Tesis de licenciatura

Procesamiento rápido de EEG utilizando GPU

Federico Raimondo

Directores: Dr. Diego Fernández Slezak, Ing. Alejandro Furfaro

Co-director: Lic. Juan Kamienkowski

Julio de 2011

1. Resumen

La observación de señales analógicas es inherentemente ruidosa. *Independent Component Analysis* (ICA) surge como un método estadístico para la separación de fuentes (componentes independientes) que dan origen a las señales, técnica muy utilizada, por ejemplo, en la remoción de ruido y detección de artefactos en el estudio de electroencefalogramas (EEG).

Un problema central que posee este método es su complejidad computacional. Su implementación se basa en un método de gradiente ascendente cuya aplicación a un EEG estándar de 132 canales, a 512 muestras por segundo y una hora de duración puede alcanzar las 10 horas de cómputo en un procesador de última generación. Este procesamiento realiza, en un 90 % del tiempo, operaciones de punto flotante consecuencia de las multiplicaciones de matrices y vectores utilizando la biblioteca *BLAS*. *CUDA* surge hace algunos años como una arquitectura de cómputo de propósito general fuertemente orientada a las operaciones de punto flotante en paralelo. En el desarrollo de esta tesis se propone utilizar las ventajas de los GPU para acelerar el análisis de ICA.

Una primera aproximación para una optimización consiste en reemplazar el uso de la biblioteca *BLAS* por la biblioteca desarrollada por *Nvidia* (*CUBLAS*) y, de esta forma, aprovechar las características de la arquitectura *CUDA*. A pesar de los resultados que *Nvidia* utiliza para promocionar el uso de la biblioteca *CUBLAS*, en este caso las dimensiones de las matrices y vectores no aprovechan al máximo todos los recursos de las placas gráficas. Los tiempos arrojados por diversos experimentos mostraron que la utilización de *CUBLAS* no resulta eficiente y, dependiendo de la plataforma donde se realizan las mediciones, estas son mayores a las obtenidas utilizando CPU y la implementación ATLAS de *BLAS*.

Una segunda aproximación es implementar el algoritmo nuevamente, esta vez utilizando *CUDA*, sin hacer uso de bibliotecas. En consecuencia, se analizaron los puntos críticos y se determinó que, a pesar de que no existieran resultados prometedores con *CUBLAS*, se podían realizar optimizaciones y mejorar los tiempos. Aprovechando el conocimiento sobre las características de los datos a ser analizados, las operaciones sobre matrices y vectores realizadas y los detalles de la arquitectura *CUDA*, la segunda implementación del algoritmo mostró optimizaciones del orden del 4X para el análisis de EEG.

2. Introducción

Un problema central en el análisis de señales es la búsqueda de una representación o transformación adecuada que explique la señal observada. Este análisis estadístico multivariado para la separación de señales es un tema muy estudiado que reviste una alta complejidad debido al ruido, inherente a este tipo de señales. Muchos enfoques distintos han sido desarrollados con el fin de separar las señales generadas por las fuentes de estudio de aquellas que únicamente aportan ruido, por ejemplo PCA [Oja92], *factor analysis* [Har76], *projection pursuit* [Fri87], entre otros.

En los últimos años, también, ha surgido *Independent Component Analysis* (ICA) [BS95, Ama98] como un método efectivo para la separación de fuentes y remoción de ruido y artefactos que ha demostrado ser muy útil en diversos escenarios.

A modo de ejemplo, en la figura 1 se muestra la representación de 3 señales de audio, $s_1(t)$, $s_2(t)$ y $s_3(t)$ (imagen extraída de [HOK01]).

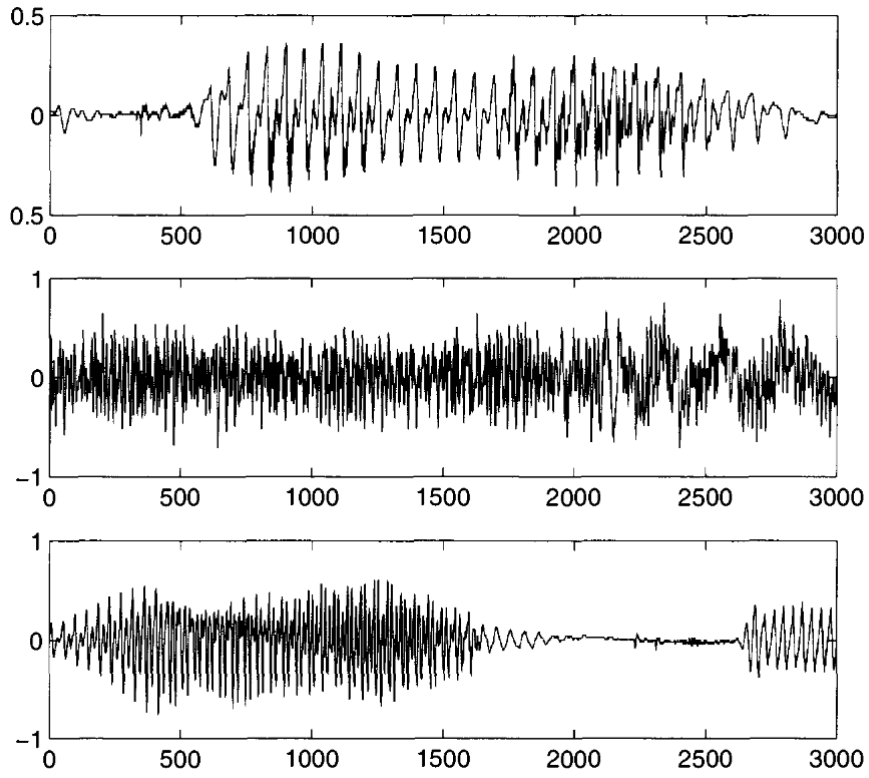


Figura 1: Grafico de 3 señales de audio independientes generadas en simultáneo.

Cada una de estas señales son generadas en simultáneo produciendo una señal resultante que proviene de la suma de estas tres ondas. Supongamos que se registra sonido mediante tres micrófonos obteniendo grabaciones de esta señal mixta, como se puede ver en la figura 2 (extraída de [HOK01]): $x_1(t)$, $x_2(t)$ y $x3(t)$.

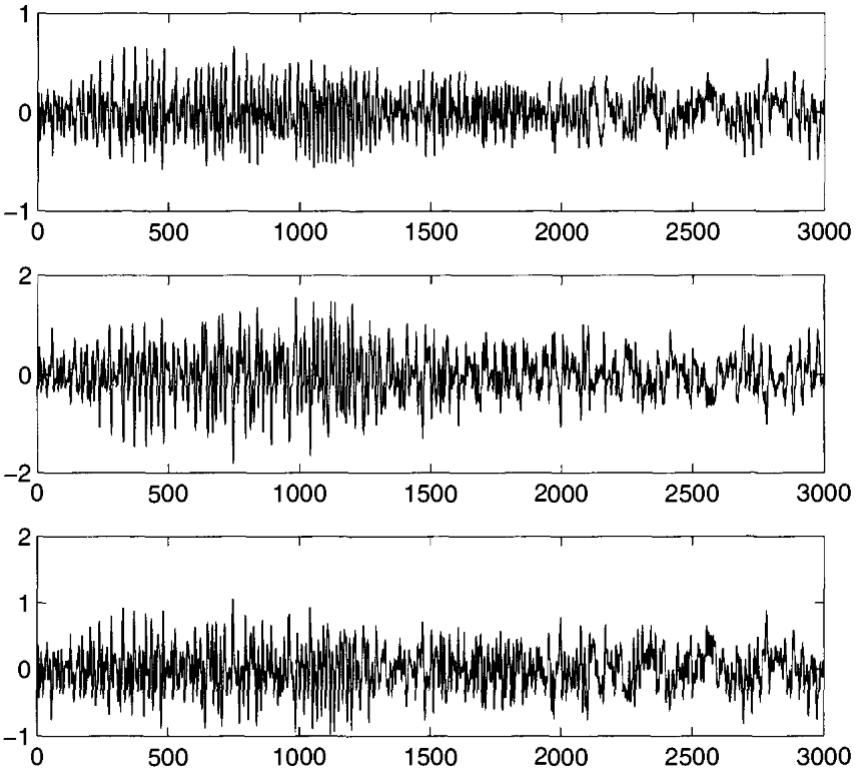


Figura 2: Gráfico de las tres señales de audio de la figura 1 registradas con tres micrófonos.

Cada una de estas señales de la figura 2 es la resultante de una suma ponderada de las 3 señales originales emitidas $s_1(t)$, $s_2(t)$ y $s_3(t)$:

$$\begin{aligned} x_1(t) &= a_{11}s_1(t) + a_{12}s_2(t) + a_{13}s_3(t) \\ x_2(t) &= a_{21}s_1(t) + a_{22}s_2(t) + a_{23}s_3(t) \\ x_3(t) &= a_{31}s_1(t) + a_{32}s_2(t) + a_{33}s_3(t) \end{aligned} \tag{1}$$

donde los a_{ij} son parámetros que dependen de la posición de los micrófonos con respecto a los generadores de las señales. El problema planteado, conocido como el *cocktail-party problem*, consiste en recuperar las señales originales $s_1(t)$, $s_2(t)$ y $s_3(t)$ a partir del registro de las señales $x_i(t)$.

Teniendo la información de los distintos a_{ij} se resolvería simplemente invirtiendo la matriz del sistema lineal. La dificultad radica en que no hay información acerca de los a_{ij} ni de los $s_i(t)$. Una aproximación al problema sería utilizar más información sobre las propiedades estadísticas de las señales $s_i(t)$ para poder estimar los a_{ij} y los $s_i(t)$. Sorprendentemente, resulta suficiente suponer que las señales son estadísticamente independientes para discriminar las señales originales a partir de las señales registradas. Esta suposición no es poco realista, aunque puede resultar incorrecta en la práctica. El análisis de componentes independientes puede usarse para estimar los a_{ij} basándose en la información de independencia de las señales. En la figura 3 (extraída de [HOK01]) se puede ver la estimación obtenida utilizando este método a partir de las señales observadas en la figura 2.

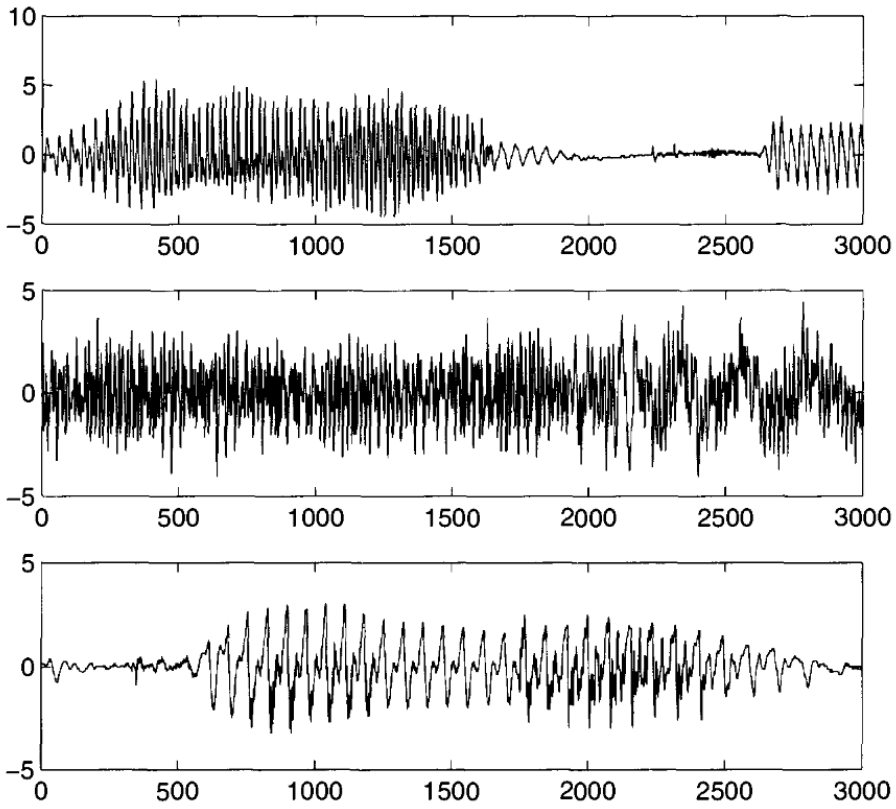
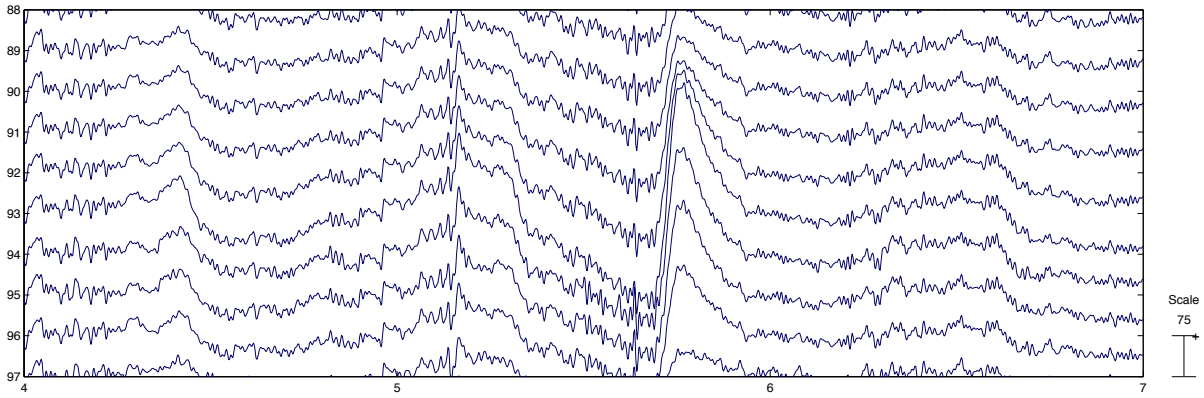
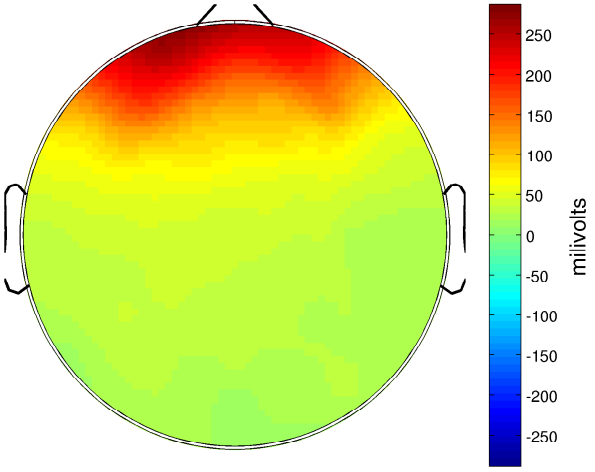


Figura 3: Estimación de las señales de audio originales a partir del registro realizado con tres micrófonos. Las variaciones con respecto a la señal original se deben a que el estimado puede estar multiplicado por alguna constante positiva o negativa.

