec{x}$'
VII. (optional) Scaling $\vec{y}$

From Algorithm 1, steps III and VI account for most of the computation time because both have $O(n^3)$ complexity. Even though the eigenvalue decomposition (step IV) also has that same complexity, the number of possible bands in imaging spectrometer data nowadays is still in the hundreds, which takes a relatively short time to complete compared to steps III and VI. These two steps are key in the understanding of the performance of the current PCA implementation in GRASS.

2.2 Covariance Matrix Creation

For the covariance matrix creation, Equation [2] can be extended as shown in Equation [3] for implementation purposes.

$$\boldsymbol{C} = \frac{1}{n}\sum_{i=1}^{n}(\vec{x} - \bar{x})\,(\vec{x} - \bar{x})^T$$
$$= \frac{1}{n}\sum_{i=1}^{n}\left(\begin{bmatrix} x_{1i} - \overline{x_1} \\ \vdots \\ x_{mi} - \overline{x_m} \end{bmatrix}[(x_{1i} - \overline{x_1}), \quad \cdots \quad, (x_{mi} - \overline{x_m})]\right)$$
$$= \frac{1}{n}\begin{pmatrix} \sum_{i=1}^{n}(x_{1i} - \overline{x_1})^2 & \cdots & \sum_{i=1}^{n}(x_{1i} - \overline{x_1})(x_{mi} - \overline{x_m}) \\ \vdots & \ddots & \vdots \\ \sum_{i=1}^{n}(x_{mi} - \overline{x_n})(x_{1i} - \overline{x_1}) & \cdots & \sum_{i=1}^{n}(x_{mi} - \overline{x_m})^2 \end{pmatrix} \qquad [3]$$

where n is the total number of samples, and m is the total number of bands.

From Equation [3], it can be seen that an element at row i and column j in the covariance matrix is calculated by taking all the data points in bands i and j, mean centering and then multiplying each corresponding pair of elements from the two bands, and finally adding this to the result.

GRASS uses a row-buffer method to implement the algorithm in Equation [3], which adheres to the software infrastructure and avoids duplicating data. The imaging spectrometer data is handled as a multiple-layer raster image (referred to as a “map”) by GRASS, which provides a set of APIs to access raster image files. The input dataset, or image cube, is split into series of 2D images by

spectral bands. Only a few rows of the image in a particular band can reside in the memory buffer at a time, as illustrated by Figure 2.

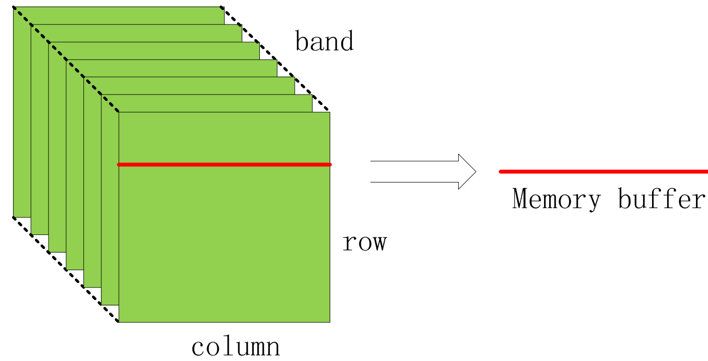

Figure 2: Memory buffer in the PCA of GRASS 7

The dataset is placed into memory through the raster API get_map_row(…) one row at a time; beforehand, cells are checked for NULL in each element to ensure data constancy and results reliability.

Figure 3 shows the covariance matrix creation. GRASS uses two internal buffers. One row in band i is read in by get_map_row(…) to buffer 1; the same row in band j where $j > i$ is read into buffer 2. The data in these two buffers are then mean centered and added up to generate a partial result of the element in row i column j of the covariance matrix. All partial results are completed as a loop goes through all the rows.

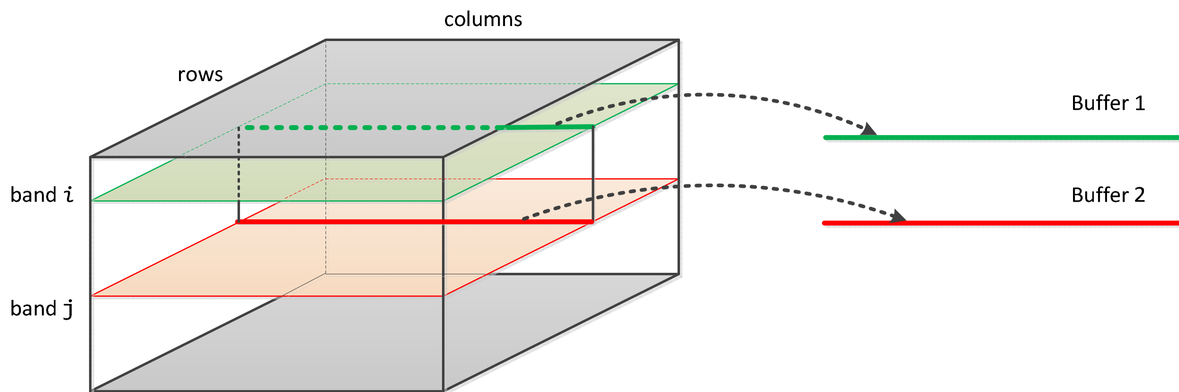

Figure 3: Creating covariance matrix through internal buffer

2.3 Projection

The projection routine applies the eigenvector matrix to the entire dataset: $\vec{y} = G^T\vec{x}$. GRASS's implementation of this step is illustrated in Figure 4.

After step V in Algorithm 1, the eigenvectors are stored in a row major matrix where row i contains the ith eigenvector. To utilize the row-buffer mechanism like in the covariance matrix creation step, GRASS carries out the computation as follows:

Algorithm 2: Projection

I. Take row i of the eigenvector matrix, denoted as $[a_1, a_2, \cdots, a_m]$, where m is the number of bands

II. Take a slice of the image cube by row r (red rectangle in Figure 4), denoted as $B = \begin{bmatrix} \overrightarrow{b_1} \\ \vdots \\ \overrightarrow{b_m} \end{bmatrix}$, where $\overrightarrow{b_i}$ $(1 \leq i \leq m)$ is one row of the image

III. Compute $\vec{z} = [a_1, a_2, \cdots, a_m] \begin{bmatrix} \overrightarrow{b_1} \\ \vdots \\ \overrightarrow{b_m} \end{bmatrix} = a_1\overrightarrow{b_1} + a_2\overrightarrow{b_2} + \cdots + a_m\overrightarrow{b_m}$ and write to row r in band i

IV. Loop through all the rows to complete the result in band i (green plane in Figure 4)

V. Loop through all the bands, and complete the projection

In step II $\overrightarrow{b_i}$ $(1 \leq i \leq n)$ is fetched from memory and stored in an row buffer one at a time and checking for NULL, it is then scaled by scalar $a_i (1 \leq i \leq n)$. Note that the first time the image data is used in GRASS GIS, it is read from disk and buffered into memory for future use; this is a common way to improve performance by avoiding disk I/O overhead. In particular, the get_map_row() API, buffers the data when it is first called to compute mean values, when it is used again in the projection step the operation only carries the overhead of memory copy.

