UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Adaptación de algoritmos de aprendizaje automático para su ejecución sobre GPUs

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Román Navarro Ponce

Tutor: Jon Ander Gómez Adrián

Curso 2015-2016

Resumen

El objetivo de este proyecto es la implementación y posterior análisis de un algoritmo de Aprendizaje Automático, el algoritmo *Expectation-Maximization* (*EM*), usando *Gaussian Mixtures*. Para ello se implementará una versión de *CPU* en C++ y una versión de *GPU* en C++ usando *CUDA* para paralelizar su ejecución. Se analizarán las diferencias de rendimiento y de implementación de ambas versiones. En el marco teórico se explicará el contexto del algoritmo *EM* así como *CUDA*.

Palabras clave: CUDA, Gaussian Mixtures, Expectation-Maximization algorithm

Abstract

The main objective of this Project is the implemetantion and next analysis of a Machine Learning algorithm, the Expectation-Maximization (EM) algorithm, using Gaussian Mixtures. For this, a version of CPU in C++ and another version of GPU in C++ using CUDA will be developed to parallelize their execution. Different performance and implemetation of both versions will be analyzed. CUDA and EM will be explained in the context of the theorical framework.

Key words: CUDA, Gaussian Mixtures, Expectation-Maximization algorithm

Índice general

Índice de figuras

CAPÍTULO 1
Introducción

1.1 Motivación

En las últimas décadas y más concretamente en los últimos años se ha producido un incremento exponencial en la generación de información. El acceso a la tecnología es cada vez más asequible, y que combinado con la evolución del almacenamiento hace que la necesidad de procesar información sea cada vez mayor.

A diario se almacenan gran cantidad de datos, muchos de ellos no sólo con la finalidad de preservarlos sino también de analizarlos, ya sea para clasificarlos según determinadas similitudes, diferencias o bien para analizarlos buscando un determinado patrón. Los datos almacenados en sí no tienen ningún valor sino se obtiene alguna información útil de ellos, el análisis de un gran conjunto de ellos podría revelar alguna relación que hasta entonces no se había descubierto siendo ésta de importancia.

Esto nos conduce a la necesidad de poder analizar dicha cantidad de datos, ya sean datos meteorológicos, modelos físicos complejos, biológicos, ... Para ello necesitamos ciertos algoritmos especializados de los conocidos como de Aprendizaje Automático ya que debido a la gran cantidad de datos a manejar es necesario desarrollar algoritmos que clasifiquen o encuentren patrones.

La gran cantidad de información generada en los últimos años crece exponencialmente, como ya se ha comentado, pero no así la potencia de cálculo de los ordenadores actuales, que pese a seguir creciendo se acerca cada vez más a la barrera tecnológica que utilizamos actualmente. Por lo que en vez de insertar más transistores en las CPU como hasta hace relativamente poco tiempo, se ha optado por utilizar más *cores* simultáneamente, sirviendo estos para "paralelizar" los cómputos.

Esto nos lleva a la necesidad de paralelizar los algoritmos ya existentes como el mencionado en este proyecto, el algoritmo *EM*, pero no solo paralelizarlo en base a los *cores* de una *CPU*, ya que actualmente y por el desarrollo en gran parte de los gráficos en 3D los ordenadores personales poseen potentes tarjetas gráficas o *GPUs*. Las cuales debido a la naturaleza de los datos que manejan tienen muchos más *cores* que una *CPU* habitual, lo que las convierte en una gran opción para el cálculo paralelo sobre todo por su precio comparado con otras alternativas.

1.2 Estado del arte

Actualmente se pueden encontrar múltiples implementaciones del algoritmo *EM* en una gran variedad de lenguajes de programación. Esto es debido a que el algoritmo fue desarrollado en las décadas pasadas, por ejemplo podemos encontrar tanto una completa implementación como documentación del algoritmo *EM* en Python [1], así como también en R [2], Matlab [3], incluso en Java podemos encontrar una librería ya implementada y lista para utilizar [4]. Sin embargo, estas implementaciones no están centradas en mejorar el rendimiento del algoritmo *EM* sino son de carácter divulgativo y pedagógico.

En referencia a implementaciones para mejorar el rendimiento del algoritmo *EM* de forma paralela podemos encontrar *OpenMP* [5], en la cual se implementa una versión para múltiples *cores* de *CPU*. No obstante, no se indican los resultados ni la aceleración obtenida. Por otro lado existe la posibilidad de usar *OpenCL* [6], consiste en unas librerías que permiten la ejecución en *GPUs* libre de licencias, en dicha implementación los resultados arrojan unas mejoras en torno a 8x de aceleración aunque no especifica los datos técnicos del *dataset* (número de elementos, dimensiones) utilizado o del criterio de convergencia seguido, entre otras características omitidas.

En el aspecto teórico del algoritmo se han desarrollado variantes que intentar corregir ciertas partes del algoritmo *EM*, como por ejemplo la variante de Quasin-Newton [7].

La novedad que intentamos aportar en este proyecto no es solo intentar paralelizar el algoritmo en *GPU* utilizando *CUDA*, es a su vez intentar superar ese 8x mencionado anteriormente sin usar librerías externas de *CUDA* y siendo precisos en la metodología seguida.

CAPÍTULO 2
CUDA

CUDA son las siglas de *Compute Unified Device Architecture* (Arquitectura Unificada de Dispositivos de Cómputo) que hace referencia tanto a un compilador como a un conjunto de herramientas de desarrollo creadas por *NVIDIA* que permiten a los programadores usar una variación del lenguaje de programación C para codificar algoritmos en *GPUs* de *NVIDIA*, aunque también se puede usar *Python*, *Fortran* y *Java* entre otros [8].