ICA fue originalmente desarrollado para resolver problemas relacionados con el *cocktail-party problem* [BS95]. Sin embargo, actualmente el interés en ICA ha aumentado y ha quedado claro que este método resulta interesante para varias aplicaciones. Si se considera, por ejemplo, el registro de la actividad cerebral realizado por un electroencefalograma (EEG), se observa que las grabaciones son realizadas en varias ubicaciones distintas del cráneo, obteniendo señales resultantes de la suma de la actividad producida en la cabeza. Por ejemplo los dos paquetes abiertos más populares – EEGLAB (`http://sccn.ucsd.edu/eeglab/`) y FieldTrip (`http://fieldtrip.fcdonders.nl/`) usan fuertemente ICA. Estas señales son generadas por la actividad del cerebro y la actividad muscular. Análogamente, en este caso, se desean obtener las señales originales emitidas por la actividad cerebral, a partir del registro de múltiples señales resultantes donde se superponen la actividad cerebral y muscular [MWJ+02], por ejemplo removiendo los artefactos en la señal debidos al parpadeo de ojos [JMM+02].



(a) Extracto de EEG: Actividad eléctrica de los canales 88 a 97 en función del tiempo.



(b) Mapa encefálico de la actividad eléctrica en el cráneo de todos los canales a los 5.77 segundos

En la figura 2 se puede observar el resultado de un análisis de ICA realizado sobre un set de datos de prueba. En el extracto de EEG de la figura 4(a) se puede observar una variación significativa a los 5.77 segundos. El mapa de canales (4(b)) muestra la ubicación efectiva de la actividad, centrada en la parte anterior de la cabeza. Esto se puede asociar, luego del análisis ICA a la acción de pestañeo por parte del invididuo.

El algoritmo involucrado en este procesamiento resulta muy costoso debido a diversos aspectos. Un experimento con toma de datos de EEG consiste en un conjunto de series temporales de 132 canales, a 512Hz, con datos de punto flotante simple precisión; lo que implica que un experimento de 3 horas se aproxima a un tamaño del conjunto de datos de 3GB por sujeto.

Varios algoritmos han sido desarrollados para realizar este análisis. Entre los más populares se encuentran FastICA [Hyv02] e Infomax [BS95] [Ama96].

Estos métodos utilizan estos datos como matrices, las cuales deben operarse en una serie de pasos de preprocesamiento que involucran operaciones sencillas sobre éstas como centrar alrededor de la media, rotaciones, cálculo de autovalores y autovectores. Luego busca las direcciones estadísticamente independientes con una heurística iterativa, que requiere muchos pasos de procesamiento sobre esta matriz. Debido a este procesamiento y el gran tamaño de los datos, un experimento de las características mencionadas tiene un

tiempo de cómputo por sujeto de alrededor de 72 horas en una computadora de escritorio actual (Intel Core2duo 4GB RAM).

Los procesadores actuales no incluyen herramientas efectivas para operar con gran cantidad de números en punto flotante. Hoy en día, éstos incluyen características avanzadas como SSE que permiten este procesamiento para pocos datos en simultáneo lo que los hacen ineficientes para este tipo de cálculo. Un enfoque posible para resolver este inconveniente es el cálculo en paralelo en cluster beowulf [KHFM06]. Sin embargo, las aceleradoras gráficas se muestran como una alternativa interesante para este tipo de análisis.

Las placas de video o GPUs (*Graphical Processing Units*) han evolucionado en conjunto con la gran demanda provocada por la industria de los juegos de video. Los estándares de performance y calidad requeridos por esta industria son cada vez más exigentes. *CUDA* (acrónimo en inglés para *Common Unified Device Architecture*) es una arquitectura de cómputo paralelo desarrollada por *Nvidia*. Es una arquitectura que permite a los desarolladores de software utilizar estas GPUs para ejecutar programas.

Éstas incluyen cientos de procesadores vectoriales, diseñados específicamente para operar en punto flotante, y memoria de alta velocidad que permiten realizar una misma instrucción en una gran cantidad de datos en paralelo. Este tipo de ejecución resulta muy conveniente para los métodos de ICA, ya que siempre trabajan con números en punto flotante de doble precisión.

En esta tesis, se propone la implementación de ICA utilizando todo el poder de cómputo disponible en las aceleradoras gráficas para la descomposición de las señales de EEG y remoción de artefactos, con aplicación en experimentos cognitivos de aritmética.

En la sección 3 se describe el análisis teórico del problema y la justificación del algoritmo utilizado para realizar el análisis de componentes independientes. Luego, en la sección 4 se resume un conjunto de características técnicas sobre la arquitectura *CUDA* para poder comprender las optimizaciónes realizadas en el algoritmo descriptas en la sección 6. En esta misma sección, se hace un profundo análisis de las mejoras implementadas y se justifica la performance obtenida en los resultados de los experimentos. Finalmente, se detallan algunos casos de estudio (sección 7) y se proponen futuros trabajos relacionados con este tema (sección 9).

3. ICA

El análisis de componentes independientes (ICA en inglés) es un concepto definido por Pierre Comon en el año 1994 [Com94]. Este análisis, para un vector cualquiera, consiste en la búsqueda de una transformación lineal que minimice la dependencia estadística entre sus componentes. Con el fin de definir un criterio de búsqueda adecuados, se utiliza la expansión de información mutua como función de "cumulants" en orden creciente. Este concepto se puede ver como una extensión del análisis de componentes principales (PCA) que sólo acepta independencia hasta el segundo orden y, en consecuencia, define direcciones ortogonales.

Se presume el siguiente modelo estadístico:

$$x = My + v \tag{2}$$

donde x, y y v son vectores con valores en $\mathbb{R}$ o $\mathbb{C}$ con promedio cero y covarianza finita, M es una matriz rectangular con al menos tantas columnas como filas y el vector y tiene componentes estadísticamente independientes. El problema presentado por ICA puede ser resumido como sigue: dadas T muestras del vector x, se desea estimar la matriz M y sus muestras correspondientes del vector y. Sin embargo, por la presencia del ruido v, es generalmente imposible reconstruir el vector exacto y. Como se presume que el ruido v tiene una distribución desconocida, solo puede tratarse como una molestia y el análisis que se describe a continuación no se puede derivar del modelo anterior. En su reemplazo:

$$x = As \tag{3}$$

donde s es un vector cuyas componentes maximizan una función de *contraste*. Este contraste del vector s es máximo cuando las componentes son estadísticamente independientes.

3.1. Definiciones [HO00]

3.1.1. Independencia

Considerando dos variables y_1 e y_2, se dice que son independientes si la información sobre el valor de y_1 no da ninguna información sobre el valor de y_2 y viceversa.

Técnicamente, la independencia se puede definir por las densidades probabilísticas. Sea $p(y_1, y_2)$ la función de densidad de probabilidad (pdf) de y_1 e y_2, y sea $p_1(y_1)$ la función de densidad de probabilidad marginal de y_1:

$$p_1(y_1) = \int p(y_1, y_2) dy_2 \tag{4}$$

y de manera similar para y_2. Definimos a y_1 e y_2 independientes si y sólo si la función de densidad de probabilidad conjunta es factorizable de la siguiente forma:

$$p(y_1, y_2) = p_1(y_1) p_2(y_2) \tag{5}$$

Esta definición se extiende para n variables, en cuyo caso la densidad conjunta debe ser un producto de n términos.

Esta definición se utiliza para derivar una propiedad importante sobre las propiedades de variables aleatorias independientes. Dadas dos funciones h_1 y h_2, se tiene que:

$$E\{h_1(y_1)h_2(y_2)\} = E\{h_1(y_1)\}E\{h_2(y_2)\} \tag{6}$$

donde E es la esperanza.

Una restricción fundamental de ICA es que sus componentes independientes deben ser no-gaussianas para que ICA sea posible. Para verificar esta restricción, se presume que la matriz A es ortogonal y los s_i son gaussianos. Entonces x_1 y x_2 son gaussianos, sin correlación y de varianza unitaria. Entonces, la función de densidad es, por definición:

$$p(x_1, x_2) = \frac{1}{2\pi} exp(-\frac{x_1^2 + x_2^2}{2}) \tag{7}$$

Esta densidad es completamente simétrica, por lo que no contiene información sobre las direcciones de las columnas de la matriz A y, en consecuencia, A no puede ser estimada.

Siendo más rigurosos, se puede probar que la distribución de cualquier transformación ortogonal de un par (x_1, x_2) gaussiano tiene exactamente la misma dirección que (x_1, x_2) y que x_1 y x_2 son independientes. En consecuncia, en el caso de variables gaussianas, sólo podemos estimar el método ICA hasta una transformación ortogonal.

3.1.2. *Nongaussianity* como medida de independencia

La clave para estimar el modelo ICA es la *nongaussianity* o no-gausseanidad. El *Teorema del Límite Central* dice que la distribución de una suma de variables aleatorias independientes bajo ciertas condiciones tiende a una distribución gaussiana. En consecuencia, la suma de dos variables aleatorias independientes usualmente tiene una distribución mas cercana a la gaussiana que las variables originales. Sea x un vector de datos distribuídos de acuerdo al modelo de datos de ICA en la ecuación (3). Para estimar una de las componentes independientes, consideramos una combinación lineal de las variables x_i:

$$y = w^T x = \sum_i w_i x_i \tag{8}$$

donde w es el vector a ser determinado. Si w es una de las filas de la inversa de A, entonces esta combinación lineal sería, efectivamente, una de las componentes independientes. La pregunta es: ¿cómo se puede utilizar el Teorma del Límite Central para determinar w tal que iguale una de las filas de la inversa de A? En la práctica, no se puede determinar un w exacto, pero se puede usar un estimador que da una buena aproximación. Sea $z = A^T w$. Entonces, de (8) tenemos

$$y = w^T x = w^T A s = z^T s \tag{9}$$

y es entonces una combinación lineal de s_i, con pesos dados por z_i. Como la suma de dos variables aleatorias independientes es mas gaussiana que las variables originales, $z^T s$ es aún más gaussiana que cualquiera de los s_i y es de menor gausseanidad cuando es igual a uno de los s_i. En este caso, obviamente, sólo uno de los elementos z_i de z es distinto de cero. En consecuencia, podemos tomar w a un vector que maximize la no-gausseanidad de $w^T x$. Esto significa que $w^T x = z^T x$ equivale a una de las componentes independientes. Maximizando la no-gausseanidad de $w^T x$ nos da una componente independiente.

Si buscamos la no-gausseanidad del espacio n-dimensional de vectores w, obtenemos $2n$ máximos locales: dos para cada componente independiente correspondientes a $-s_i$ y s_i. Obtener las componentes independientes es simplemente obtener estos máximos locales.

Kurtosis La medida clásica para obtener la no-gausseanidad es la *kurtosis* o el "cumulant" de cuarto orden. La kurtosis de y se define como:

$$kurt(y) = E\{y^4\} - 3(E\{y^2\})^2 \tag{10}$$

Como y se supuso de varianza unitaria, se puede simplificar el lado derecho de la ecuación por $E\{y^4\} - 3$. Esto indica que la kurtosis es simplemente una versión normalizada del cuarto momento $E\{y^4\}$. Para una variable gaussiana, esto equivale a $3(E\{y^2\})^2$. En consecuencia, la kurtosis es cero para una variable aleatoria gaussiana. Esta medida puede ser tanto negativa como positiva. Las variables con kurtosis negativas se las llama subgaussianas, y las de kurtosis positivas, supergaussianas. La no-gausseanidad se puede medir como el valor absoluto de la kurtosis o el cuadrado. La principal motivación para utilizar la kurtosis como medida es su simpleza teórica y computacional. Teóricamente, se simplifica por la existencia de la siguiente propiedad:

$$kurt(x_1 + x_2) = kurt(x_1) + kurt(x_2) \tag{11}$$

y

$$kurt(\alpha x_1) = \alpha^4 kurt(x_1) \tag{12}$$

donde α es un escalar.

A modo ilustrativo, el siguiente ejemplo muestra como se estima la kurtosis y, en consecuencia, como se obtienen las componentes independientes a partir de la minimización o maximización de la kurtosis. Tomemos un modelo bidimensional $x = As$. Se presume que las variables independientes s_1 y s_2 tienen kurtosis distintas a cero y varianza unitaria. La idea es obtener una de las componentes independientes de forma tal que $y = w^T x$. Utilizando la transformación aplicada en (9) obtenemos que

$$z^T s = z_1 s_1 + z_2 s_2 \tag{13}$$

Por la propiedad de la kurtosis:

$$kurt(y) = kurt(z_1 s_1) + kurt(z_2 s_2) = z_1^4 kurt(s_1) + z_2^4 kurt(s_2) \tag{14}$$

Por otro lado, basándose en las mismas suposiciones que s_1 y s_2, la varianza de y es igual a 1. Esto implica la siguiente restricción:

$$E\{y^2\} = z_1^2 + z_2^2 = 1 \tag{15}$$

Esto indica que el vector z esta restringido al círculo de radio unitario en el plano bidimensional. El problema de optimización ahora es maximizar

$$|kurt(y)| = |z_1^4 kurt(s_1) + z_2^4 kurt(s_2)| \tag{16}$$

en el círculo unitario.

Como se demuestra en [DL95], los máximos ocurren cuando exactamente uno de los elementos del vector z es cero y el otro distinto de cero. Por la condición del circulo, el elemento distinto de cero puede ser 1 o -1. Pero estos puntos son los que igualan y con algunas de las componentes independientes $\pm s_i$, por lo que el problema ya esta resuelto. En la práctica, se toma un vector de pesos w, se computa la dirección en la cual la kurtosis de $y = w^T x$ aumenta más rápidamente basándose en una muestra $x(1), \ldots, x(T)$ de un vector de mezcla x y se usa el método del gradiente o alguna de sus extensiones para encontrar un nuevo vector w. Este ejemplo puede ser generalizado a dimensiones arbitrarias, demostrando que la kurtosis se puede usar como un problema de optimización para resolver ICA.

Negentropy Otra medida para la no-gausseanidad está dada por la *negentropy* o entropía negativa. Se basa en la cantidad teórica de entropía diferencial. La entropía es un concepto básico en la teoría de la información. Dada una variable aleatoria, la entropía se puede interpretar como el grado de información que da la observación de dicha variable. Mientras más aleatoria es la variable, más información da, mayor es su entropía.