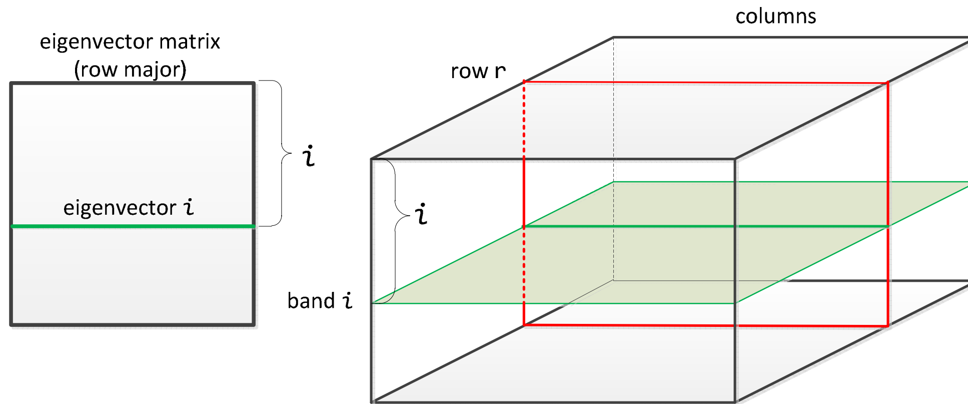

Figure 4: Projection

2.4 Other routines

Mean values calculation and scaling (step I and VII in Algorithm 1) are the least computation intensive, but they are time consuming routines in GRASS PCA implementation. In essence both routines are $O(n)$ operations which are memory-bound. In each mean-values calculation, an internal buffer is used and one row in a band is read in at a time to calculate the sum.

Scaling is an optional feature that uses linear scaling with a maximum and minimum value (normally 0 and 255), as Equation [4] shows.

$$x = \frac{x - min}{max - min} \times max \quad [4]$$

Scaling is implemented as part of the projection with a two-pass method. Both passes go through Algorithm 2 once. For band i, the first pass calculates the projection result to record the min and max values. The second pass scales the projection result using the min, max values and outputs the final result of band i. Therefore this two-pass scaling method requires higher complexity than $O(n)$.

3. Optimization for modern architecture with multi-core CPU

3.1 High performance computing with multi-core CPU

While GRASS has a straightforward implementation of the EVD-based PCA methodology, much of the effort has focused on minimizing the amount of memory copy and I/O overhead as shown in the previous section with the row-buffer mechanism and loop structure. Its performance still drops dramatically as the number of bands or image dimension increases. The original GRASS implementation does not take into consideration the cache based memory hierarchy (Van der Pas et al 2002) and the multi-core architecture (Kevin et al 2008) (Buttari et al 2007).

Figure 5 is a typical memory hierarchy of current computer architecture. Note that as the size of each level increases by 10^3 from on-chip register to disk storage, the latency grows exponentially. Without careful arrangement of the data access pattern, CPU could be stalled most of the time as it waits for data from far locations.

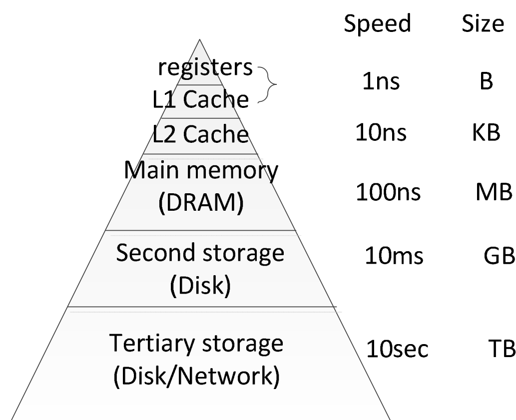

Figure 5: Memory hierarchy

Multi-core CPU is another feature that offers opportunity for performance improvement. Before the introduction of multi-core, the CPU speed is increased by raising clock cycle and having deeper pipelines, and this has hit a few limits such as power consumption. One of the solutions is through increasing the number of processing units (aka ‘cores’) on a single processor die and keeping a relative moderate clock cycle. Ultimately, higher performance can be achieved with less power consumption. Most of the CPUs used today have two to eight cores. Figure 6 is the die of an Intel core 2 i7 (Nehalem) processor which has 4 physical core and can appear as eight-core when use hyper-threading.

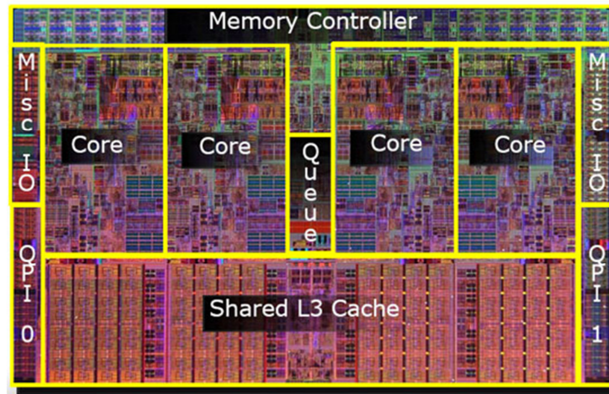

Figure 6: Intel Core2 i7 die

In the realm of high performance computing, linear algebra has enjoyed a rich development since the single-core age. For single node machine, BLAS (Basic Linear Algebra Subprograms)(Lawson et al 1979) is the de facto standard for basic linear algebra operations, and LAPACK (Linear Algebra Package) (Angerson et al 1990), which is based on BLAS, provides routines for solving higher level problems like systems of linear equations, eigenvalue problems, etc. ScaLAPACK (Scalable LAPACK) (Choi et al 1992) is the distributed version of LAPACK that targets cluster systems. The basic idea behind these libraries from single-core age is to fully take advantage of the memory hierarchy by using block algorithms that splits matrices into small blocks that can be kept in higher levels of memory like L1, L2 cache in the hierarchy and reuse them excessively. In the multi-core age, threaded version of BLAS and LAPACK like GotoBLAS (Goto et al 2008), MKL(Math Kernel Library) (MKL product brief) and PLASMA (Agullo et al 2009) provides higher performance by distributing the computing task to different cores. And PLASMA also makes used of dynamic scheduling to break the fork-join parallel model of LAPACK for even higher performance.

