Plataforma Hibrida de Procesamiento Paralelo (PHPP)

Equipo_6 Adrian Vazquez - Ricardo Lastra

29 de mayo de 2017

- **Introducción**

Mediante el siguiente reporte mostraremos la aplicación de herramientas innovadoras de cómputo en paralelo y cómputo matricial, así como explicaremos una implementación de la factorización SVD a la vida real en el ramo de Seguros de Automóviles.

Describiremos un panorama innovador para la lectura y cómputo de imágenes, su descomposición y composición de una forma matricial a otra a través de un método de cómputo en paralelo implementado en una extensión del conocido lenguaje C.

- **Objetivo**

Nuestro objetivo es diseñar e implementar una plataforma hibrida basada en el procesamiento de GPU's para la ejecución en paralelo de la factorización SVD como sigue:

1. Implementar Plataforma Hibrida de procesamiento en Paralelo (PHPP)
2. Implementar la factorización de una matriz SVD en cómputo en paralelo con CUDA-C. Dentro de este objetivo buscaremos cubrir los siguientes objetivos particulares:
   a. Obtener los valores singulares de una imagen visualizada como una matriz.
   b. Lograr una la reconstrucción de una imagen a partir de valores singulares computados por CUDA-C y las funciones cuBLAS.

- **Problema a resolver**

En la actualidad las empresas de seguros tienen gastos considerables para el manejo de documentos digitales. La cantidad de información que se genera a partir de fotografías o documentos digitalizados derivados de la operación de seguros crece muy rápido. Por esta situación cada vez se hace más complejo el control de dicha información.

En una compañía de seguros de tamaño medio, se digitalizan más de 3,000 documentos diariamente, además de los documentos que se reciben a través de la red por parte de los proveedores, ajustadores y clientes mismos. Es por ello que integrar cada documento a un expediente digital se hace una tarea difícil.

Aunque existen ya herramientas que provee el mercado para el manejo y visualización de estos documentos, trabajar con imágenes de gran tamaño es todo un reto. Entonces se buscara resolver el problema de dimensión o tamaño de una imagen y una forma inteligente de almacenarla y visualizarla a través de las herramientas de computo en paralelo que existen hoy en día.

- **Motivación**

La motivación de realizar una implementación inteligente a nuestro problema se fundamenta por 2 cosas.

1. La gran potencia computacional y ancho de banda de memoria muy alta con la que se cuenta hoy en día a través de una GPU es increíble, gracias a NVIDIA y su capacidad de satisfacer al mercado de un insaciable "tiempo real" ha permitido que sus tarjetas gráficas a través del lenguaje CUDA-C conviertan el procesamiento de gráficos a un propósito general; entonces las aplicaciones pueden ser tantas nuestra mente pueda imaginar.

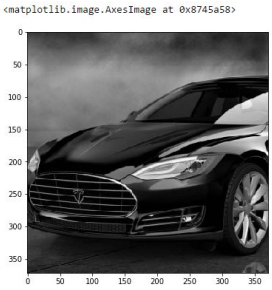
2. La gran tendencia al uso de programas OpenSource y la eficiencia de las rutinas ya establecidas en los diferentes lenguajes de programación, nos impulsa a generar modelos de rápido desarrollo y fácil aplicación. La simplicidad de lenguajes como R o Python nos permite computar operaciones básicas del algebra lineal y a través de extensiones del
lenguaje C podemos computar estas operaciones en paralelo, lo que hace esto aún más útil e interesante.

- **Software utilizado**
- CUDA-C 8.0
  - CuBlas 8.0.6
  - cusolverDn
  - cusolverDnDgesvd
- RStudio
  - R Markdown
  - mrbsizeR
- Python 3.5
  - matplotlib
  - numpy
  - PIL
  - csv

- **Datos**

Para nuestro problema utilizamos varias imagenes de muestra las cuales fueron cargadas a Python y con la libreria Pil obtuvimos el contenido de las imagenes como un objeto con cada valor de un pixel. Posteriormente estos valores fueron acomodados al orden de una matriz `A` de MxN.

La imagen que usamos fue la siguiente:



Su representación matricial es:

```
[[ 109.  107.  105. ...,   54.   58.   59.]
 [  62.   60.   60. ...,   50.   52.   54.]
 [  62.   60.   61. ...,   49.   52.   55.]
 ...,
 [  37.   36.   44. ...,   59.   52.   51.]
 [  42.   50.   36. ...,   68.   55.   49.]
 [  11.    0.    5. ...,   76.   55.   43.]]
```

- **Arquitectura**

La arquitectura propuesta al inicio de la investigación con Sun Grid Engine o MPI convertía a nuestro problema en un problema híbrido, ya que se trataban de diferentes componentes de arquitectura para resolver un mismo problema. Sin embargo durante la investigación nos percatamos que el uso de GPU local era muy eficiente.