La entropía H para una variable aleatoria discreta Y se define como

$$H(Y) = -\sum_i P(Y = a_i) log P(Y = a_i) \tag{17}$$

donde a_i son los posibles valores de Y. Esta definición se puede generalizar para variables aleatorias continuas y vectores, en cuyo caso se denomina entropía diferencial. La entropía diferencial para un vector aleatorio y con densidad $f(y)$ se define en [CTW+91] como:

$$H(y) = -\int f(y) log f(y) dy \tag{18}$$

Un resultado fundamental de la teoría de la información es que una variable gaussiana tiene la mayor entropía entre todas las variables aleatorias de igual varianza [CTW+91]. Esto indica que la entropia puede ser utilizada como una medida de no-gausseanidad. Para obtener una medida de no-gausseanidad que es cero para una variable gaussiana y siempre no-negativa, se usa una pequeña modificación de la definición de entropía diferencial, llamada *negentropy* o entropía negativa. Se define a la entropía negativa J como

$$J(y) = H(y_{gauss}) - H(y) \tag{19}$$

donde y_{gauss} es una variable aleatoria con la misma matriz de covarianza que y.

Este método es óptimo para obtener la no-gausseanidad, sin embargo, es computacionalmente complicado. Estimar la entropía negativa usando la definición requeriría una estimación, posiblemente no paramétrica, de la función de densidad de probabilidad. Un método simple para obtener una aproximación de la entropía negativa es el siguiente [JS87]:

$$J(y) \approx \frac{1}{12} E\{y^3\}^2 + \frac{1}{48} kurt(y)^2 \tag{20}$$

3.1.3. Minimización de la información mutua

Inspirado en la teoría de la información, otra aproximación para estimar el modelo ICA es utilizar la minimización de la información mutua. Si bien la explicación es distinta, lleva al mismo concepto explicado en la seccion 3.1.2 de buscar las direcciones no-gaussianas.

Usando el concepto de entropía diferencial, se define la información mutua I entre m variables aleatorias y_i como

$$I(y_1, y_2, \ldots, y_m) = \sum_{i=1}^{m} H(y_i) - H(y) \tag{21}$$

La información mutua es una medida natural de dependencia entre variables aleatorias. Es siempre no-negativa y es cero si y sólo si las variables son estadísticamente independientes. Este método tiene en cuenta toda la dependencia estructural de las variables y no sólo la covarianza como PCA y sus métodos relacionados.

3.2. Infomax (maximización de la información)

El principio que utiliza este algoritmo es el de maximizar la información mutua que la salida Y de un procesador de una red neuronal contiene acerca de su entrada X. En este caso, se define como

$$I(Y, X) = H(Y) - H(Y|X) \tag{22}$$

donde $H(Y)$ es la entropía de la salida Y y $H(Y|X)$ es la entropía de la salida que no se obtiene de la entrada. $H(Y)$ es, de hecho, la entropía diferencial de Y con respecto a alguna referencia como el nivel de ruido o la precisión de la discretización de la variables en X e Y. Para solucionar estas complejidades, sólo se considera el gradiente de cantidades teóricas de información con respecto a algún parametro, en este caso, w. Estos gradientes se comportan como entropías de variables discretas, dado que los terminos que hacen involucrar estas referencias desaparecen de la ecuacion. La ecuación (22) puede ser diferenciada, con respecto a un parámetro w, como

$$\frac{\partial}{\partial w} I(X, Y) = \frac{\partial}{\partial w} H(Y) \tag{23}$$

porque $H(X|Y)$ no depende de w.

En el sistema (2), $H(X|Y) = v$. Sin importar cual es el nivel de ruido adicional, maximizar la información mutua $I(X, Y)$ es equivalente a la maximización de la entropía de la salida $H(Y)$ porque $\frac{\partial}{\partial w} H(v) = 0$

En consecuencia, para cualquier relación continua, invertible y determinística entre X e Y, la información mutua puede maximizarse mediante la entropía de la salida: $H(Y)$.

Para el caso unitario, sea x la entrada y $g(x)$ la función de transformación tal que $g(x) = y$ con y la salida, tanto $I(y, x)$ como $H(y)$ son maximizadas cuando se alinean las partes de alta densidad de la función de densidad de probabilidad (pdf) de x con las pendientes altas de la función $g(x)$. Esta acción se puede describir como "corresponder la función de entrada-salida de una neurona con la distribución esperada de las señales" expresada en [Lau81].

Cuando $g(x)$ es monótona creciente o decreciente, la función de densidad de probabilidad de la salida $f_y(y)$ se puede escribir en función de la función de densidad de probabilidad de la entrada $f_x(x)$:

$$f_y(y) = \frac{f_x(x)}{|\partial y/\partial x|} \tag{24}$$

La entropía de la salida $H(y)$ viene dada por:

$$H(y) = -E\left[lnf_y(y)\right] = -\int_{-\infty}^{\infty} f_y(y) lnf_y(y) dy \tag{25}$$

Sustituyendo (24) en (25) obtenemos:

$$H(y) = -E\left[ln\left|\frac{\partial y}{\partial x}\right|\right] - E\left[lnf_x(x)\right] \tag{26}$$

El segundo término de la derecha (la entropía de x) puede presumirse que no es afectada por alteraciones en un parámetro w que determine $g(x)$. En consecuencia, para maximizar la entropía de y al cambiar w, sólo es necesario analizar el primer término. Esto se puede hacer considerando el "conjunto de entrenamiento" de los x para aproximar la función densidad $f_x(x)$ y derivar en una regla estocástica de gradiente ascendente:

$$\Delta w \propto \frac{\partial H}{\partial w} = \frac{\partial}{\partial w}\left(ln\left|\frac{\partial y}{\partial x}\right|\right) = \left(\frac{\partial y}{\partial x}\right)^{-1} \frac{\partial}{\partial w}\left(\frac{\partial y}{\partial x}\right) \tag{27}$$

Para el módelo de red neuronal descripto en (22), en el caso de una función de transferencia logística, tenemos:

$$y = \frac{1}{1+e^{-u}} \qquad u = wx \tag{28}$$

en cuyo caso la entrada es multiplicada por un peso w. El término evalúa en:

$$\frac{\partial y}{\partial x} = wy(1-y) \tag{29}$$

$$\frac{\partial}{\partial w}\left(\frac{\partial y}{\partial x}\right) = y(1-y)\left[1+wx(1-2y)\right] \tag{30}$$

Dividiendo (30) por (29) tenemos la regla de aprendizaje para la función logística:

$$\Delta w \propto \frac{1}{w} + x(1-2y) \tag{31}$$

Utilizando un razonamiento similar, obtenemos el sesgo:

$$\Delta w_0 \propto 1 - 2y \tag{32}$$

La regla que maximiza la información para una entrada y una salida puede ser efectiva para estructuras como la sinapsis y fotoreceptores que deben posicionar la ganancia de su

no-linealidad a un nivel apropiado para el valor promedio y tamaño de las fluctuaciones [Lau81].

Análogamente se puede derivar el modelo para el caso multidimensional:

$$\Delta W \propto \left[W^T\right]^{-1} + (1-2y)x^T \tag{33}$$

$$\Delta W_0 \propto 1 - 2y \tag{34}$$

Por la siguiente igualdad:

$$1 - 2\left(\frac{1}{1+e^{-(Wx)}}\right) = -tanh(\frac{Wx}{2}) \tag{35}$$

Reemplazando en (33):

$$\Delta W \propto \left[W^T\right]^{-1} - tanh(\frac{Wx}{2})x^T \tag{36}$$

El método del gradiente natural (o relativo) simplifica considerablemente el método del gradiente. El principio del gradiente natural [Ama96], [Ama97] se basa en la estructura geométrica del espacio de los parámetros y se relaciona con el principio del gradiente relativo [CL96] que utiliza ICA. Estos dos principios concluyen en multiplicar el lado derecho de (36) por W^TW.

$$\Delta W \propto \left(\left[W^T\right]^{-1} - tanh(\frac{Wx}{2})x^T\right) W^TW \tag{37}$$

Deriva en:

$$\Delta W \propto \ W - tanh(\frac{Wx}{2})(Wx)^TW \tag{38}$$

3.2.1. ¿Por qué el algoritmo de maximización de la información minimiza la dependencia estadística?

Consideremos, por ejemplo, un sistema con dos salidas y_1 y y_2. La entropía conjunta de estas variables se puede escribir como:

$$H(y_1, y_2) = H(y_1) + H(y_2) - I(y_1, y_2) \tag{39}$$

Maximizar la entropía conjunta consiste en maximizar la entropía individual, mientras se minimiza la información mutua $I(y_1, y_2)$ compartida por las dos variables. Cuando esta última cantidad es cero, las dos variables son estadísticamente independientes, y la función de densidad de probabilidad se puede factorizar

$$f_{y_1,y_2}(y_{1,y_2} = f_{y_1}(y_1)f_{y_2}(y_2) \tag{40}$$

ICA es un ejemplo de minimización de $I(y_1, y_2)$ para todo par (y_1, y_2). El algoritmo presentado en la sección 3.2 es un algoritmo estocástico de gradiente ascendente que maximiza la entropía conjunta en (39). Al hacerlo, generalmente reduce $I(y_1, y_2)$, reduciendo la dependencia estadística entre ambas salidas.

Sin embargo, este algoritmo no garantiza el alcance del mínimo absoluto de $I(y_1, y_2)$ debido a algunas interferencias por otros términos en las ecuaciones. En muchas situaciones el efecto de estas interferencias es mínimo. Una conjetura [BS95] es que sólo cuando las funciones de densidad de probabilidad de las entradas son sub-gaussianas, se pueden encontrar soluciones no deseadas con mayor entropía. Muchas de las señales analógicas del mundo real son super-gaussianas. Para estas señales, maximizar la entropía minimiza la información mutua entre las salidas.

Existe una modificación de este algoritmo llamado *Infomax extendido* [LGS99] que utiliza un razonamiento similar para obtener un algoritmo que acelara su convergencia y es eficiente tanto para señales sub- y super-gaussianas. No se hará el análisis del algortimo por su similitud, sin embargo serán optimizados ambos algoritmos como se explica en la sección 6.

4. *CUDA*

La principal función de una placa de video es generar una imagen bidimensional dados una ubicación en el espacio y los datos vectoriales sobre los distintos elementos en el. Esto requiere de millones de operaciones de punto flotante por segundo para obtener una imagen en alta resolución y un nivel aceptable de cantidad de cuadros renderizados por segundo. No hay muchas alternativas para lograr aumentar la cantidad de operaciones que se pueden realizar por segundo. Una opción es aumentar la velocidad de procesamiento. Otra, aumentar la cantidad de procesadores. Las operaciones críticas en este ámbito son las transformaciones lineales, operaciones que son completamente paralelizables. Para mejorar la performance de las placas de video, los diseñadores e ingenieros optaron por la inclusión de muchas unidades de procesamiento con una velocidad de cómputo equivalente a la mitad de un procesador de escritorio.

Una de las diferencias más significativas con los CPUs, es que *CUDA* hace énfasis en el procesamiento de muchos *threads* en parallelo, mientras que los CPUs enfatizan la ejecución rápida de un sólo *thread*.

El cuadro 1 muestra las diferencias entre una placa de video y un procesador de alta performance. La velocidad por procesador de un CPU es dos veces mayor a la de un GPU. Sin embargo, cuando se utilizan todos los procesadores de un GPU, la cantidad teórica de operaciones de punto flotante por segundo se ve incrementada en un orden de magnitud.

Cuadro 1: Comparación entre GPU y CPU

	Tesla C2070 GPU	Intel Core i7 975 Extreme Edition
GFLOPS[c]:	515[a]	55.36[b]
Cores:	448	4
Clock Speed (Ghz):	1.15	3.33
GFLOPS[c] por core:	1.149	13.75

[a] `http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf`

[b] `http://www.intel.com/support/processors/sb/cs-023143.htm`

[c] GFLOPS teóricos.

Para aprovechar todo este poder de procesamiento, *CUDA* provee al programador de las herramientas necesarias para generar un programa que puede ser ejecutado en estos multiprocesadores. Actualmente, el kit de desarrollo está compuesto por un compilador con extensiones, un analizador de performance visual y bibliotecas aceleradas por GPU entre las que se encuentra BLAS[1]. El compilador soporta un subconjunto de funcionalidades de los lenguajes C y C++ y extensiones para compilar código para ser ejecutado en el GPU mediante la introducción de nuevos elementos sintácticos.

4.1. Modelo de programación

Los diseñadores de *CUDA* hacen énfasis en su escalabilidad. No sólo soporta el procesamiento en paralelo. Está pensada para abstraer al programador del dispositivo en el que

[1] `http://www.netlib.org/blas/`

va a ser procesado. Las diferentes plataformas que soportan *CUDA* tienen características muy diversas en lo que a cantidad de procesadores y capacidad de procesamiento refiere. El desafío que presenta es el desarrollo de software que escale su paralelismo al mismo tiempo que aumenta la cantidad de procesadores disponibles. *CUDA* fue pensada para abstraer al desarrollador de este desafío mientras se mantiene simple el modelo de programación.

Una porción de código para ser ejecutado en *CUDA* se define como un *kernel*. Es una función en lenguaje C que es prefijada con una palabra clave y una configuración de ejecución. La configuración de ejecución define dos nuevos conceptos: *blocks* y *threads*. En la configuración se definen cuantos *blocks* y *threads* se van a ejecutar para ese *kernel*. En el ejemplo 4 se puede ver la sintaxis en un ejemplo de suma de vectores.

```
// Kernel
__global__ void sumarVector(float* A, float* B, float* C) {
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main() {
    ...
    // Invocacion a sumarVector con 1 bloque y N threads
    sumarVector<<<1, N>>>(A, B, C);
}
```

Ejemplo 4: Suma de vectores en *CUDA*

Una GPU está compuesta por varios *cores* o multiprocesadores. Un mutiprocesador, a su vez, es un grupo de unidades de procesamiento. Cada *thread* se ejecuta en una unidad de procesamiento mediante *warps*, los que serán explicados más adelante. Los threads se agrupan en *blocks*. Cada *block* es asignado a un multiprocesador. Cuando el multiprocesador termina de ejecutar todos los *threads* de un *block*, se le asigna un nuevo *block* Ċuando se terminan de ejecutar todos los *blocks* de un *kernel*, se da por finalizada la ejecución del *kernel*.

Adicionalmente, *CUDA* tiene su propio espacio de memoria, divido en memoria global, compartida y local. La memoria global es accesible desde todos los *threads* y permanente entre ejecuciones de distintos kernels. La memoria compartida es accesible por todos los *threads* de un *block* y los datos son eliminados luego de la finalización de la ejecución del *block* La memoria local es accesible por un sólo *thread* Ėstos conceptos serán explicados con mayor profundidad en la sección 4.5.

Dentro del código de un *kernel* existen dos variables para identificar el *block* y el *thread*. Las variables especiales `threadIdx` y `blockIdx` determinan el *block* y *thread* en ejecución.

En el ejemplo 6 se puede observar las variables `threadIdx` y `blockIdx`. Estas se pueden definir en forma unidimensional, bidimensional o tridimensional (sólo para *threads*), permitiendo una mejor organización dentro del código.