The PCA of GRASS only has data reuse at the level of main memory and secondary storage where an image is copied from files to main memory and then to the internal buffer where it is used for computation. From the above direction, better performance is possible if PCA is implemented by casting most of the computations to linear algebra operations and utilizing parallelism from the multi-core architecture. This section covers the optimization with regard to memory hierarchy. The multi-core related work will be discussed in the next section together with optimization for the GPU since it is an extension of work done for multi-core CPU implementations.

3.2 Casting PCA to BLAS operation

The programming paradigm shift to multi-core configurations makes it challenging to achieve performance improvements. For the GRASS PCA algorithm, many components are naturally parallelizable, like the mean value calculation and scaling, while other parts, like covariance matrix creation and projection, require different data manipulation techniques. We pursue the performance optimization by casting the bulk computation of PCA to level 3 BLAS calls which are directed to

high performance implementations and optimize from there.

Now we re-design the PCA algorithm. The covariance matrix creation is changed from Equation [2] to the following:

$$\boldsymbol{C} = \frac{1}{n}\sum_{i=1}^{n}(\vec{x} - \bar{x})\,(\vec{x} - \bar{x})^T = \frac{1}{n}\sum_{i=1}^{n}\vec{z}\,\vec{z}^T$$

$$= \frac{1}{n}(\vec{z_1}\vec{z_1}^T + \vec{z_2}\vec{z_2}^T + \cdots + \vec{z_n}\vec{z_n}^T)$$

$$= \frac{1}{n}[\vec{z_1}, \vec{z_2}, \cdots, \vec{z_n}][\vec{z_1}, \vec{z_2}, \cdots, \vec{z_n}]^T$$

$$= \frac{1}{n}\boldsymbol{Z}\boldsymbol{Z}^T \qquad [5]$$

Equation [5] is the symmetric rank-k update ('SYRK' in short from BLAS) in dense linear algebra, whose standard format is:

$$\boldsymbol{C} \leftarrow \alpha\boldsymbol{A}\boldsymbol{A}^T + \beta\boldsymbol{C} \qquad [6]$$

where A is of size $n \times k$ and $\boldsymbol{C}$ is a $n \times n$ symmetric matrix.

The operation count for SYRK is $nk(n + 1)$.

Similarly, the projection algorithm can be factored into matrix-matrix multiplication (GEMM from BLAS) with the operation count $2mkn$. SYRK is further optimized in the GPU section, and the details can be found in section 4.

We use GotoBLAS for our implementation of SRYK and GEMM. GotoBLAS differentiates GEMM through a layered approached and optimizing the block size of each layer so that the data movement between levels in the memory hierarchy is minimized. This way cache and TLB miss are small so that the CPU is kept busy computing as much as possible.

For data to be used by BLAS routines, the row-buffer mechanism is replaced with a larger buffer where the whole image data is held. The image cube is sliced by band and column direction as shown in Figure 7 where each band in the flattened data is represented by its corresponding color from the image cube. The dataset is stored in GRASS in the cube shape on the left, and during covariance matrix computation and projection, data is flattened to a separate buffer in memory in the shape on the right. This buffer is the matrix $\boldsymbol{Z}$ in Equation [5].

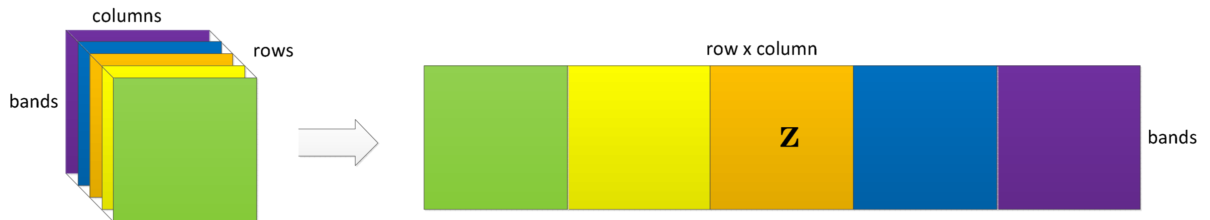

Figure 7: Data Flattening

Since almost all BLAS packages require column major and GRASS stores 2D image in row major, there is a flipping process involved in the flattening procedure which has a negative impact on performance. As shown in Figure 8, during the flipping, data is fetched from disk or main memory by get_map_row(…) to the green row. Here a NULL values are checked for by

Rast_is_null_value(…), and then each element is copied to the yellow region of the $\boldsymbol{Z}$ buffer in Figure 8. The $\boldsymbol{Z}$ buffer is non-contiguously accessed in memory. This access pattern triggers a high cache miss and TLB (Translation Lookaside Buffers) miss rate because the large stride (224 for example) makes elements in the same (green) row fall on different virtual memory pages in the yellow region. Accessing these elements frequently flushes the cache due to associativity that keeps mapping elements in the row to the same cache lines; this makes the write-back cache-strategy (Hennessy et al 2003) nearly behave in a write-through fashion. During a TLB miss, CPU is completely stalled and on some architectures CPU pipeline is flushed to handle the virtual address translation interrupt. A TLB miss has a typical penalty of 10-30 clock cycles while a TLB hit takes 0.5-1 clock cycles (Translation_lookaside_buffer wiki).

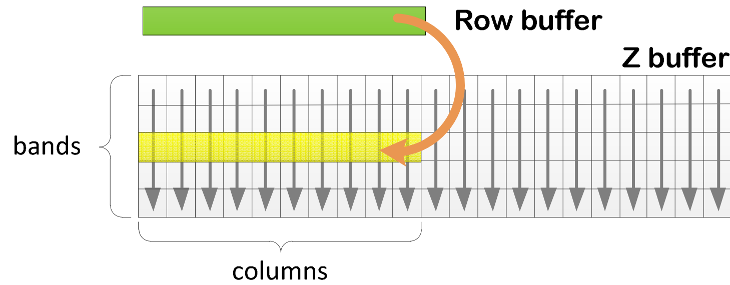

Figure 8: Non-contiguous memory issue

Table 1 shows the impact of TLB misses in different data access patters where the code snippet puts 1s into a column major 1D array of size $224(bands) \times 512(row) \times 512(columns)$. This is a similar operation to the data assembly to the $\boldsymbol{Z}$ buffer. The dimension is chosen to match the experiment dataset that we use in this work, and case 2 is the same as the flipping procedure. It is clear that by removing the TLB misses, runtime of the flipping could be cut 50% boosting the performance of the PCA implementation. This is especially true for the GPU code discussed in the next section.