Logramos configurar algunas máquinas de Amazon así como levantar algunos clusters con Docker para el uso de CUDA, sin embargo, la arquitectura final fue local para convertir el problema en algo más sencillo.

Se definió un pipeline local, usando Python para la lectura de imágenes y su transformación a forma matricial, se importaron estos valores con CUDA-C, computando la SVD con **cusolverDnDgesvd** para que posteriormente los resultados fueran leídos nuevamente por Python y a su vez interpretados.

De esta forma, dejamos la parte computacional pesada a la tarjeta grafica de NVIDIA y la parte sencilla a nuestro CPU.

- **Método**

Para nuestro problema usamos la factorización SVD o "Descomposicion de Valores Singulares".

La forma de SVD es: $A = U\Sigma V^T$

Donde: U es una matriz unitaria mxm (entonces $K = R$ las matrices unitarias son matrices ortogonales).

Σ es una matriz diagonal mxn con numeros reales no negativos en la diagonal.

V es una matriz unitaria nxn sobre K.

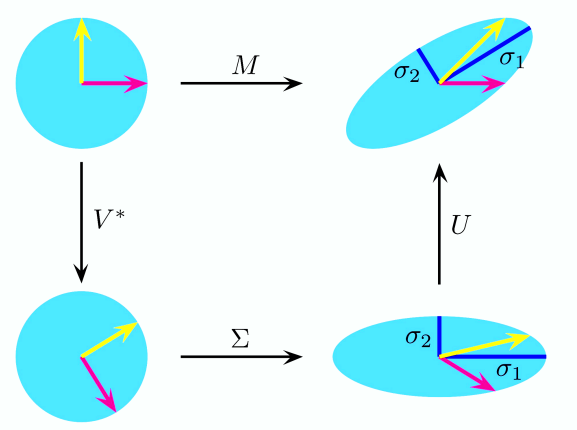
V^* es la matriz unitaria transpuesta ortogonal nxn de V.

El método numerico para calcular la SVD que usa nuestra libreria implementada **cusolverDnDgesvd** es el metodo `thin`.

Este método `thin` nos dice que necesitamos encontrar la matriz Vi ortogonal de nxn y una matriz Ui con columnas ortonormales de mxn tales que $Ui^T A - V = B$ sea Bidiagonal.

Las entradas diagonales $Sigmai$ de Σ son conocidos como los valores singulares de A, los Ui son los vectores singulares izquierdos de A, los Vi son los vectores singulares derechos de A

La siguiente imagen ilustra como suceden estas transformaciones:



$$M = U \cdot \Sigma \cdot V^*$$

- **Código**

La implementación de SVD en CUDA con **cuBLAS** y **cusolverDnDgesvd** fue lo más complejo ya que hay que entender los parametros de las librerias, la forma de alojamiento de los datos en memoria y lo mas importante los **inputs** y **outputs** que brindan las rutinas.

Al inicio de nuestra investigación pudimos correr prubas locales, comparando los resultados obtenidos a través de nuestro demo con CUDA y los mismos resultados con Python. De esta forma pudimos optimizar el tiempo para buscar soluciones en la arquitectura.

Al introducir una imagen real a nuestro programa, pudimos observar cosas importantes en esta implemetación.

a. Observamos la facilidad de reconstruccion de una imagen vector a vector teniendo ya los valores singulares.
b. Observamos que los calculos con la rutina **gesvd** solo soportaban matrices $m >= n$.
c. Tambien nos dimos cuenta que la rutina **gesvd** en su valor de ssalida de V nos devuielve unicamente V^T y no solo V.