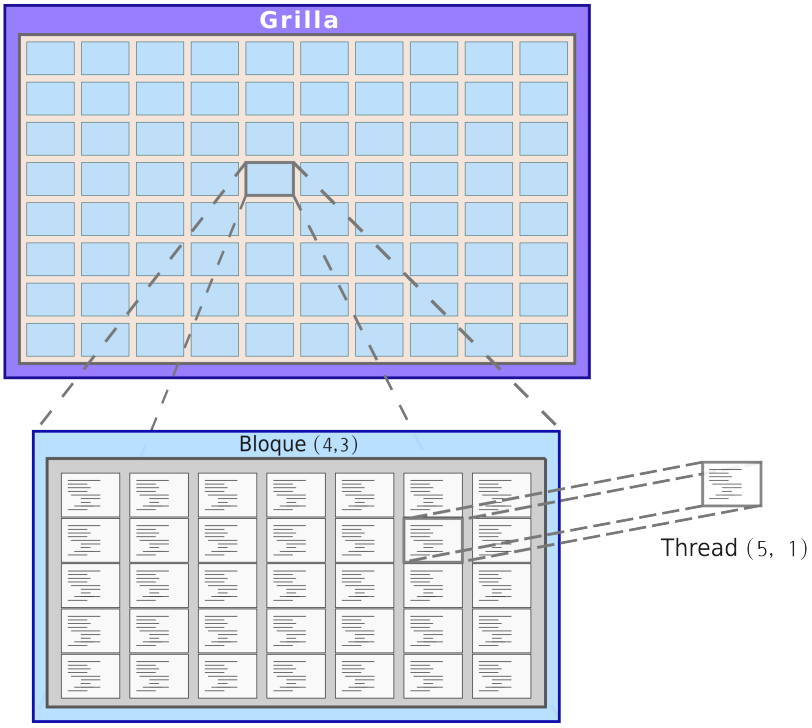


Figura 5: Detalle de una grilla de *blocks* y *threads*

Existen otras variables especiales como `gridDim` y `blockDim` que identifican las dimensiones de la grilla (conjunto de *blocks*) y las dimensiones del *block* (conjunto de *threads*). En el ejemplo 7 se puede ver una suma de matrices genérica que, sin importar el número de filas o columnas de la matriz, realiza el cálculo correcto.

4.1.1. Capacidad de cómputo

La capacidad de cómputo de un dispostivo se define a partir de un par de números de revisión: *major* y *minor*. Dispositivos con el mismo *major* tiene la misma arquitectura en el núcleo. El número *minor* indica mejoras en la arquitectura. La tabla 2 muestra un subconjunto de las características técnicas por capacidad de cómputo.

```
// Kernel
__global__ void sumarMatriz(float* A, float* B, float* C, int columnas) {
    int i = threadIdx.x;
    int j = blockIdx.x;
    C[i + columnas * j] = A[i + columnas * j] + B[i + columnas * j];
}

int main() {
    ...
    // Invocacion a sumarMatriz con N bloques y M threads
    sumarMatriz<<<N, M>>>(A, B, C, M);
}
```

Ejemplo 6: Suma de matrices en *CUDA*

```
// Kernel
__global__ void sumarMatriz(float* A, float* B, float* C) {
    int i = threadIdx.x;
    int j = blockIdx.x;
    int columnas = blockDim.x;
    C[i + columnas * j] = A[i + columnas * j] + B[i + columnas * j];
}

int main() {
    ...
    // Invocacion a sumarMatriz con N bloques y M threads
    sumarMatriz<<<N, M>>>(A, B, C);
}
```

Ejemplo 7: Suma genérica de matrices en *CUDA*

	Capacidad de cómputo				
Característica	**1.0**	**1.1**	**1.2**	**1.3**	**2.X**
Doble precisión	No				Sí
Máximo valor para dimensiones X e Y de una grilla de *blocks*	65535				
Cantidad máxima de *threads* por *block*	512				1024
Máximo valor para dimensiones X e Y de un *block*	512				1024
Máximo valor para dimensiones Z de un *block*	64				
Cantidad de *threads* por *warp*	32				
Cantidad máxima de *blocks* residentes por multiprocesador	8				
Cantidad máxima de *warps* residentes por multiprocesador	24		32		48
Cantidad máxima de *threads* residentes por multiprocesador	768		1024		1536
Cantidad de registros por multiprocesador	8K		16K		32K
Cantidad máxima de memoria compartida por multiprocesador	16 Kb				48 Kb
Cantidad de bancos de memoria compartida por multiprocesador	16				32
Cantidad de memoria local por *thread*	16 Kb				512 Kb
Cantidad máxima de instrucciones por *kernel*	2 millones				

4.2. Implementación

El GPU mantiene un *scheduler* y ejecuta los *threads* en grupos de 32, llamados *warps*. Todos los *threads* que componen un *warp* comienzan su ejecución en el mismo punto, pero tienen distintos contadores de instrucciones y registros de estados. Cada *thread* puede tomar un camino distinto y ejecutar independientemente de los otros en el *warp*.

Cada *warp* ejecuta una instrucción por vez. Si algún *thread* de un *warp* se bifurca y toma otro camino, se ejecutan ambos en serie, deshabilitando los *threads* que no correspondan en cada caso. Luego, al final de todas las bifurcaciones cuando los *threads* convergen, se continúa la ejecución en forma paralela. Por este motivo es muy importante mantener todos los *threads* del mismo *warp* sobre el mismo camino de ejecución.

El contexto de ejecución (contadores de programa, registros, etc.) de cada *warp* procesado por un multiprocesador es mantenido en el chip hasta la finalización de la ejecución. En consecuencia, el cambio de contexto no tiene costo.

Cada multiprocesador tiene un conjunto de registros de 32 bits que son particionados entre los *warps* y memoria compartida que es particionada entre los *blocks*.

El número de *blocks* y *warps* que pueden residir y ser procesados por un multiprocesador depende de la cantidad de registros y memoria utilizada por el *kernel* al que corresponden. Existe también un límite prefijado que fija la cantidad máxima de *warps* por procesador. Este límite, junto con la cantidad de registros y el tamaño de la memoria compartida depende de la capacidad de cómputo del dispositivo.

Si no hay suficientes registros o memoria compartida por multiprocesador para procesar al menos un *block*, le ejecución del *kernel* correspondiente fallará.

El número total de *warps* por *block* se define de la siguiente manera:

$$W_{block} = \lceil \frac{T_{block}}{32} \rceil$$

donde T_{block} es la cantidad de *threads* del *block*.

El número total de registros por *block* se puede obtener realizando el siguiente cálculo:

- Para dispositivos de capacidad de cómputo 1.X:

$$R_{block} = \lceil \frac{\lceil \frac{W_{block}}{2} \rceil * 32 * R_{kernel}}{G_{thread}} \rceil$$

- Para dispositivos de capacidad de cómputo 2.X:

$$R_{block} = \lceil \frac{R_{kernel} * 32}{G_{thread}} \rceil * W_{block}$$

donde R_{kernel} es el número de registros utilizados por el *kernel* y G_{thread} es la granularidad de reserva de *threads*, equivalente a 256 para dispositivos de capacidad de cómputo 1.0 y 1.1, 512 para 1.2 y 1.3 y 64 para 2.X.

La cantidad total de memoria compartida se calcula de la siguiente forma:

$$S_{block} = \lceil \frac{S_{kernel}}{G_{shared}} \rceil$$

donde S_{kernel} es la cantidad de memoria compartida utilizada por el *kernel* y G_{shared} es la granularidad de reserva de memoria compartida: 512 para dispositivos de capacidad de cómputo 1.X y 128 para 2.X.

Una de las claves para obtener una buena performance es mantener a cada uno los multiprocesadores en el dispositivo lo más ocupado posible. La ejecución de los *threads* en el dispositivo no es independiente: se ejecutan en *warps*. Como el número de *threads* por *warp* es fijo, al igual que el número de unidades de ejecución por multiprocesador, mantener el número de *threads* por *block* múltiplo del tamaño de un warp es una forma de utilizar todas las unidades de ejecución del multiprocesador al mismo tiempo. El problema es que además de la cantidad de *threads* por *block*, hay otros recursos que limitan la cantidad de *threads* que se pueden ejecutar en paralelo dentro de un multiprocesador, dejando sin uso unidades de ejecución: la cantidad de registros y memoria compartida.

Se define una métrica: *Ocupación*. Es el radio de *warps* activos sobre el máximo número de *warps* activos soportado por un multiprocesador.

La cantidad de registros utilizados por *thread* y la cantidad de memoria compartida por *block* limitan la cantidad de *warps* que pueden estar ejecutándose en paralelo dentro de un multiprocesador. Maximizar la ocupación es un factor clave para lograr una mejor optimización. Sin embargo, no siempre garantiza una mejor performance:

- El uso de registros es mucho más rápido que el uso de memoria local. Utilizar memoria local puede aumentar la disponibilidad de registros y la ocupación, pero puede aumentar el tiempo de procesamiento de cada thread.
- La alineación de los accesos a memoria es más importante que la maximización de la ocupación (ver 4.5).

4.3. Sincronización CPU-GPU

Un programa que utiliza en cuda no es más que una porción de código *host* (código a ser ejecutado en el CPU) con llamadas a *kernels* cuando se requiera ejecutar código en el GPU. La ejecución de kernels es asincrónica: el CPU continúa la ejecución de código host inmediatamente después de lanzar el *kernel* en el GPU. Exceptuando las últimas versiones de *CUDA*[2], sólo se ejecuta un *kernel* por vez en un GPU. Las llamadas quedan encoladas en el GPU y cuando se desaloja un *kernel*, se continúa con la ejecución del próximo.

Como ventaja, este mecanismo permite la ejecución paralela de código en el CPU y el GPU. La desventaja se puede percibir a la hora de la programación: el programador debe ser conciente de esto y tomar los recaudos necesarios para sincronizar el código *host* y el GPU. Un claro ejemplo de ello es la espera de resultados. Si un programa debe realizar cálculos iterativamente basándose en una condición sobre uno de los datos y esta se verifica en el CPU, es importante esperar que el GPU termine de computar antes de

[2] *CUDA* 2.0 en adelante permite la ejecución concurrente de *kernels* siempre y cuando sea explicíta

verificar si debe continuar computándose o no. Mediante directivas específicas, se puede indicar al CPU que la ejecución, en cierto punto del código, debe esperar la finalización de todos los *kernels* pendientes en el GPU.

```
int main() {
    ...
    while ( maximo > MAX_VALUE) {
        // Invocacion a promediarMatriz con N bloques y M threads
        promediarMatriz<<<N, M>>>(A);

        // Sincronizacion con GPU
        cudaThreadSyncrhonize();

        // Obtencion del maximo valor de A
        maximo = obtenerMaximo(A);

    }
}
```

Ejemplo 8: Ejemplo de sincronización en código host

4.4. Sincronización en GPU

Uno de los espacios de memoria disponibles en un *thread* es la memoria compartida (ver 4.5.2). Los datos en este espacio son accesibles desde cualquier *thread* dentro de un *block*. Las lecturas a este espacio de memoria no generan problemas. Una operación de escritura sí los genera. No todos los *threads* de un *block* se ejecutan en paralelo. Si un *kernel* realiza lecturas a memoria compartida, procesa, y sobreescribe los datos, puede darse el caso en que no todos los *threads* del *kernel* procesen los mismos datos de entrada. Este conflicto se denomina *write after read (WAR o escritura previa a lectura)*. Otros conflictos similares son *read after write (RAW o lectura prevua a escritura)* y *write after write (WAW o escritura previa a escritura)*

Para solucionar esto conflictos, *CUDA* provee la directiva `void __syncthreads()`. Cuando la ejecución de un *thread* llega a la directiva, ese *thread* se bloquea hasta que el resto de los *threads* del *block* lleguen a ese punto.

No existen directivas para la sincronización entre *blocks*, aunque en la referencia de *CUDA* se dan a conocer técnicas para detectar el útlimo bloque que es ejecutado.

4.5. Modelo de memoría

Así como el CPU tiene su memoria, el GPU opera sobre su propia memoria. Es un espacio de memoria separado y organizado en forma jerárquica. Existen 3 espacios memoria:

- Memoria Global (*device memory*): es la memoria principal. Todos los threads de todos los bloques pueden acceder a esta memoria. Es persistente en relación con las

llamadas a *kernels* distintos. Usando directivas funcionalmente similares a *malloc* y *free*, el programador puede hacer uso de este espacio. Dependiendo del dispositivo, esta memoria puede ser de hasta 6 GB.

- Memoria Compartida (*shared memory*): esta memoria es compartida por los distintos *threads* que componen un mismo *block*. No es persistente a través de la ejecución de distintos *blocks* y, en consecuencia, tampoco lo es con *kernels*. Dependiendo de la versión de *CUDA* soportada por el dispositivo, puede ser de 16 Kb o 48 Kb por multiprocesador. Es mucho mas rápida que la memoria global.

- Memoria Local (*local memory*): Es una memoria que sólo esta disponible para un *thread* Ṗuede ser de 16 Kb o 512 Kb, dependiendo de la versión de *CUDA*.

Adicionalmente existen dos espacios de memoria de sólo lectura: la memoria de constantes y la memoria de texturas.

Cada memoria descripta tiene distintas características y métodos de acceso. Esto hace que, dependiendo del objetivo del programa, sea muy importante la elección del espacio correcto.

4.5.1. Memoria global

Esta es la memoria del dispositivo. Se accede mediante transacciones de 32, 64 y 128 bytes. Todas estas transacciones se realizan a direcciones alineadas. Cuando un *warp* accede a memoria, se combinan los accesos de todos los *threads* del *warp* en una o varias transacciones de memoria de estos tipos. Por ejemplo: si hay 32 *threads* en un *warp* que realizan una operación de lectura de 4 bytes, esto puede desencadenar una transacción de 128 bytes o, en el peor caso, 32 transacciones de 32 bytes. Mantener optimizados los accesos a memoria es fundamental para obtener una optimización eficiente utilizando *CUDA*.

Existen tres conceptos que se deben tener en cuenta al momento de realizar transacciones en memoria:

- Alineación: La memoria soporta lecturas o escrituras de 1, 2, 4, 8 o 16 bytes. Cada acceso se traduce en una sola transacción siempre y cuando la dirección de memoria a acceder este alineada al tamaño. En el caso de que no lo sea, esto puede desencadenar más de una transacción. Se recomienda utilizar tipos de datos que se encuentren bajo estos requerimientos de alineación y tamaño.

- *Padding*: Un problema con la alineación ocurre con los arreglos multidimensionales. Una matriz guardada por filas, de 10 filas por 31 columnas en memoria, solo tiene alineada una de cada 4 filas. *CUDA* tiene funciones especiales para obtener espacios de memoria para arreglos bidimensionales. Especificando la longitud de cada fila en bytes y la cantidad de filas necesarias, se puede obtener un puntero al espacio asignado y un valor que indica cual es el tamaño en bytes que se debe utilizar para obtener el primer elemento de la siguiente fila. De esta forma, el padding se realiza automáticamente. Sólo hay que tener en cuenta el valor correcto al momento de recorrer la matriz.