Table 1. TLB miss overhead

case	1	2	3

Code snippet	for (j=0; j<512*512; j++)
	for (i=0; i<224; i++)
	p[i+j*224] = 1;
		for (k=0; k<512; k++)
		for (i=0; i<224; i++)
		for (j=0; j<512; j++)
		p[k*224*512+i+j*224] = 1;
			for (i=0; i<224; i++)
			for (j=0; j<512*512; j++)
			p[i+j*224] = 1;

TLB Miss	Low	Medium	High
Time(s)	0.452	1.109554	6.179042

So far there is no solution to this row-column major issue because GRASS does not provide APIs to access the map data by column. Possible speed-up could be obtained by using OpenMP (http://openmp.org/wp/) to parallelize the memory copy, even though this has a potential of bottleneck of saturating memory or I/O bandwidth.

3.3 Other routines

As discussed in 2.4, scaling and mean value calculation are both $O(n)$ operations; attempts to parallelize these two routines could be hampered by memory bandwidth saturation. Currently Rast_get_row() is not thread safe when different rows in the same map are accessed concurrently, and this makes it hard to apply OpenMP optimizations. Threading libraries like Pthread can be used to parallelize these routines.

4. Optimization for GPU

In section 3, PCA has been tuned based on memory hierarchy and the performance has been greatly improved. One issue with the method is the memory usage. GRASS assumes that map data is too large to fit in the CPU memory, while our CPU implementation decreases runtime at the expense of space assuming that main memory nowadays is less expensive and easily added. Another issue is that a multi-core CPU has been the trend and but the amount of parallelism is still confined by the number of cores in most off-the-shelf multi-core CPUs. In this section, the optimization is extended by using GPGPU (General purpose GPU), and these two issues are solved.

4.1 High performance computing with GPGPU

Since the release of CUDA (Compute Unified Device Architecture) (*NVIDIA CUDA™, Programming Guide*) by NVIDIA in 2008 which is the computing engine for NVIDIA GPUs, developers have been offered an easier way to access the massive computing power of GPUs through C interface in a SIMT (Single Instruction Multiple Threads) programming model. There are also other interfaces to program GPU. For example, OpenCL (http://www.khronos.org/opencl/) provides portability across different GPU vendors, and DirectCompute (DirectX Software Development Kit) which exclusively runs on the Windows platform allows accessing GPUs through DirectX.

Figure 9 shows the architecture of CUDA.

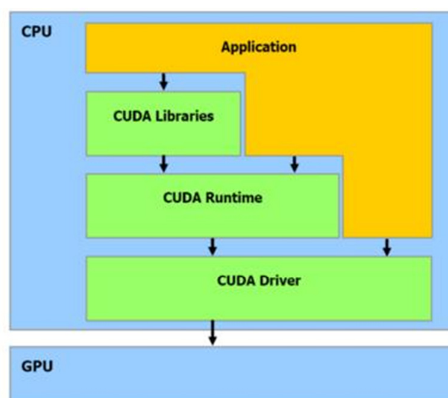

Figure 9: CUDA architecture [24]

The high performance of the GPGPU comes from its simpler circuit logics which are designed specifically for floating point operations, whereas the CPU targets a wider range of customers thus having more overhead due to its more general circuit logic design. Another source of computing

power of the GPU is that it has more computer units (aka ‘cores’) than the CPU. Table 2 compares the Intel Q6600 CPU and the NVIDIA GTX 470.

Table 2. CPU vs GPU

	Intel Q6600	NVIDIA GTX 470
Number of cores	4	448
Core/Shader clock	3.2 GHz	1.215 MHz
Peak performance (double precision)	51.2 Gflop/s*	149 Gflop/s

(*Gflop/s = billions of floating point operations per second)

The card we use for our experiment is the new generation GeForce GTX 400 from Nvidia, code named ‘Fermi’ (Fermi White Paper). Among the many new features this card introduced that affect our work over the previous GT200 series are higher number of cores, higher double precision performance, and increase in the size of shared memory on the card per SM (streaming Multiprocessor) from 16KB to 48KB.

To help achieve higher performance in linear algebra operations, NVIDIA has provided CUBLAS (Compute Unified Basic Linear Algebra Subprograms) which is the BLAS packages on NVIDIA GPU. We make use of CUBLAS routines and seek opportunities for further optimization by re-implementing the GPU kernels.

4.2 Interfacing CUDA with GRASS

GRASS is compiled using the C compiler while CUDA uses the C++ compiler for the host code and the kernel compiler called *nvcc* for GPU kernel code. To integrate CUDA functions to GRASS, both host and GPU codes are compiled into a static library and linked to the PCA code along with the CUBLAS library.

CUDA requires data to reside in the GPU’s main memory. We move and keep the data on the GPU until the very end of the PCA work because the overhead of data transferring, in terms of runtime is non-negligible. The only exception is for the eigenvalue decomposition where the size of the covariance is relatively small (224 x 224 in our experiment), and this computation is carried out on CPU by the eigenvalue decomposition routines from the original PCA code of GRASS.

The workflow of our GPU implementation of PCA is shown in Figure 10.

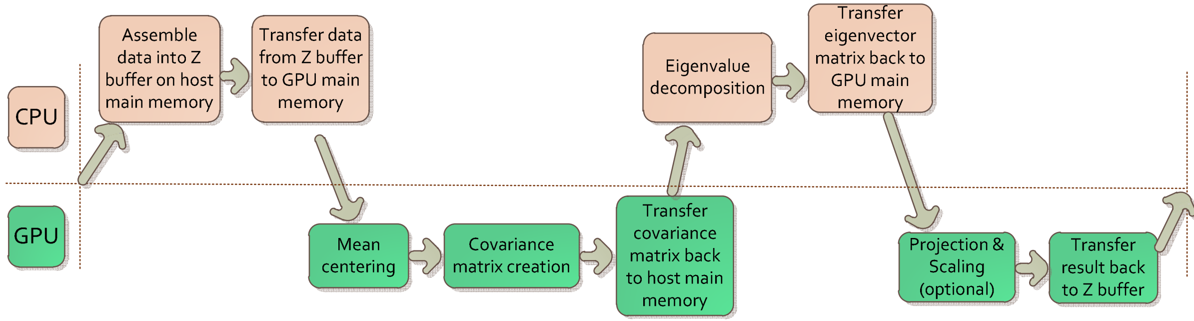

Figure 10: GPU workflow for PCA

Because most of the computation is carried out on GPU with the complete dataset after transferring data to GPU main memory, the host can release the Z buffer for other uses. If the data is too large to fit in the main memory of the CPU and GPU, data can be sliced as shown in Figure 7 to a certain number of slices per group. Each group is sent through the workflow in Figure10 and processed in sequence to avoid memory overflow.