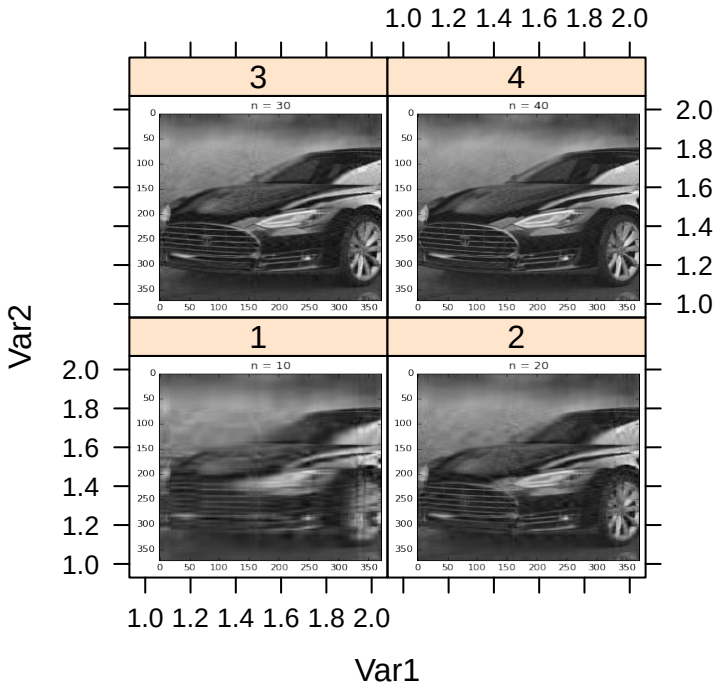
- **Resultados**

En la primera prueba pudimos validar los resultados de la siguiente forma:

```
=====
S = (matlab base-1)
S(1,1) = 8.360097
S(2,1) = 2.634734
S(3,1) = 0.408596
=====
U = (matlab base-1)
U(1,1) = -0.281380
U(1,2) = 0.214819
U(1,3) = 0.935242
U(2,1) = -0.834936
U(2,2) = 0.425575
U(2,3) = -0.348954
U(3,1) = -0.472977
U(3,2) = -0.879056
U(3,3) = 0.059612
=====
VT = (matlab base-1)
VT(1,1) = -0.546295
VT(1,2) = -0.623248
VT(1,3) = -0.559574
VT(2,1) = 0.060350
VT(2,2) = 0.637049
VT(2,3) = -0.768457
VT(3,1) = -0.835416
VT(3,2) = 0.453574
VT(3,3) = 0.310404
```

```
U:
[[-0.28138021  0.21481929  0.93524213]
 [-0.83493576  0.42557458 -0.34895351]
 [-0.47297723 -0.87905571  0.05961205]]
Sigma:
[ 8.36009713  2.63473436  0.40859623]
Vt:
[[-0.54629481 -0.62324831 -0.55957441]
 [ 0.06034999  0.63704934 -0.7684569 ]
 [-0.83541598  0.45357433  0.31040372]]
```

Entonces comparando los resultados obtenidos de una matriz mas grande con Python y con CUDA visualizando solo los numeros, no es lo optimo. Asi que en el pipeline despues de computar la SVD con **cusolverDnDgesvd** los resultados son regresados a Python para que puedan ser computadas las aproximaciones usando la primera columna de U y la primera fila de V reporduciendo la imagen, cada columna de pixeles es una ponderacion de los mismos valores originales $\overrightarrow{U}_1$. Con estas aproximaciones validamos que conforme tomamos mas vectores, la imagen se reconstruye con una mejor calidad visual. (Ver siguientes 4 imágenes)



- **Conclusiones**

Pudimos observar el poder de cómputo a través de la GPU y comprobamos que es muy viable su uso en producción, es decir, es posible implementar en un flujo de trabajo ya establecido algunos métodos numéricos programados en CUDA. Como principal ventaja de los modelos en paralelo es que las rutinas pueden ser llamadas múltiples veces, dependerá de la capacidad del programador la eficiencia del modelo.

Logramos al final un pipeline muy sencillo para resolver nuestro problema, evitando arquitecturas complejas y robustas. Pudimos comprobar a través de este pipeline la calidad de los resultados al computar **SVD** con **cusolverDnDgesvd**, ya que pudimos reconstruir la imagen original

Observamos tal como vimos en clase que la **SVD** existe para todas las matrices y con **cusolverDn** CUDA define los inputs de las matrices, es decir, CUDA permite alojar diferentes tipos de variables de entrada.

Así mismo llegamos a la conclusión que las áreas que puede abarcar aplicar SVD a imágenes son muy amplias, ya que podemos hacer un ensamble de modelos de Machine Learning para detección de patrones, para indexado automático, y para búsquedas rápidas de imágenes.

- **Por hacer**

Nos quedamos con la satisfacción del modelo para implementarlo, aunque la curva de aprendizaje fue difícil, no tardamos mucho tiempo en obtener resultados positivos de la factorización `SVD`. El comprobarlos con Python fue fácil, lo que nos despierta más entusiasmo para poder implementar modelos de Machine Learning e irlos cotejando como lo hicimos con `SVD`.

Pensamos que una versión mas nueva en `AWS` de las Amis de CUDA podrían ser una solución muy rentable a muchos problemas de computo numérico, la relación costo beneficio hoy en día es un equilibrio perfecto para considerar una EC2 en un producto de datos.

Así mismo se piensa desarrollar un proyecto interno de Seguros para implementar `SVD` en algunas áreas operativas que por cuestiones de privacidad no se darán muchos detalles.

- **Referencias**

http://math.nist.gov/~RPozo/

https://en.wikipedia.org/wiki/JAMA_(numerical_linear_algebra_library)

https://en.wikipedia.org/wiki/Singular_value_decomposition

ftp://ece.buap.mx/pub/profesor/academ48/Libros/TesisDavid.pdf

SVD en cuda: S. Lahabar, P. J. Narayanan. Singular Value Decomposition on GPU using CUDA

G. Golub, W. Kahan. Calculating the singular values and pseudo inverse of a matrix y el capítulo 8 del libro: G. H. Golub, C. F. Van Loan, Matrix Computations. John Hopkins University Press, 2013

Algebra Lineal de Mina S. de Carakushansky y guilherme de La Penha Editorial Ma Graw Hill