- Patrones de acceso a memoria: No sólo hay que tener en cuenta los accesos a memoria dentro del código, sino el conjunto de accesos que desencadena la ejecución de cada *warp*. La referencia de programación C de *CUDA* define varios patrones de acceso que se pueden dar. Básicamente, lo óptimo es mantener los accesos alineados (el *thread* 0 de cada *warp* accede a un elemento alineado) y hacerlos en forma secuencial (el *thread* N accede al elemento siguiente al que accedió el *thread* N-1).

4.5.2. Memoria compartida

Esta memoria reside en el chip. Es mucho mas rápida que la memoria global. Esta dividida en 16 [3] porciones de igual tamaño llamados *bancos*. Cada banco tiene un ancho de banda de 32 bits cada dos ciclos de reloj. Si dos (o más) *threads* acceden a algún byte dentro de diferentes palabras de 32 bytes pertenecientes a un mismo banco, se produce un *conflicto de banco*. Cuando se produce un conflicto, los accesos a memoria se serializan.

Existen patrones de acceso a memoria descriptos en la referencia C de *CUDA*. Principalmente, se puede observar que el acceso alineado y secuencial de arreglos y matrices no genera conflictos de bancos. Mejor aún, una forma de optimizar una multiplicación de matrices es utilizar memoria compartida para optimizar los accesos no alineados (Ejemplo 9).

[3] Sólo en *CUDA* 1.X. *CUDA* 2.X tiene 32

```
/*
 * Arreglo para guardar la columna en memoria compartida
 */
__shared__ float columna[N];

/*
 * Multiplica dos matrices A (M x N) y B (N x R).
 * El numero de bloque representa la columna de B.
 * El numero de thread representa la fila de A.
 */
__global__ multMat(float * A, float * B) {

    // Copia alineada y secuencial la columna a memoria compartida
    columna[threadIdx.x] = B[blockIdx.x * blockDim.x + threadIdx.x]

    __syncthreads();

    float valor = 0.0f;

    for (int i = 0; i < gridDim.x; i++) {
        // Acceso alineado a A
        // Acceso sin conflicto de bancos a columna (todos los threads al
            mismo elemento)
        valor += A[i * blockDim.x + threadIdx.x] * columna[i];
    }
}

void main() {
    ...
    multMat<<<N, M>>>(A, B);

}
```

Ejemplo 9: Ejemplo de multiplicación optimizada con memoria compartida

4.5.3. Memoria local

Los accesos a memoria local se producen para algunas variables que el compilador ubica automáticamente en esta memoria:

- Arreglos para los cuales no puede determinar si son indexados con cantidad constantes.
- Estructuras y arreglos cuyo tamaño puede consumir muchos registros.
- Cuando el *kernel* ocupa demasiados registros, ubica cualquier variable.

Esta memoria es parte de la memoria global del dispositivo, por lo que necesita los mismos patrones de acceso que la memoria global especificados en la sección 4.5.1.

5. Análisis de Infomax ICA

En esta sección se realiza un análisis exhaustivo de la implementación de Infomax ICA realizada por Sigurd Enghoff[4]. A partir del *profiling* realizado se propone una optimización sencilla utilizando bibliotecas estándar disponibles.

5.1. Análisis de performance

Resumiendo, una iteración de Infomax consiste en los siguientes pasos:

1. Se calcula una matriz U multiplicando los pesos W (matriz de canales x canales, inicializada en I) con un subconjunto de muestras permutadas de forma aleatoria de los datos x.
2. La matriz Y se calcula como $-tanh(U/2)$
3. La matriz YU se calcula como $Y \times U^T$
4. Se suma la matriz I a YU
5. Se calcula $W = lrate * YU * W + W$ donde $lrate$ es el coeficiente de aprendizaje en cada paso.

La cantidad de muestras de x que se utilizan puede ser un parámetro de configuración o, en el caso de que no este configurado, una heurística calculada a partir de $\sqrt{\frac{\#muestras}{3}}$.

Esta iteración se realiza n veces por cada *Step* del algoritmo, donde $n = \frac{\#muestras}{\sqrt{\frac{\#muestras}{3}}}$. Luego, al final de cada paso se verifica cuánto fue el cambio con respecto a los pesos W anteriores. Cuando este cambio es menor al criterio de parada o la cantidad máxima de *steps* se alcanza, el algoritmo se da por finalizado. Otras operaciones de matrices son necesarias para calcular estos valores, aunque la cantidad de veces que se calcula por ejecución no supera la cantidad máxima de *steps*, fijada en 512 salvo que el usuario indique lo contrario.

Utilizando la herramienta de análisis secuencial y optimización *callgrind* [Wei08], es posible realizar un detallado análisis de la ejecución del algoritmo que indica la cantidad de llamadas a funciones realizadas por el programa ejecutable y la porción de tiempo utilizada por cada una de ellas. A modo de ejemplo, se realizó un análisis de la ejecución del algoritmo implementado en C para un set de datos de 136 canales y 22528 muestras. La figura 10 muestra el mapa de las llamadas a BLAS realizadas durante esta ejecución: 47,9 % del tiempo fue consumido por el símbolo `dgemv` mientras que el 40,78 % fue consumido por `dgemm`. Este set de datos, a 512 Hz, corresponde a un experimento de 44 segundos. En experimentos reales el tiempo puede ser de varias horas, donde el análisis del porcentaje de tiempo consumido por estos símbolos se eleva a valores cercanos al 100 %.

Los símbolos `dgemv` y `dgemm` corresponden a las funciones de multiplicación de matriz-vector y matriz-matriz, de la biblioteca BLAS.

[4] `http://cnl.salk.edu/~enghoff/`

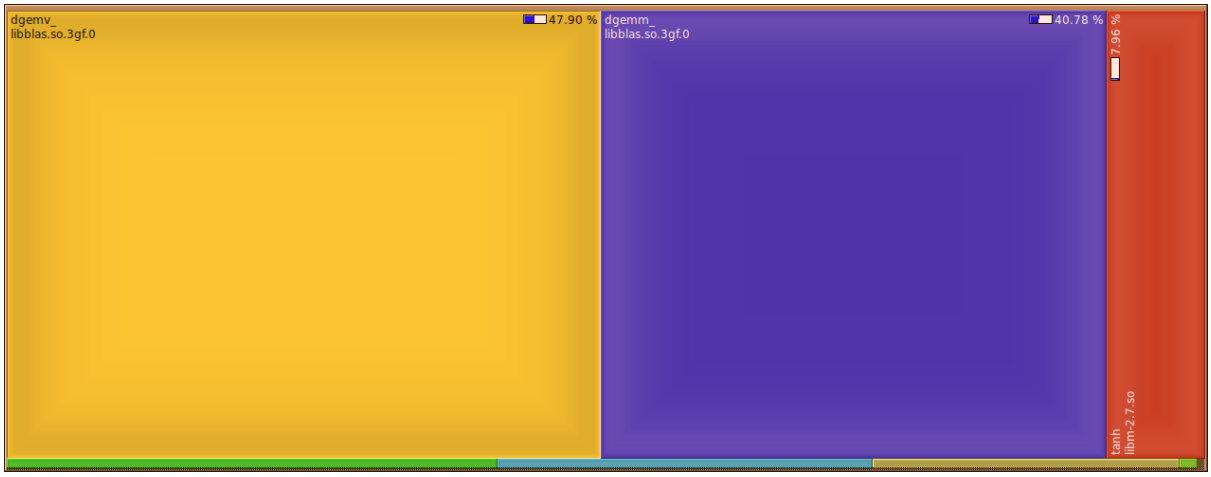


Figura 10: Mapa de llamadas de Infomax ICA. Las rutinas de *BLAS* `dgemm` y `dgemv` consumen alrededor del 80 % del tiempo total de ejecución. El gra'fico completo representa el 100 % del tiempo de ejecución y es una captura de pantalla de la herramienta *kcachegrind*

`void dgemv(char* trans, int m, int n, double alpha, double *a, int lda, double *x, int incx, double beta, double * y, int incy)`

Esto calcula: `y = alpha * a * x + beta * y` donde:

- `trans` determina si la matriz `a` debe ser transpuesta o no.
- `m` es el número de filas de `a`.
- `n` es el número de columnas `a`.
- `alpha` es un escalar.
- `a` es la matriz.
- `lda` es la dimensión principal de `a`.
- `x` es el vector.
- `incx` es el valor utilizado para acceder al siguiente elemento de `x`.
- `beta` es un escalar.
- `y` es el vector de salida.
- `incy` es el valor utilizado para acceder al siguiente elemento de `y`.

En el algoritmo, los parámetros se fijan para computar `y = a * x`, donde a es una matriz cuadrada con tantas filas como canales tiene el set de datos. El vector x corresponde a una muestra del experimento.

`dgemm(char *transa, char *trasnb, int m, int n, int k, double alpha, double *a, int lda, double *b, int ldb, double beta, double *c, int ldc)`

Esto calcula: `y = alpha * a * b + beta * c`, donde:

- `transa` determina si la matriz `a` debe ser transpuesta o no.
- `transb` determina si la matriz `b` debe ser transpuesta o no.
- `m` es el número de filas de `a` y `c`.
- `n` es el número de columnas de `b` y `c`.
- `k` es el número de columnas `a` y el número de filas de `c`.
- `alpha` es un escalar.
- `a` es la matriz.
- `lda` es la dimensión principal de `a`.
- `b` es la matriz.
- `ldb` es la dimensión principal de `b`.
- `beta` es un escalar.
- `c` es la matriz del resultado.
- `ldc` es la dimensión principal de `c`.

Los parámetros, en Infomax, se fijan para calcular `c = a * b`, donde la matriz `a` es cuadrada y el número de filas es igual al número de canales en el set de datos. Las dimensiones de `b` pueden ser iguales a las de `a` o, en otros casos, corresponden a $\#canales \times \sqrt{\frac{\#muestras}{3}}$

5.2. Implementación con *CUBLAS*

CUBLAS es una implementación de *BLAS* que utiliza *CUDA* como arquitectura para su ejecución. La biblioteca es auto contenida, por lo que no requiere de interacción con el driver de *Nvidia*.

El modelo básico de utilización requiere de la creación de matrices y vectores en espacio de memoria del GPU y su copia desde el espacio de memoria del CPU. Luego del procesamiento, utilizando las funciones correspondientes, estos datos deben ser copiados nuevamente al espacio de memoria del CPU.

La solución ad hoc consiste en:

- Crear todas las matrices en espacio de memoria del GPU.
- Inicializar los datos correspondientes en el espacio de memoria del GPU.
- Cambiar las llamadas a rutinas de *BLAS* por las correspondientes en *CUBLAS*.
- Copiar los resultados desde espacio de memoria del GPU al CPU.

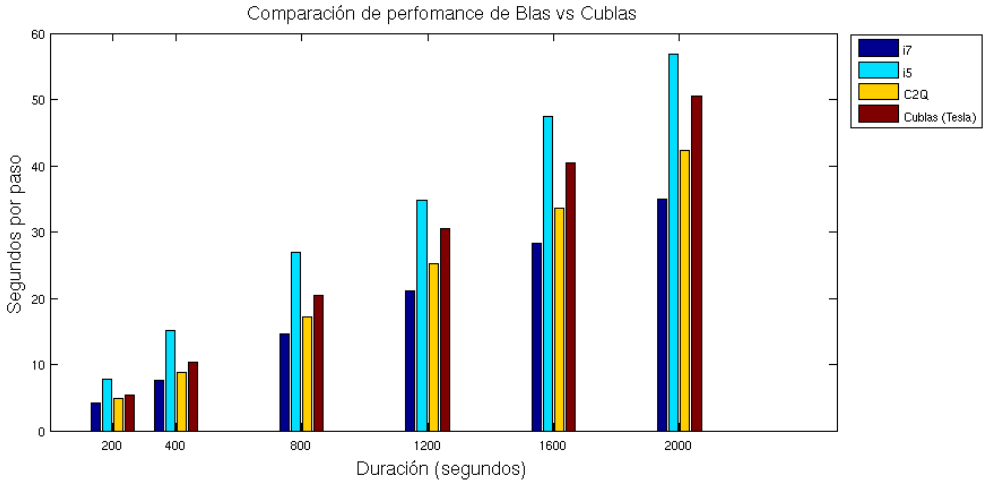


Figura 11: Detalle de tiempos obtenidos en la comparación de performance de *BLAS* y *CUBLAS*

La implementación de esta solución no resulta compleja, pero sus resultados no ofrecen una optimización razonable. De hecho, se realizaron pruebas sobre algunos experimentos y se corroboró que esta solución no mejoraba el tiempo de ejecución.

En la figura 11 se comparan los resultados obtenidos de procesar un EEG de 128 canales desde 200 hasta 2000 segundos de duración, utilizando la implemetación de Infomax usando *BLAS* en tres procesadores distintos: Intel Core i7-2600, Intel Core i5-750 e Intel Core2 Quad 9400 (ver 4). Como se puede observar, *CUBLAS* sólo presenta una leve mejora con respecto al procesador i5. Con respecto al procesador i7, esto representa un incremento del tiempo total de ejecución en un valor cercano al 50 %.

Un análisis mas detallado de esta implementación, muestra el punto débil: la multiplicación de matriz por vector no resulta óptima para las dimensiones de las matrices y los vectores de los experimentos.

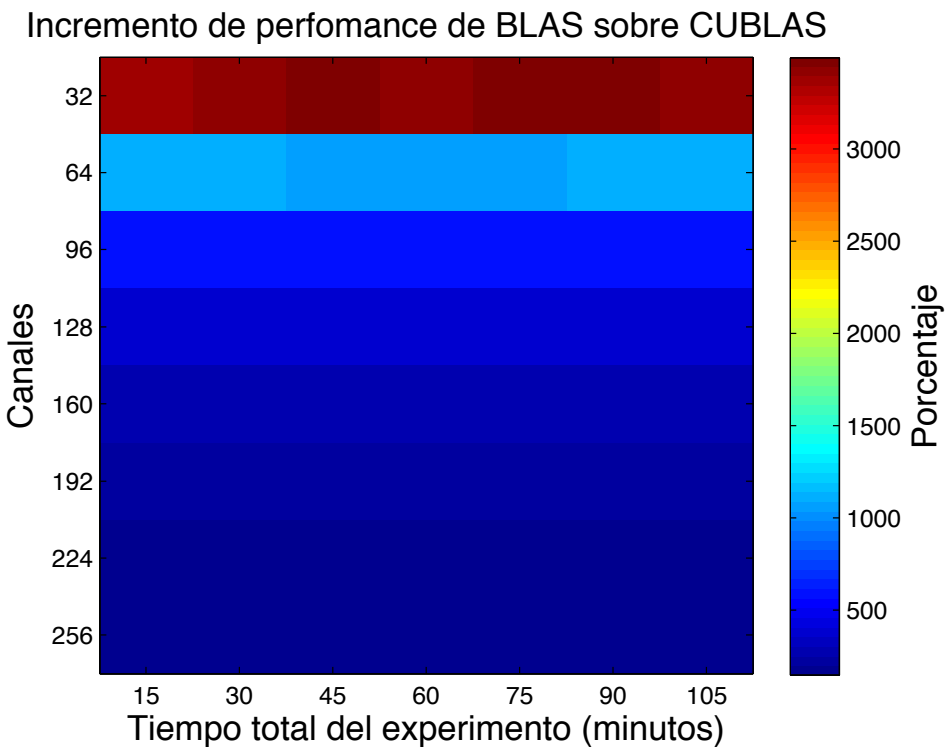


Figura 12: Porcentaje de incremento de performance de BLAS sobre CUBLAS para la operación de matriz-vector. En el eje X están representadas las duraciones de los experimentos en intervalos de 15 minutos. En el eje Y se representan la cantidad de canales.