4.3 Optimization to rank-k update

The covariance matrix creation and projection are both level-3 BLAS operations and share similar characteristics of having a relatively low ratio of computation to data size. For example, rank-k update as shown in Equation [6], which is at the heart of covariance matrix creation, in general has total flops (floating point operations) of $kn(n+1)$ where the matrix is of size n rows, k columns. Therefore, the larger n is, the more data reuse and the higher the performance. However, for the covariance matrix creation in PCA, n is the number of bands and $k = rows \times columns$. This makes n much smaller than k, and thus making it harder to exploit parallelism provided on the GPU.

4.4 Optimization based on Volkov's fast GEMM

The first direction of optimizing SYRK is to base the code on Volkov's fast GEMM (Volkov et al 2008). This implementation splits data into $\frac{M}{64} \times \frac{N}{16}$ thread blocks, where M and N are rows and columns of matrix $\boldsymbol{A}$ and $\boldsymbol{B}$ in $\boldsymbol{C} = \boldsymbol{C} + \boldsymbol{A} \times \boldsymbol{B}$. Each thread block uses 128 threads. The design objective of this algorithm is to keep blocks of $\boldsymbol{A}$ and $\boldsymbol{C}$ in registers and blocks of $\boldsymbol{A}$ in shared memory so that faster storage is used with the following priority ordering: register, shared memory and global memory.

We implemented SYRK based on Volkov's GEMM. Figure 11 shows the strip mining flow.

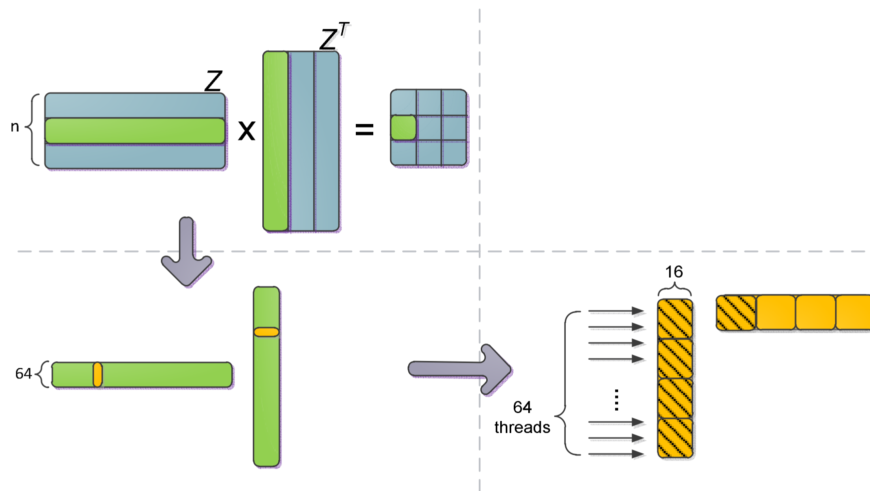

Figure 11: Strip mining of SYRK based on Volkov's GEMM

Same color represents the same section, and large arrows show the narrowing down from the entire problem size to the block of data that is solved by one thread block, denoted by shadows. Since the number of rows n is small, more parallelism can be achieved by splitting the SYRK into $\left(\frac{M}{64}\right)^2$ number of smaller SYRK each solved by one thread block of 64 threads.

The green curve in Figures 13 shows the testing result of this version in double precision. The performance is actually 20 Gflop/s away from the CUBLAS version, which was designed for early generation of NVIDIA GPUs. To gain insight about the run, NVIDIA provides CUDA-profiler (NVIDIA Compute Visual Profiler Version 3.1) which keeps track of multiple counters that are closely related to performance. Table 3 is the profiler output for this SYRK kernel.

Table 3.Occupancy Analysis of the Volkov's GEMM based SYRK (n=128, k=10240)

Grid size	Block size	Register ratio	Shared memory ratio	Active Blocks per SM	Active threads per SM	occupancy
2 × 8	16 × 4	1 (32768 / 32768)	0.416667 (20480 / 49152)	8 : 8	512 : 1536	0.333333 (16 / 48)

One of the most important measurements of kernel running is called the multiprocessor occupancy. The multiprocessor occupancy is the ratio of active warps to the maximum number of warps supported on a multiprocessor of the GPU. High occupancy normally translates to higher performance because it means GPU cores are busier solving problems in parallel. Exceptions include bandwidth bound applications with which higher occupancy leads to communication contention and performance drop. On each GPU, cores are grouped onto SMs (Streaming Processors) and each SM has a limited amount of computing resources like registers and shared memory. The number of thread blocks that could run simultaneously on each SMs is determined by the resource consumption of each thread block. In the case of the SYRK kernel, each thread

requires 63 registers, and each thread block uses 2.5KB shared memory. On older generation NVIDIA cards like GTX 280, this could cause a register contention because only 16384 registers are available on each SM. This only allows $16384/64/63 \approx 4$ thread blocks whereas GTX470 provides 32768 registers which increases the offer to 8 thread blocks.

From the profiler, the maximum amount of blocks is already fulfilled and yet the performance is still limited by low occupancy. The reason is that each thread block has 64 threads running, which totals 512 threads active threads per SM. Some GPU's architectures, like the GTX470, allow for higher thread count per thread block (e.g., 1024 for GTX470) which permits even higher performance improvements.

4.5 Square-split Optimization

By analyzing the profiler output of CUBLAS DSYRK, we propose a different approach as shown in Figure 12. The data is split by rows with a block size 32 (16 in DSYRK of CUBLAS), and the computation of rank-k update is expressed as in Equation [7]:

$$\boldsymbol{Z}_i \times \boldsymbol{Z}_i^T = [Z_{i,1}, Z_{i,2}, \cdots, Z_{i,m}] \times [Z_{i,1}, Z_{i,2}, \cdots, Z_{i,m}]^T = \sum_{j=1}^{m} (Z_{ij} \times {Z_{ij}}^T) \qquad [7]$$

Here $\boldsymbol{Z}_i$ is the green strip and Z_{ij} is the yellow block in Figure 12.

The output $n \times n$ square matrix is split by a square blocks of size 32 by 32 (hence called 'square-split'), and each thread block is in charge of producing the result for one of these blocks.

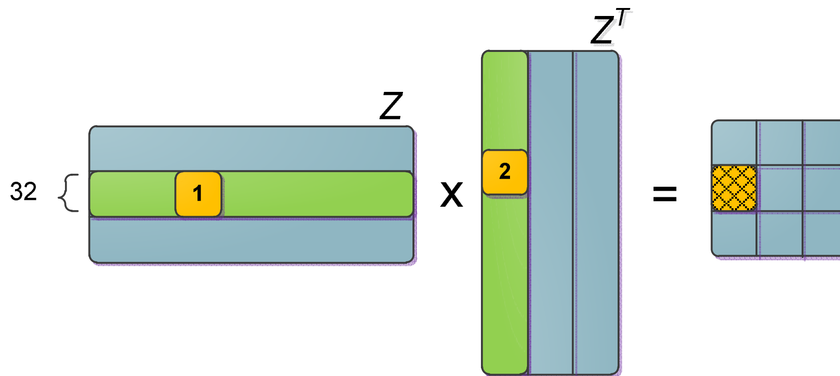

Figure 12: Strip mining of SYRK based on Volkov's code

During computation, each thread block has of $32 \times 32 = 1024$ threads. The two yellow blocks in Figure 12 are loaded into shared memory, consuming 16KB shared memory. Since both yellow blocks in share memory store data in row major and each thread in one block reads one element for both blocks, bank conflicts need to be avoided with array skewing shown in Figure14. CUDA shared memory is divided into equally-sized memory modules called memory banks that can be accessed concurrently. For GTX 400 series GPUs, shared memory has 32 banks that hold a successive 32-bit value (float, for instance). Bank conflicts occur when multiple threads request data from the same bank, and as a result, all the requests are serialized. This has a substantially negative impact on performance. With row major 2D array, array skewing works by allocating one extra column (32 rows by 33 columns in our case) in shared memory, and this shifts the data in the

same column to different banks thus reducing bank conflict.

At each step j of Equation [7], each thread in the thread block computes one element in the result block by accessing a row in yellow block 1 and column in block 2. With older generation NVIDIA cards, the data block size has to drop from 32 to 16, which has performance close to that of DSYRK in CUBLAS. Table 4 shows the Occupancy analysis of this algorithm. The new kernel consumes 24 registers in each thread. Even though the number of active blocks per SM dropped from 8 to 1, the active threads per SM is twice as many as in the previous optimization attempt, and occupancy is doubled. In Figure 13, the purple line is the performance of this version, and comparing to the previous version, a doubling in occupancy has led to a doubling of performance and nearly a 20% performance increase to DSYRK in CUBLAS when n is small. Note that the code based on Volkov's GEMM has higher performance than both the square-split and CUBLAS version when n is large enough. For the best performance, a switch can be inserted to select the best code according to the shape of matrix Z.

Table 4. Occupancy Analysis of the Square-split SYRK (n=128, k=10240)

Grid size	Block size	Register ratio	Shared memory ratio	Active Blocks per SM	Active threads per SM	occupancy
4×4	32×32	0.75 (24576 / 32768)	0.354167 (17408 / 49152)	1 : 8	1024 : 1536	0.666667 (32 / 48)

The tuning of GEMM for the projection procedure is similar to that of SYRK and is omitted in this text. In our implementation, we use the auto-tuned DGEMM (GEMM in double precision) from the MAGMA package.

4.6 Optimization for mean value calculation and scaling

Mean value and scaling are embarrassingly parallelizable which makes them perfect operations for GPU speed ups.

Mean value routines are threaded by having a 1D thread block where each thread calculates the mean value of one row of matrix $\mathbf{Z}$. For $n < 1024$, only one thread block is necessary. The result is stored on a 1D buffer in GPU main memory.

Like the original code and CPU optimization, scaling on the GPU also uses a two-pass method with one kernel for each pass. The first kernel is similar to the mean value GPU kernel but computes the maximum and minimum values and stores in these in a 1D buffer on GPU main memory, and the second kernel scales each element by having as many threads as elements using the max/min result produced by step one. Each thread scales its assigned element with Equation [4].

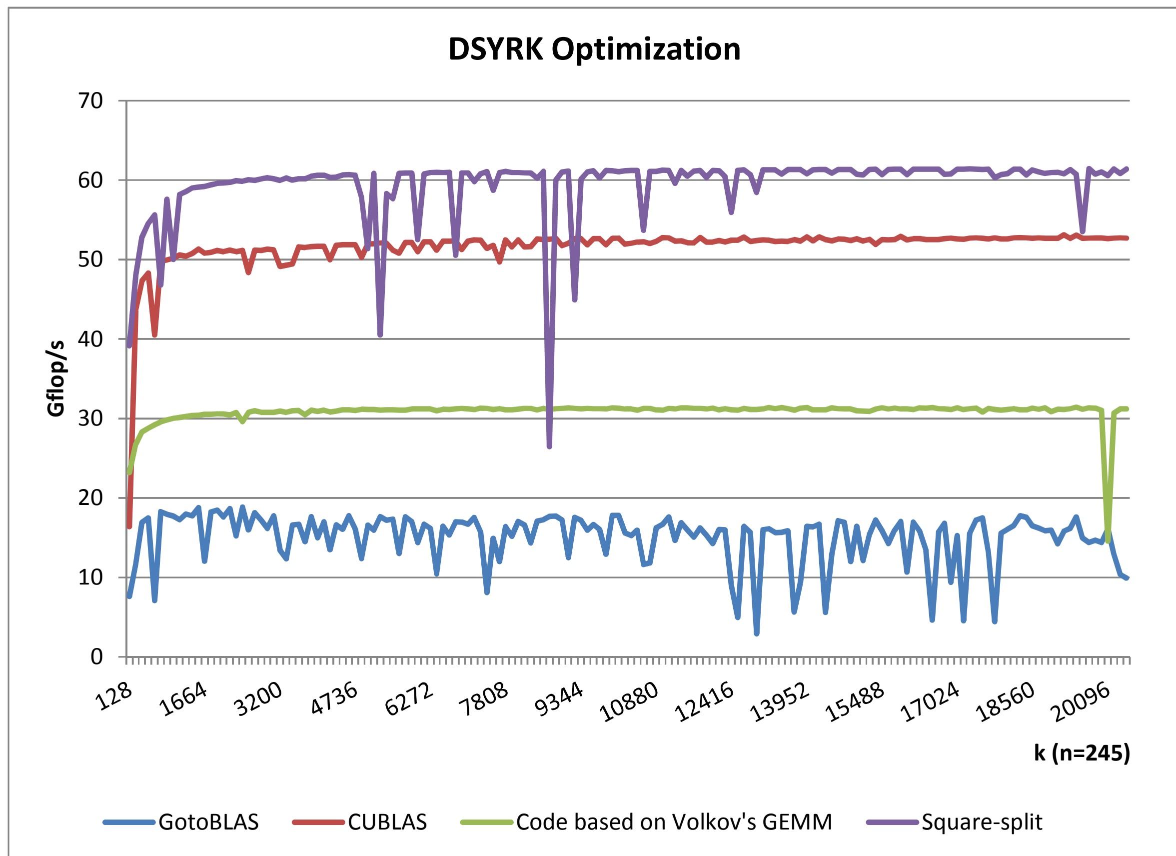

Figure 13: Performance of different optimizations to SYRK

4.7 Data transferring

The last issue about using the GPU is the need to transfer the image data from the host's main memory to the GPU's main memory because this time is counted as part of runtime. We use cudaMemcpy() to do the transfer and further optimization includes using page-locked memory which makes allocating device memory and explicitly transferring data between device and host memory no longer necessary. This way data copies are implicitly performed every time when kernels access the mapped memory.