Este tipo de resultados ha sido visto previamente [FSH04] donde multiplicaciones de matrices en GPU puede resultar incluso más lentas que en implementaciones en CPU con conocimiento del cache.

Por estos motivos se procedió a implementar este algoritmo en GPU (basado en CUDA) re-implementando sin el uso de bibliotecas estándar, teniendo particular atención en la estructura del programa y la arquitectura donde sería ejecutada.

6. Implementación de Cudaica

En esta sección se describe la solución implementada en *CUDA* sin la utilización de las rutinas de *CUBLAS* así como las decisiones tomadas y las implementaciónes realizadas.

Siguiendo los análisis de resultados obtenidos en la secciones 5.1 y 5.2, se determinó que tanto las multiplicaciónes de matriz-matriz y matriz-vector debían ser optimizadas para obtener un incremento en la performance.

6.1. Matriz-Vector en *CUDA*

Los resultados arrojados por *CUBLAS* para las operaciones de matriz-vector no presentaron mejora respecto a las ejecuciones en procesador. Un análisis de la arquitectura realizado a partir de lo descripto en 4.5, utilizando un *profiler* provisto por *Nvidia*[5] determinó que el motivo por el que no se obtuvieron ventajas utilizando la biblioteca *CUBLAS* son los accesos a memoria y ocupación de la placa gráfica. En la figura 13 se muestran los accesos a la memoria global (*dram reads*), lecturas y escrituras en memoria compartida (*shared load/store*) y *warps* activos por ciclo (*active warps/active cycle*) para un caso de test en el que se comparan las operaciones de matriz-vector que se deberían realizar para procesar ICA en un EEG de 128 canales, 30 segundos de duración, utilizando *CUDA* y *CUBLAS*. Como puede observar se detecta una cantidad de *warps* activos muy baja (menor al 12 %) y una elevada cantidad de accesos a memoria global, así como la baja utilización de memoria compartida.

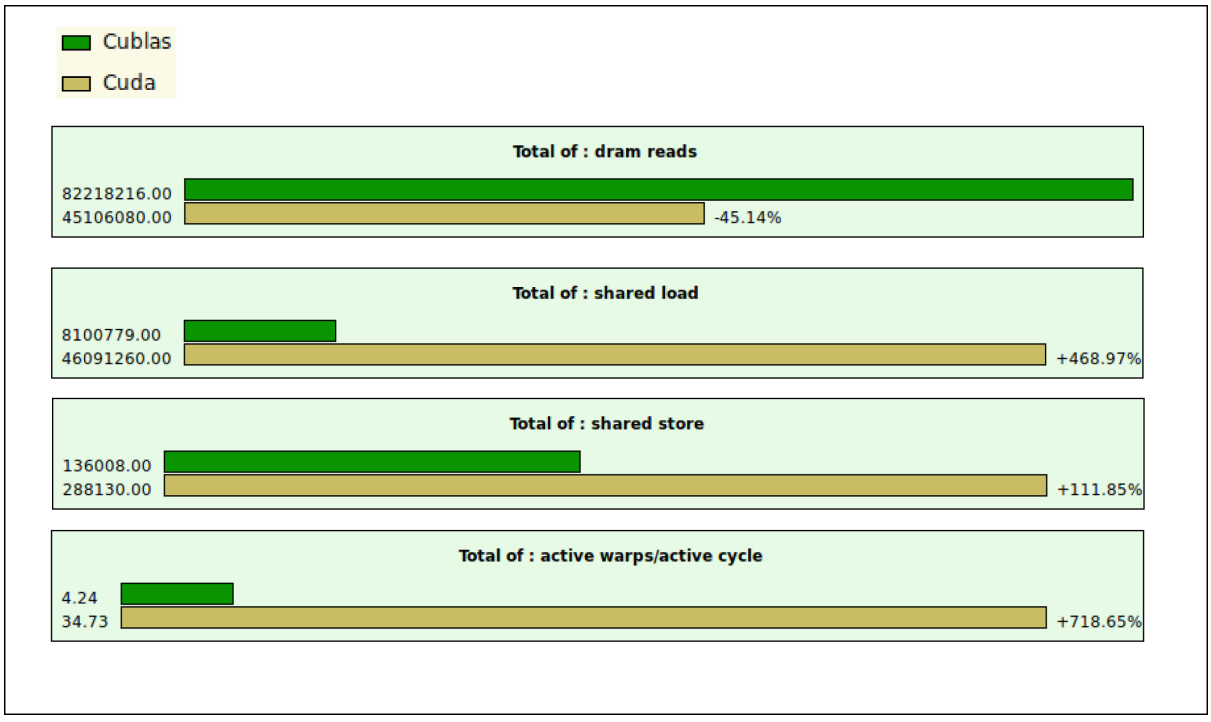


Figura 13: Resultados del análisis de la ejecución de matriz vector en *CUDA* y *CUBLAS*

[5] `http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/VisualProfiler/computeprof.html`

A partir de este análisis se decidió implementar todas las operaciones de matriz-vector y matriz-matriz utilizando memoria compartida para optimizar los accesos a memoria y buscar, de esta forma, a partir de la utilización de memoria alineada, un incremento en la perfomance. Además, se buscó minimizar el uso de los recursos compartidos para la optimización de la ocupación, muy dependiente de los recursos utilizados. Cada multiplicación matriz-vector, antes de realizar las operaciones, copia desde memoria global a memoria compartida la muestra correspondiente (ver código 14).

```
/*
 * Arreglo para guardar la muestra en memoria compartida
 */
__shared__ float sample[N];

/*
 * Multiplica la matriz A (M x N) y el vector B.
 * El numero de bloque representa la columna de B.
 * El numero de thread representa la fila de A.
 */
__global__ matvec(float * A, float * B, float * resultado) {

    // Copia alineada y secuencial la columna a memoria compartida
    sample[threadIdx.x] = B[blockIdx.x * blockDim.x + threadIdx.x]

    __syncthreads();

    float valor = 0.0f;

    for (int i = 0; i < gridDim.x; i++) {
        // Acceso alineado a A
        // Acceso sin conflicto de bancos a columna (todos los threads al
            mismo elemento)
        valor += A[i * blockDim.x + threadIdx.x] * sample[i];
    }
    resultado[blockIdx.x * blockDim.x + threadIdx.x] = valor;
}
```

Ejemplo 14: Multiplicación matriz-vector optimizada con memoria compartida.

Notar que, en el código, la variable `blockDim` sirve para indicar la dimensión del segundo parámetro. En caso de ser 1 corresponde a un vector; en caso de ser N corresponde a una matriz de NxN.

A modo de prueba, se realizaron una serie de multiplicaciones de matriz por vector utilizando BLAS y la nueva implementación. En este experimento, se utilizaron dimensiones de matrices comunes a las realizadas por el algoritmo de cómputo de ICA. La matriz, cuadrada, tenía tantas filas y columnas como canales en un EEG. Comenzando en 32 y finalizando en 256 canales en intervalos de 32. Esta matriz se multiplicó por permutaciones aleatorias de muestras de experimentos de distintas longitudes: desde 15 hasta 105

minutos en intervalos de 15 minutos a 512 Hz. Estó significa desde 460800 muestras hasta 3225600 en intervalos de 460800. Los resultados se muestran en la figura 15.

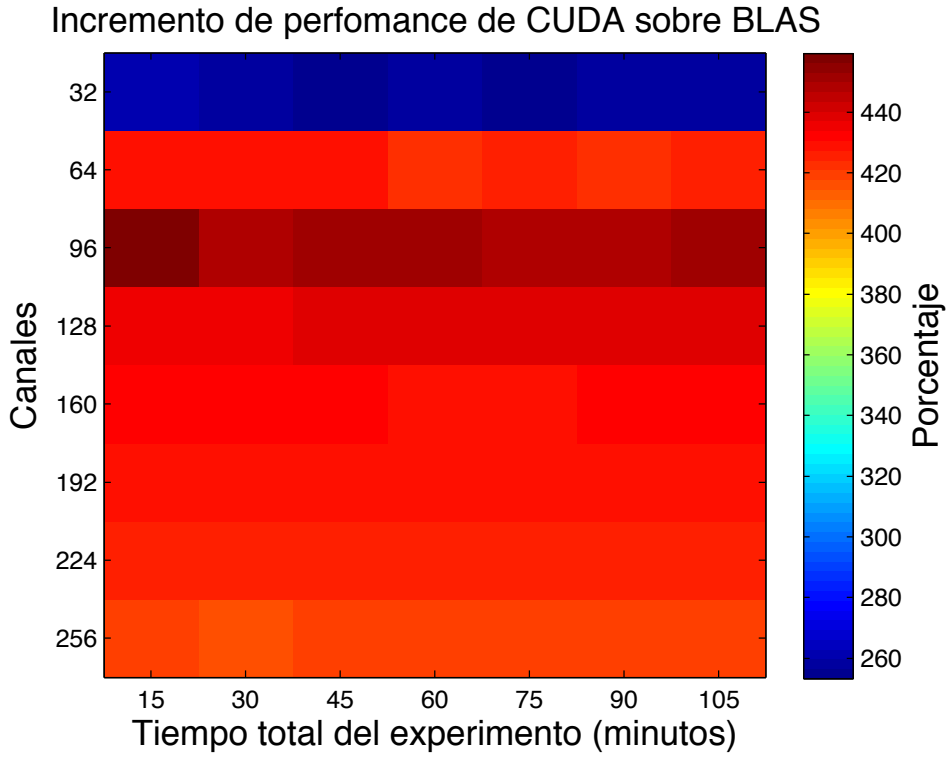


Figura 15: Porcentaje de incremento de performance de *CUDA* sobre *BLAS* para la operación matriz x vector. En el eje X están representadas las duraciones de los experimentos en intervalos de 15 minutos. En el eje Y se representan la cantidad de canales.

Se puede observar un incremento en la perfomance de alrededor del 260 % para un experimento de 32 canales. Este número asciende a 430 % para mayores cantidades de canales, obteniendo un máximo en 460 % para 96 canales. No se observan variaciones significativas en cuanto a la longitud del experimento.

El hecho de que 32 canales no presente un incremento significativo con respecto al resto de las dimensiones, no es una mera coincidencia: en la sección 4.2 se describe la métrica de la ocupación. Esta métrica se le asocia a cada *kernel* ejecutado en el dispositivo. La figura 16 muestra la métrica asociada a la operación de matriz-vector. Se puede observar que para 32 canales sólo un 17 % del dispositivo estaría ocupado.

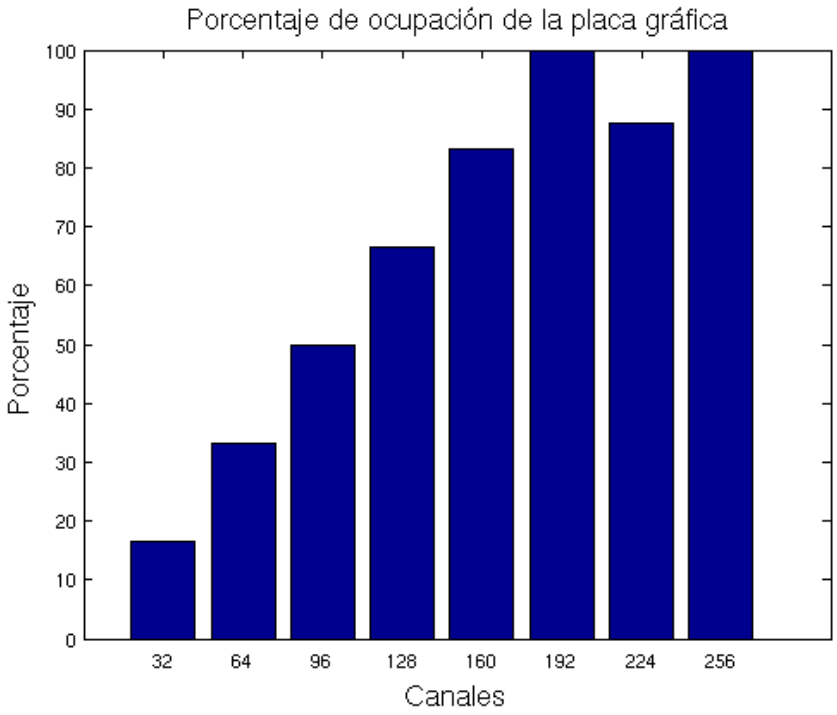


Figura 16: Porcentaje de aprovechamiento de los recursos de la placa gráfica.

Recordar que para este tipo de operación (matriz-vector) la biblioteca estándar *BLAS* se comportó con mejor rendimiento que *CUBLAS*.

6.2. Matriz-Matriz en *CUDA*

Para las operaciones de matriz-matriz se utilizó la misma optimización descripta en la sección anterior, pero sin utilizar una permutación aleatoria de las muestras, dado que el segundo operando también se considera una matriz. La dimensión del primer operando se mantuvo igual que en las pruebas realizadas sobre la multiplicación de matriz por vector: una matriz cuadrada con la cantidad de canales como dimensión. Como segundo operando, se utilizó la matriz generada por todas las muestras del experimento. Los resultados se muestran en la figura 17.

En este caso se puede observar un incremento aún mayor para 32 canales: 350-380 %. En 96 canales alcanza un máximo de 485 %. A diferencia de la operación matriz-vector, en este caso se puede observar una pequeña disminución de la performance a medida que aumenta la cantidad de canales, siempre con una perfomance mayor a 4x.

6.3. Integración con EEGLAB

Adicionalmente, se modificó brevemente el plugin EEGLAB para Matlab a modo de presentar, entre las opciones para realizar el análisis ICA, la opción para utilizar *cudaica* como el método para realizar dicho análisis. De esta forma los usuarios de EEGLAB pueden, en forma transparente, realizar el análisis de ICA en GPU.

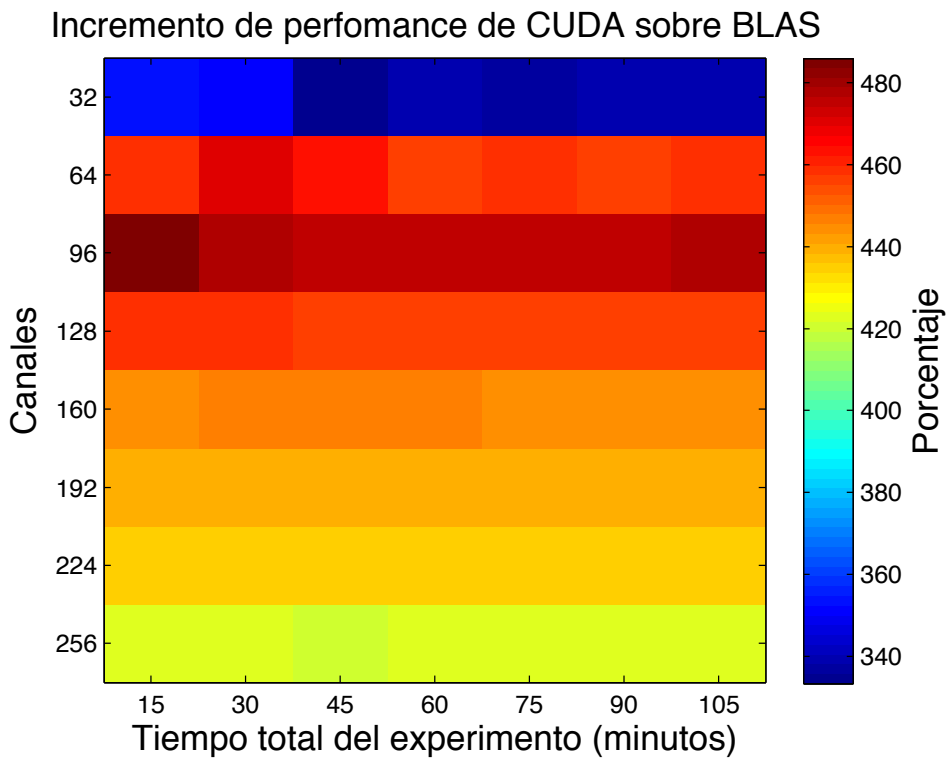


Figura 17: Porcentaje de incremento de performance de *CUDA* sobre *BLAS* para la operación matriz x matriz. En el eje X están representadas las duraciones de los experimentos en intervalos de 15 minutos. En el eje Y se representan la cantidad de canales.

7. Resultados

Para verificar la performance obtenida aplicando las optimizaciones descriptas en la seccion 6 se realizaron dos experimentos.

El primer experimento consistió en probar la performance de las distintas implementaciones utilizando tres GPUs distintas y tres CPUs. Las GPUs utilizadas son *Nvidia* Tesla C2070, *Nvidia* Quadro 4000 y una *Nvidia* GTX 480:

Modelo	Memoria (Mb)	Nucleos	Clock (Mhz)	GFLOPS[6]	Ver. Cuda
GTX 480	1536	15x32	700	168	2.0
Quadro 4000	2048	8x32	475	243.2	2.0
Tesla C2070	6144	14x32	1150	515.2	2.0

Cuadro 3: Comparación de GPUs utilizadas

Para realizar la comparación, se utilizaron los siguientes tres procesadores:

Modelo	Nucleos	Clock (Mhz)	GFLOPS[7]
Intel Core i7-2600	4x2	3400	50.36
Intel Core i5-750	4x1	2660	42.56
Intel Core2 Quad 9400	4x1	2660	42.56

Cuadro 4: Comparación de CPUs utilizados

Se utilizó un set de datos de un experimento real, de 128 canales. La duración del experimento varía desde 400 segundos hasta 2000 segundos en intervalos de 400 segundos. En la figura 18 se puede observar que el algoritmo implementado en *CUDA* presenta un incremento de 6 veces sobre el tiempo utilizado por el procesador i5 y 3.5 veces sobre el tiempo en el procesador i7. Este incremento se mantuvo constante con respecto a la duración del EEG.

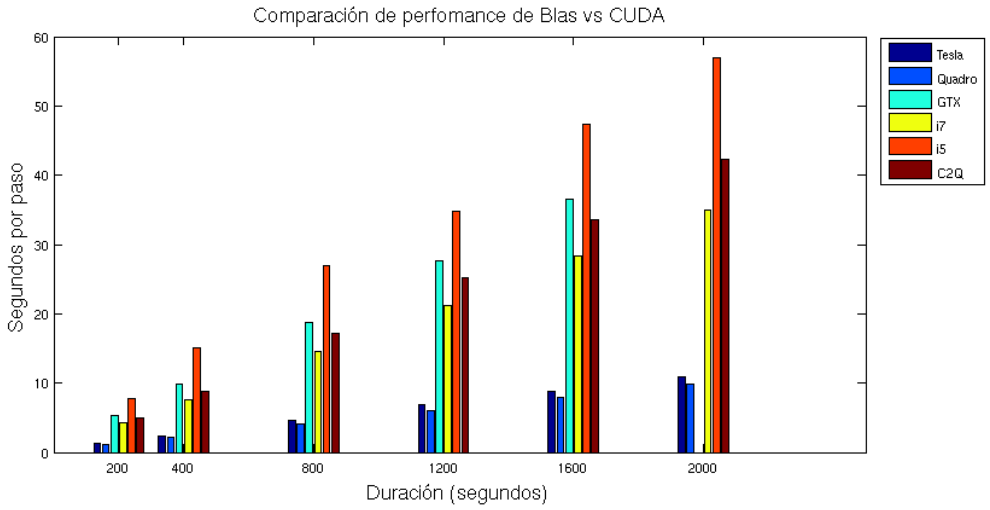


Figura 18: Detalle de tiempos obtenidos en la comparación de performance de *BLAS* y *CUDA*

El segundo experimento se realizó para analizar el impacto de las características del EEG con respecto a la performance. Se compararon los resultados obtenidos de procesar EEGs desde 32 canales hasta 256 en intevalos de 32, desde 15 minutos de duración (460800 muestras) hasta 150 minutos de duración (4608000 muestras) en intervalos de 15 minutos.

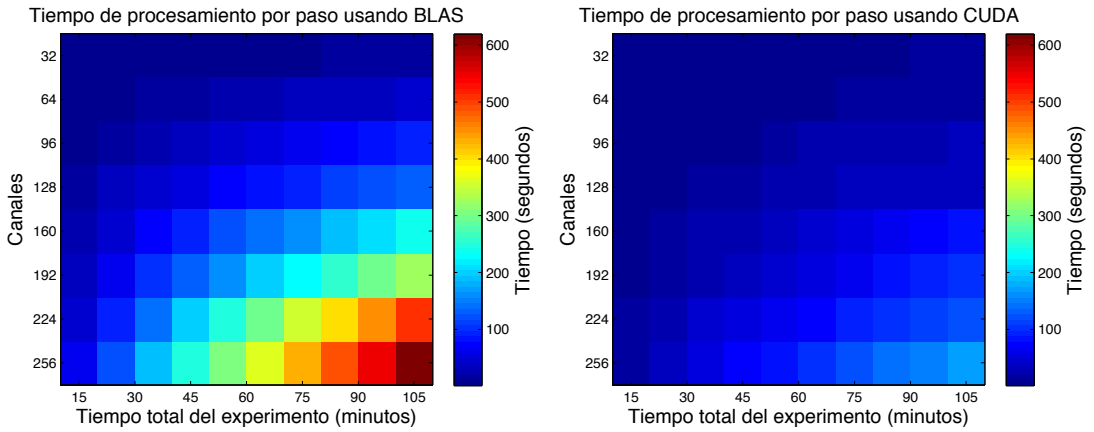


Figura 19: Tiempo de procesamiento de ICA usando *BLAS* y *CUDA*

Como se puede observar en la figura 19, los tiempos de procesamiento por paso utilizando *CUDA* presentan una distribución más uniforme con respecto a *BLAS* a medida que se ve incrementada la duración del experimento y la cantidad de canales. Utilizando 512 pasos como la cantidad máxima de iteraciones a realizar (valor por defecto), la diferencia para un EEG de 256 canales y 105 minutos de duración es de aproximadamente 63 horas de procesamiento: *BLAS* requiere de 87 horas y 14 minutos mientras que *CUDA* utiliza sólo 24 horas 9 minutos.

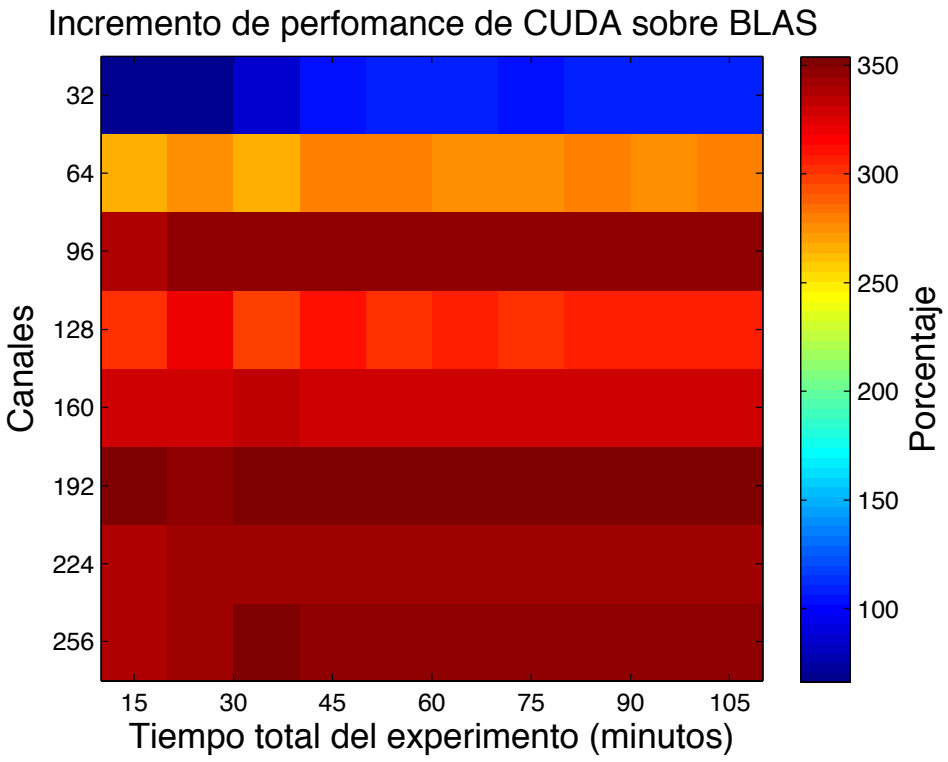


Figura 20: Incremento de perfomance de ICA en *CUDA* sobre *BLAS*

En la figura 20 se muestran los resultados del experimento. Se puede observar un incremento cercano al 350 % para un EEG de 96 canales o más. Esto indica que el tiempo

requerido por la implentación usando *BLAS* en un procesador i7 es 4.5 veces el tiempo utilizado por una *Nvidia* Tesla o *Nvidia* Quadro 4000.

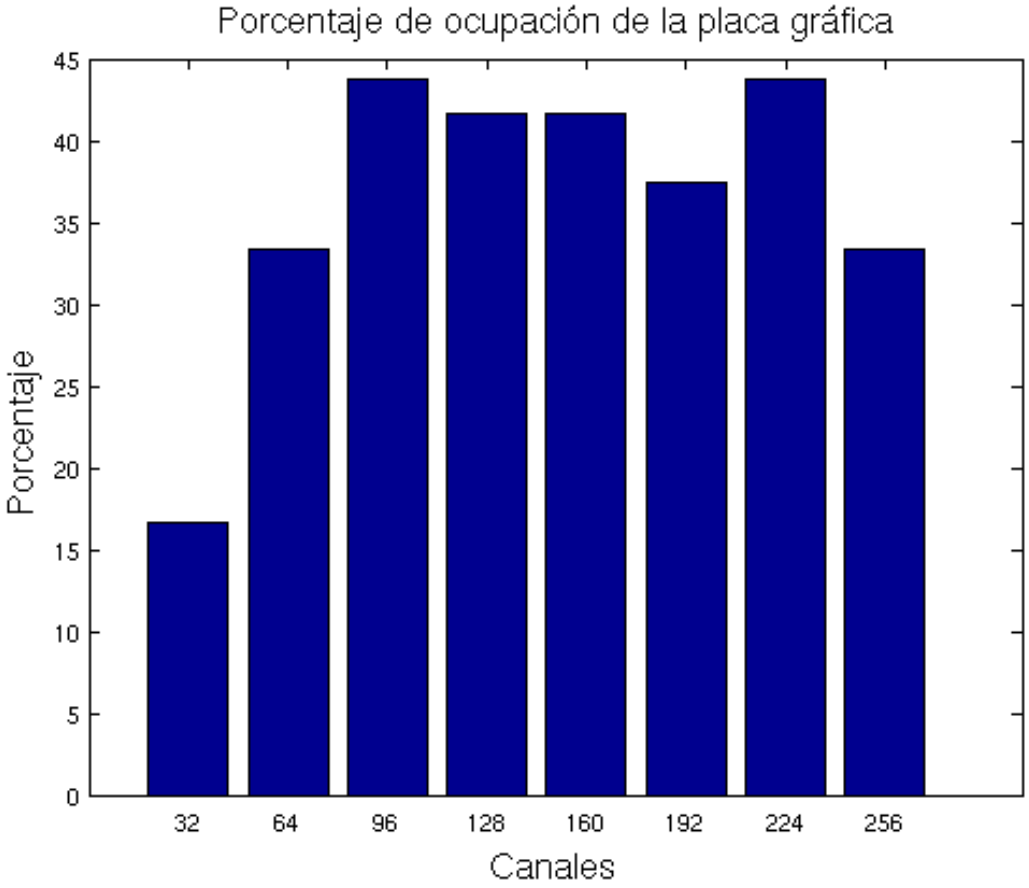


Figura 21: Porcentaje de aprovechamiento de los recursos de la placa gráfica.

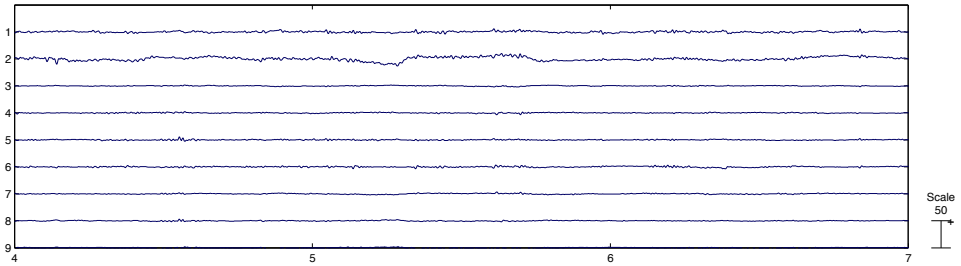
La baja performance obtenida para 32 y 64 canales se explica mediante la ocupación. La figura 21 muestra el porcentaje de aprovechamiento de los recursos de la placa gráfica en una de las operaciones que se realizan con mayor frecuencia en el procesamiento del algoritmo. Si bien esto no representa el algoritmo completamente, la operación analizada representa un porcentaje cercano el 40 % de la ejecución del algortimo.