5. Experiment and performance analysis

The experiment consists of running and comparing three different implementations of the PCA in GRASS, including the original, optimization for memory hierarchy and multi-core CPU, and a hybrid optimization (CPU and GPU). We evaluate the performance impact of our improved algorithm by running the experiment on a machine with the hardware configuration described in Table 5.

Table 5.Experiment Platform and Resources

<table>
<tr><th colspan="2">Host</th><th colspan="2">GPU</th><th colspan="2">Data</th><th>GRASS</th></tr>
<tr><td>CPU</td><td>Intel Q6600 @ 3.2GHz Quad Core</td><td>Device</td><td>GeForce GTX 470</td><td>Device</td><td>AVIRIS reflection</td><td rowspan="4">7.0 from SVN</td></tr>
<tr><td rowspan="2">Memory</td><td rowspan="2">8GB DDR3</td><td rowspan="2">Number of cores</td><td rowspan="2">448</td><td>Data dimension</td><td>512 rows
512 columns
224 bands</td></tr>
<tr><td>interleave</td><td>Bsq</td></tr>
<tr><td>Hard Disk</td><td>Western Digital Raptor WD1500ADFD</td><td>global memory</td><td>1280MB</td><td>Data type</td><td>Double</td></tr>
</table>

5.1 Correctness

Figure 14 shows the image comparison, using the first principal component, between the original GRASS and our two optimized implementations. All three implementations have the same result in terms of correctness. The rest of the bands were also compared to make sure the results are equal across implementations.

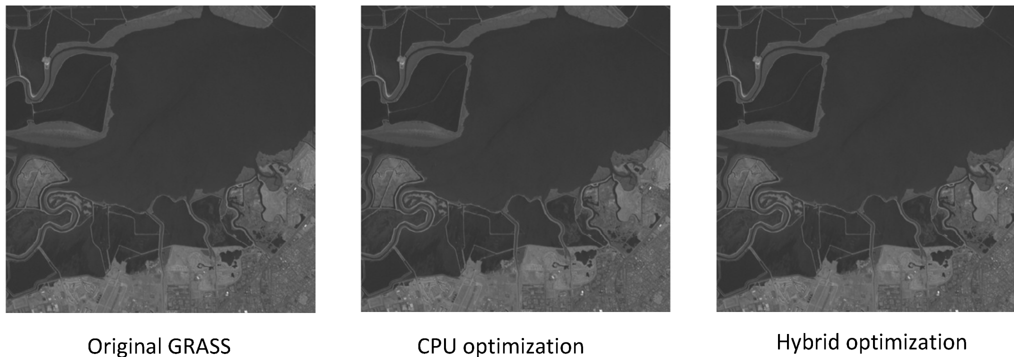

Figure 14: 1st principal component of the PCA result

5.2 Main components

Mean value calculation and centering, covariance, and scaling are the main pieces of PCA in the sense that they take up most of the runtime. Table 5 compares the performance of these 4 components between different implementations.

Table 5.Main component performance comparison (unit: sec)

	Original	Multi-core CPU	Hybrid
Mean Value Calculation	1.719453	1.675	0.120235
Mean centering+Covariance Matrix Creation	277.9111	2.562	0.173531
Projection+Scaling	842.036	6.940708	0.429601

5.3 GPU performance analysis

For the Hybrid implementation of PCA, there is an extra overhead of assembling the Z matrix and the data transferring between host and GPU. Figure 15 is the decomposing of the time spent in each part of the Hybrid PCA.

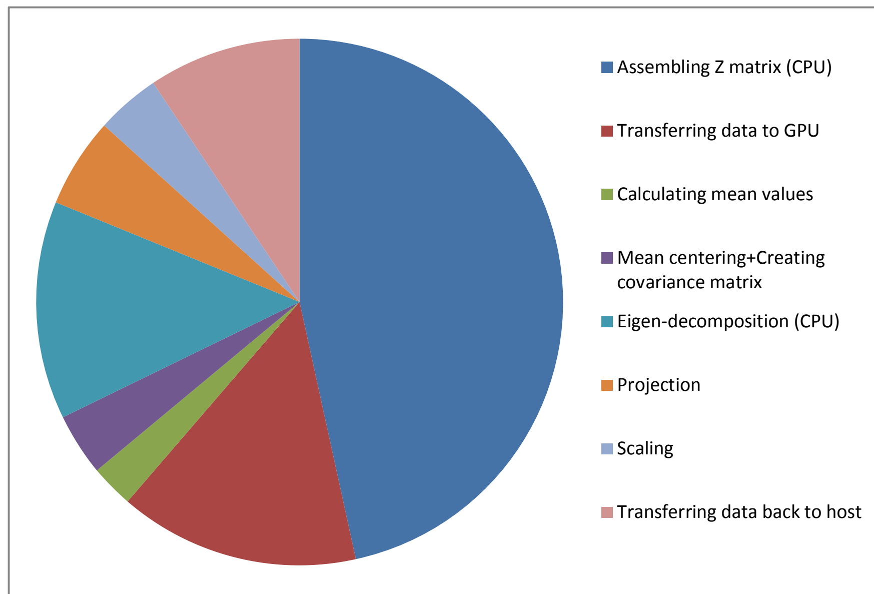

Figure 15: GPU PCA runtime decomposition

The runtime of computational kernels mean values, covariance matrix, projection and scaling is small comparing to that of Z matrix assembling, data transferring and the eigenvalue decomposition. Z matrix assembling on host using CPU is the most time consuming routines. This is attributed to the same issue of column/row major flipping in section 3.2.

5.4 Overall runtime

Figure 16 is the overall runtime comparison of the three different implementations of PCA. By optimizing the multi-core CPU with memory hierarchy and GPU, substantial speed-ups have been achieved.