7.1. Comparación con el método en CPU

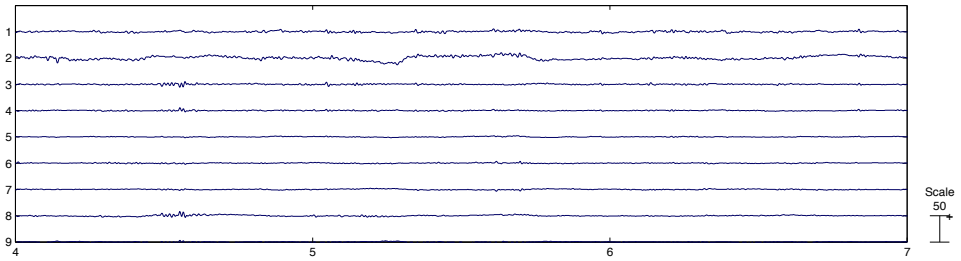
Esta optimización, desde un primer momento, fue pensada para realizar exactamente el mismo algoritmo que su versión en CPU usando *BLAS*. Para verificar su correcto funcionamiento se desactivaron, en ambas versiones, las permutaciones aleatorias y se compararon los resultados obtenidos de procesar un experimento de EEG en *CUDA* y en *BLAS*. No se obtuvieron diferencias al respecto.

Otro experimento de verificación consistió en procesar, para un EEG de 132 canales y 1211904 muestras, el algoritmo ICA implementado en *BLAS* y en *CUDA*. Se compararon

manualmente las componentes y se detectaron diferencias consistentes con la aleatoriedad del análisis que no resultan significativas en el resultado del análisis.



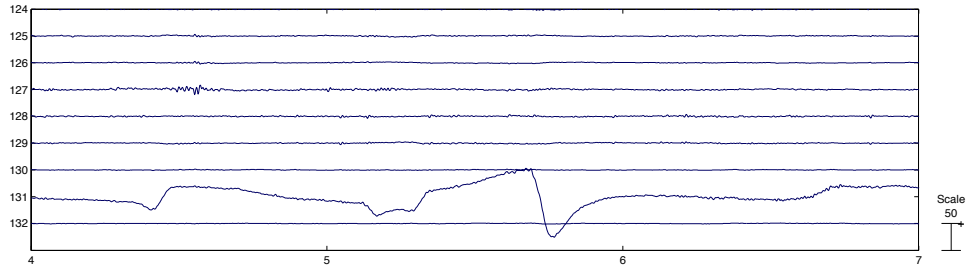
(a) Componentes 1 a 8 calculadas usando *CUDA*



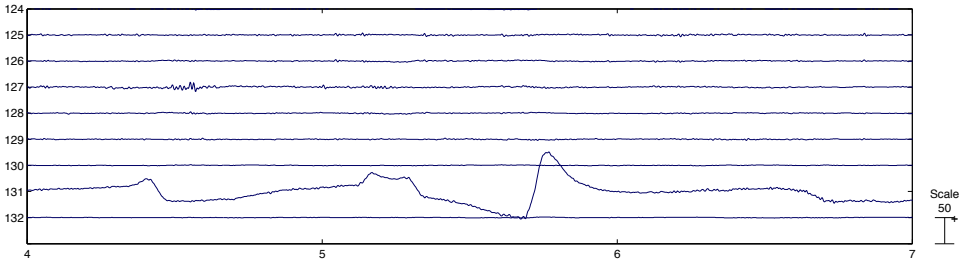
(b) Componentes 1 a 8 calculadas usando *BLAS*

Figura 22: Primeras componentes independendientes utilizando *CUDA* y *BLAS*.

En las figuras 22 y 23 se pueden observar los resultados obtenidos utilizando la implementación original (*BLAS*) y la optimización en *CUDA*. En ambos ejemplos se puede ver que las componentes no son exactamente iguales: el orden y los signos pueden variar. En la figura 22 se puede ver que la componente 2 coincide, la numero 5 de *CUDA* aparece en la cuarta posición en *BLAS* y no hay coincidencias en el resto. En la figura 23 se puede observar que la componente 131 varía su signo dependiendo de la implementación.



(a) Componentes 125 a 132 calculadas usando *CUDA*



(b) Componentes 125 a 132 calculadas usando *BLAS*

Figura 23: Últimas componentes independendientes utilizando *CUDA* y *BLAS*.

8. Conclusiones

La obtención de muestras de señales análogicas siempre ha buscado que las muestras registradas sean lo más parecido a las originales. Sin embargo, este tipo de señales cuentan inherentemente con ruido, interferencias, etc. ICA permite separar las señales observadas con el fin de obtener las señales originales y así poder detectar las fuentes originales generadoras de la señal, permitiendo separar aquellas de particular interés, o la eliminación artefactos no deseados.

El análisis de componentes independientes es un método de gradiente ascendente que se basa en la información de cada una de las muestras observadas y su independencia estadística. El procesamiento resulta muy costoso, al punto de que en ocasiones se prefiera bajar la precisión del cálculo realizado con el motivo de acelerar el procesamiento. El análisis de un experimento de una hora puede requerir 72 horas de procesamiento. La mayor parte del procesamiento consiste en realizar miles de operaciones aritméticas de punto flotante: multiplicaciones de matrices y vectores.

Las placas de video, exigidas por la industria de los juegos de video, son una tecnología en constante avance. La tendencia hacia gráficos más realistas, modelos con más detalles y pantallas con resoluciones más altas, llevan a que no sólo se incremente la velocidad de los procesadores incluidos en la placa gráfica sino también la cantidad de estos que incluyen.

Las operaciones realizadas por estas aceleradoras gráficas son, principalmente, operaciones de multiplicación de vectores y matrices con números de punto flotante, muy útiles para el álgebra lineal. Originalmente, este poder de procesamiento quedaba restringido a operaciones sobre gráficos. Utilizando *CUDA*, ahora, es posible aprovechar este hardware específico para optimizar algoritmos de propósito general que incluyan código con una fuerte dependencia del álgebra lineal. *CUDA* presenta una nueva opción para el incremento de performance: en sólo 4 años ha mostrado un nuevo horizonte en materia de optimización. Las últimas placas, preparadas exclusivamente para cómputo, obtienen una gran performance, comparable con un cluster, a un costo económico mucho menor.

Actualmente, ICA está implementado utilizando la biblioteca *BLAS*, que con mas de 30 años de desarollo se comporta de forma muy eficiente. Con la aparición de *CUDA*, una alternativa a *BLAS* es la desarrollada por *Nvidia*: *CUBLAS*, que opera sobre las placas de video, aprovechando su poder de cómputo. El uso de estas bibliotecas es muy sencillo, siendo necesario únicamente el cargado de memoria de la placa y el renombre de las funciones de cómputo, reduciendo significativamente el tiempo necesario para implementar usando *CUDA* los algoritmos que ya se encuentran desarrollados utilizando *BLAS*.

Sin embargo, esta sencillez lleva acarreado un posible uso sub-óptimo de los recursos disponibles en las placas, provenientes del espíritu de propósito general de estas bibliotecas. En tal sentido, esta tesis analizó detalladamente el uso de los recursos por parte de esta biblioteca en la optimización del método ICA, muy utilizado para la separación de señales en audio, EEG, etc. Se detectaron puntos particulares del método estudiado donde podían realizarse optimizaciones específicas.

Al análisis de perfomance y utilización de placa gráfica sugirió una mejora posible realizando las operaciones aritméticas utilizando características mas avanzadas de la arquitectura *CUDA*. Al tener más información sobre las características de las matrices y vectores con los que se opera, se puede dejar de lado las optimizaciones genéricas de

CUBLAS y aprovechar al máximo los recursos de la placa gráfica.

Utilizando datos obtenidos de EEG, se compararon los tiempos de procesamiento de ICA utilizando *BLAS*, *CUBLAS* y *CUDA*. *CUBLAS* no mostró mejoras aunque tampoco demostró un decremento en la performance. La implementación realizada en *CUDA* obtuvo mejoras en el tiempo de procesamiento 4.5 veces menor comparando una placa gráfica de procesamiento con un CPU de última generación.

9. Trabajos a futuro

La propuesta de optimización presentada en esta tesis lleva a una reducción significativa en el tiempo de procesamiento del análisis de componentes independientes. Aún durante el desarrollo de esta tesis, la arquitectura *CUDA* continuó evolucionando. Nuevas características fueron agregadas, genererando la posibilidad de realizar mejoras aún mas específicas y eficientes [NVI]. Utilizar estas nuevas versiones de la arquitectura presenta un desafío: bajar considerablemente los tiempos de procesamiento hasta llegar al procesamiento de las señales en tiempo real.

Por otro lado, en este trabajo se buscó conservar exactamente el mismo algoritmo que el original, implementado en *CUDA*. Sin embargo, este método toma muestras de un vector aleatorio que luego es multiplicado por una matriz. Para aumentar la performance es posible modificar levemente este método, tomando varios vectores contiguos que permitan realizar más multiplicaciones en paralelo, por lo que podrían necesitarse menos iteraciones del método para converger.

Los últimos experimentos realizados muestran que las operaciones de multiplicaciones de matrices pueden realizarse en forma mas eficiente aún. Para algunos casos, los paquetes de *CUBLAS* resultaron más eficientes que la implementación presentada para la multiplicación de matriz-matriz. Otras investigaciones [HCH03] muestran nuevas opciones para optimizar este operación haciendo uso de la cache de memoria. Una posible mejora de la actual implementación sería una solución híbrida que utilice parte de lo desarrollado en conjunto con las operaciones que son eficientes de la biblioteca *CUBLAS* o haciendo un uso mas eficiente de todas las ventajas que presenta la arquitectura *CUDA*. Para ello será necesario reorganizar la información en la meoria de la placa de video, para obtener los alineamientos deseados y, a la vez, poder realizar las llamadas a la biblioteca *CUBLAS*.

Por otra parte, ICA es un método que detecta las componentes independientes en cualquier tipo de señal. Su utilización se extiende mas allá de los electroencefalogramas como el magnetoencefalograma y la resonancia magnética funcional (fMRI) [SKL+10]. Luego del análisis de ICA, los resultados obtenidos se pueden utilizar con diversos fines, entre ellos la detección y remoción de artefactos. Recientemente, se han desarrollado soluciones automáticas para estos [MJBB10, NWR10], lo que podría significar un avance hacia la utilización de este método con fines clínicos. Estas técnicas poseen características similares a las estudiadas en este trabajo, por lo que permitirían optimizarse con el procesamiento por GPU.

Referencias

[Ama96] S. Amari. A new learning algorithm for blind signal separation. *Advances in Neural Information Processing Systems*, 1996.

[Ama97] S. Amari. Neural learning in structured parameter spaces-natural Riemannian gradient. In *In Advances in Neural Information Processing Systems*. Citeseer, 1997.

[Ama98] S.I. Amari. Natural gradient works efficiently in learning. *Neural computation*, 10(2):251–276, 1998.

[BS95] A.J. Bell and T.J. Sejnowski. An information-maximization approach to blind separation and blind deconvolution. *Neural computation*, 7(6):1129–1159, 1995.

[CL96] J.F. Cardoso and B.H. Laheld. Equivariant adaptive source separation. *Signal Processing, IEEE Transactions on*, 44(12):3017–3030, 1996.

[Com94] P. Comon. Independent component analysis, a new concept? *Signal processing*, 36(3):287–314, 1994.

[CTW+91] T.M. Cover, J.A. Thomas, J. Wiley, et al. *Elements of information theory*, volume 306. Wiley Online Library, 1991.

[DL95] N. Delfosse and P. Loubaton. Adaptive blind separation of independent sources: a deflation approach. *Signal Processing*, 45(1):59–83, 1995.

[Fri87] J.H. Friedman. Exploratory projection pursuit. *Journal of the American statistical association*, 82(397):249–266, 1987.

[FSH04] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 133–137. ACM, 2004.

[Har76] H.H. Harman. *Modern factor analysis*. University of Chicago Press, 1976.

[HCH03] J.D. Hall, N.A. Carr, and J.C. Hart. Cache and bandwidth aware matrix multiplication on the gpu. Technical report, University of Illinois, 2003.

[HO00] A. Hyvärinen and E. Oja. Independent component analysis: algorithms and applications. *Neural networks*, 13(4-5):411–430, 2000.

[HOK01] A. Hyvärinen, E. Oja, and J. Karhuen. *Independent component analysis*. Wiley-Interscience, 2001.

[Hyv02] A. Hyvarinen. Fast and robust fixed-point algorithms for independent component analysis. *Neural Networks, IEEE Transactions on*, 10(3):626–634, 2002.

[JMM+02] T.P. Jung, S. Makeig, M.J. McKeown, A.J. Bell, T.W. Lee, and T.J. Sejnowski. Imaging brain dynamics using independent component analysis. *Proceedings of the IEEE*, 89(7):1107–1122, 2002.

[JS87] MC Jones and R. Sibson. What is projection pursuit? *Journal of the Royal Statistical Society. Series A (General)*, 150(1):1–37, 1987.

[KHFM06] D.B. Keith, C.C. Hoge, R.M. Frank, and A.D. Malony. Parallel ICA methods for EEG neuroimaging. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, page 10. IEEE, 2006.

[Lau81] S. Laughlin. A simple coding procedure enhances a neuron's information capacity. *Z. Naturforsch*, 36(9-10):910–912, 1981.

[LGS99] T.W. Lee, M. Girolami, and T.J. Sejnowski. Independent component analysis using an extended infomax algorithm for mixed subgaussian and supergaussian sources. *Neural computation*, 11(2):417–441, 1999.

[MJBB10] A. Mognon, J. Jovicich, L. Bruzzone, and M. Buiatti. Adjust: An automatic eeg artifact detector based on the joint use of spatial and temporal features. *Psychophysiology*, 2010.

[MWJ+02] S. Makeig, M. Westerfield, T.P. Jung, S. Enghoff, J. Townsend, E. Courchesne, and TJ Sejnowski. Dynamic brain sources of visual evoked responses. *Science*, 295(5555):690, 2002.

[NVI] NVIDIA. `http://pressroom.nvidia.com/easyir/customrel.do?easyirid=A0D622CE9F579F09&version=live&releasejsp=release_157&xhtml=true&prid=726171`.

[NWR10] H. Nolan, R. Whelan, and RB Reilly. Faster: Fully automated statistical thresholding for eeg artifact rejection. *Journal of neuroscience methods*, 2010.

[Oja92] E. Oja. Principal components, minor components, and linear neural networks. *Neural Networks*, 5(6):927–935, 1992.

[SKL+10] V. Schöpf, CH Kasess, R. Lanzenberger, F. Fischmeister, C. Windischberger, and E. Moser. Fully Exploratory Network ICA (FENICA) on resting-state fMRI data. *Journal of Neuroscience Methods*, 2010.

[Wei08] J. Weidendorfer. Sequential performance analysis with callgrind and kcachegrind. *Tools for High Performance Computing*, pages 93–113, 2008.