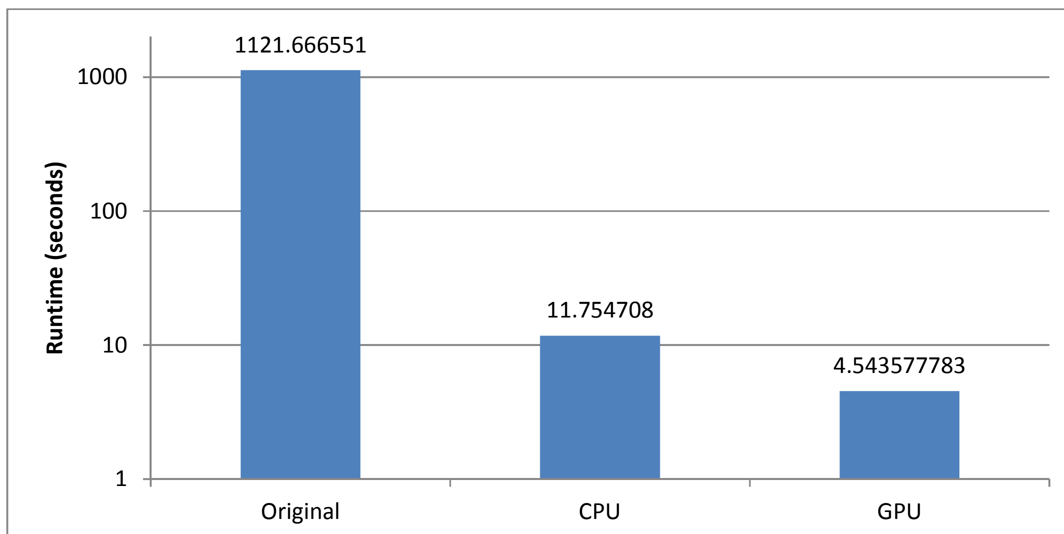

Figure 16: Overall runtime

6. Conclusions

Our optimized PCA implementations for the GRASS GIS have substantially lower runtimes over the original implementation. The dramatic decrease in runtime using imaging spectrometry data from AVISIS demonstrates the potential performance gained by leveraging multi-core and GPU technologies. As hardware advancements continue to move to larger number of cores and more advanced GPU's, the methodology presented in this paper will be imperative for GRASS GIS to take advance of to continue performance enhancements and user enrichment.

7. Acknowledgement

The authors would like to thank Rick Weber from the university of Tennessee, Knoxville for valuable discussion and help on the text.

References

[1] Zhang, B. et al 2006, 'A patch-based image classification by integrating hyperspectral data with GIS', *International Journal of Remote Sensing*, Vol. 27, No. 15 August 2006 , pp. 3337 – 3346

[2] Remote Sensing for Coastal Management
< http://www.csc.noaa.gov/crs/rs_apps/sensors/hyperspec.htm>

[3] Richards, J.A. and Jia, X. 2006, *Remote Sensing Digital Image Analysis: An Introduction*, Springer Verlag

[4] Andrecut, M. 2009, 'Parallel GPU Implementation of Iterative PCA Algorithms', *Journal of Computational Biology*, Vol.16, No. 11, pp. 1593-1599

[5] Wold, S. & Esbensen, K. & Geladi, P. 1987 'Principal Component Analysis', *Chemometrics and Intelligent Laboratory Systems*, Vol. 2, Issues 1-3, pp. 37-52

[6] Kramer, R. 1998 *Chemometric Techniques for Quantitative Analysis*, CRC Press, New York.

[7] Castaño-Díez D. et al 2008, ‘Performance evaluation of image processing algorithms on the GPU’, *Journal of structural biology*, Vol.164, No.1, pp.153-160
[8] NVIDIA, *CUDA CUBLAS Library*,
<http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/CUBLAS_Library_3.1.pdf>
[9] < http://forums.nvidia.com/index.php?showtopic=157531>
[10] < http://trac.osgeo.org/grass/wiki/DownloadSource>
[11] Jackson, J. 2003, *A User's Guide to Principal Components*, Wiley-IEEE.
[12] Van der Pas, R. and Performance, H. 2002, ‘Memory Hierarchy in Cache-Based Systems’, *Sun Blueprints*, pp. 26. 817-0742-10, Santa Clara, California: Sun Microsystems
[13] Kevin J. Barker et al 2008, ‘A performance evaluation of the Nehalem quad-core processor for scientific computing’, *Parallel Processing Letters*, Vol. 18, No.4, pp. 453-469
[14] Buttari, A. and Dongarra et al 2007, Parallel and Distributed Processing and Applications, ‘The Impact of Multicore on Math Software’, Springer-Verlag Berlin, Heidelberg
[15] Lawson, CL and Hanson, RJ and Kincaid, DR and Krogh, FT 1979, ‘Basic Linear Algebra Subprograms for FORTRAN usage’, *ACM Transactions on Mathematical Software*, Vol.5, No.3, pp. 308-323
[16] Angerson, B. et al 1990, ‘LAPACK: A Portable Linear Algebra Library for High-Performance Computers’, *IEEE Comput. Soc. Press & UTCSA report UT-CS-90-105*
[17] Choi, J. and Dongarra, J. and Pozo, R. and Walker, D.W. 1992, ‘ScaLAPACK: A Scalable Linear Algebra for Distributed Memory Concurrent Computers’, *Proceedings of Fourth Symposium on the Frontiers of Massively Parallel Computation (McLean, Virginia). IEEE Computer Society Press, Los Alamitos, CA and UTCS report UT-CS-92-181*
[18] Goto, K. and Geijn, R.A. 2008, ‘Anatomy of high-performance matrix multiplication’, *ACM Transactions on Mathematical Software*, Vol.34, No.3, Article No.12, pp.1-25
[19] _Intel® Math Kernel Library 10.2 for Windows*, Linux*, and Mac OS* X_ 2010, product brief, Intel, <http://software.intel.com/sites/products/collateral/hpc/mkl/mkl_brief.pdf>
[20] Agullo, E. and Demmel, J. and Dongarra et al 2009, ‘Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects’, *Journal of Physics: Conference Series*, vol.180, no.1, pp.012037
[21] Hennessy, J.L. and Patterson, D.A. and Goldberg, D. 2003, *Computer architecture: a quantitative approach*, Morgan Kaufmann
[22] Translation_lookaside_buffer wiki
<http://en.wikipedia.org/wiki/Translation_lookaside_buffer>
[23] <http://openmp.org/wp/>
[24] NVIDIA, *NVIDIA CUDA™, Programming Guide*,
<http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf>
[25] <http://www.khronos.org/opencl/>
[26] *DirectX Software Development Kit*,
<http://www.microsoft.com/downloads/details.aspx?FamilyID=b66e14b8-8505-4b17-bf80-edb2df5abad4&displaylang=en>
[27] Whitepaper NVIDIA’s Next Generation CUDATM Compute Architecture: FermiTM,
<http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf>

[28] Volkov, V. and Demmel, J.W.2008, ‘Benchmarking GPUs to tune dense linear algebra’, *Conference on High Performance Networking and Computing, Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, Article No.31
[29] NVIDIA, *NVIDIA Compute Visual Profiler Version 3.1,* <http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/VisualProfiler/computeprof.html>