En particular, *CUDA* intenta explotar las ventajas de las *GPU*, frente a las *CPU*, de propósito general utilizando el paralelismo que ofrecen sus múltiples núcleos, que permiten la ejecución de un altísimo número de hilos simultáneos. Una *GPU* puede ofrecer un gran rendimiento en gran variedad de campos que podrían ir desde la Bioinformática, pasando por el Aprendizaje Automático hasta el cálculo de Modelos meteorológicos y climáticos [9].

2.1 De tarjetas gráficas a procesamiento paralelo

Guiados por la continua demanda del mercado para procesar modelos más "realistas" de unos gráficos 3D, la *Graphic Processor Unit (GPU)* ha ido evolucionando en un procesador altamente paralelizable, multihilo y *multicore* con una potencia computacional y un ancho de memoria de potencial enorme, a modo de ilustración la Figura 2.1 muestra una comparación entre operaciones en coma flotante por segundo entre *CPU* y *GPU*.

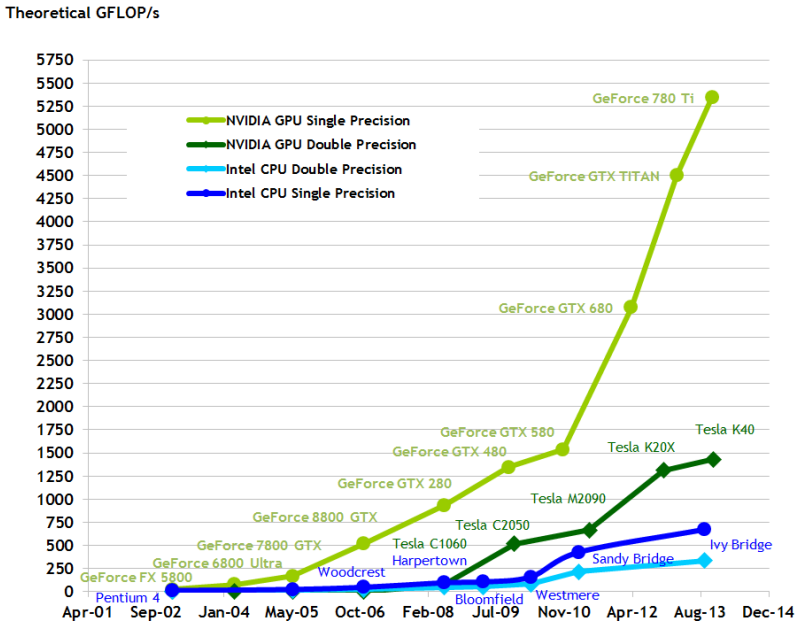

Figura 2.1: Operaciones en coma flotante por segundo CPU/GPU [10]

La razón de las diferencias entre *CPU* y *GPU* en cuanto a la capacidad en coma flotante reside en que la *GPU* está diseñada para una computación intensiva, y altamente paralela, que es exactamente lo que se necesita para *renderizar* gráficos 3D. Su tarea desde que los gráficos 3D aparecieron en los *PCs*, lo que lleva a la *GPU* a tener mas transistores para el procesamiento de datos que una *CPU*, tal como se ve en la Figura 2.2.

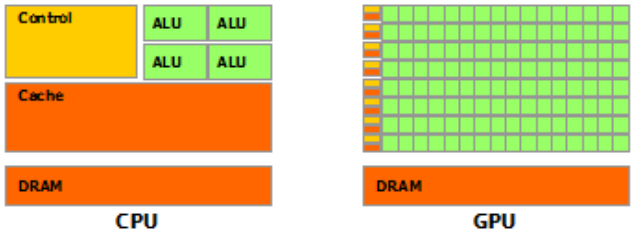

Figura 2.2: Comparación interna CPU-GPU

Concretando más, la *GPU* está diseñada para problemas en el que el programa está siendo ejecutado en muchos elementos a la vez y en paralelo con una alta intensidad aritmética. Muchas aplicaciones que procesen conjuntos de datos pueden usar el modelo de programación paralela para acelerar los cálculos, por ejemplo, en 3D *rendering* grandes conjuntos de *pixeles* y vértices son mapeados en hilos paralelos.

2.2 Un modelo de programación escalable

La ventaja de las *CPUs multicore* y de las *GPUs manycore* actuales reside en que los chips de los procesadores son en sí sistemas paralelos, el reto actualmente reside en desarrollar software que escale este paralelismo de manera transparente al número de procesadores/*cores*, como una aplicación 3D escala su paralelismo para *GPUs* con un gran y variado número de *cores*.

El programador para usar estas ventajas deberá "dividir" el problema en subproblemas que puedan resolverse independientemente unos de otros en paralelo divididos en bloques de hilos y cada subproblema en pequeños trozos que puedan ser resueltos cooperativamente en paralelo por los hilos del bloque.

Esta descomposición permite que los hilos cooperen cuando resuelven los subproblemas y al mismo tiempo activa la escalabilidad automáticamente, cada bloque de hilos puede ser programado en cualquiera de los multiprocesadores disponibles en una *GPU*, en cualquier orden, secuencial o concurrente, por eso un programa compilado en *CUDA* puede ser ejecutado en cualquier número de multiprocesadores y sólo el sistema necesita saber el número físico de multiprocesadores [10].

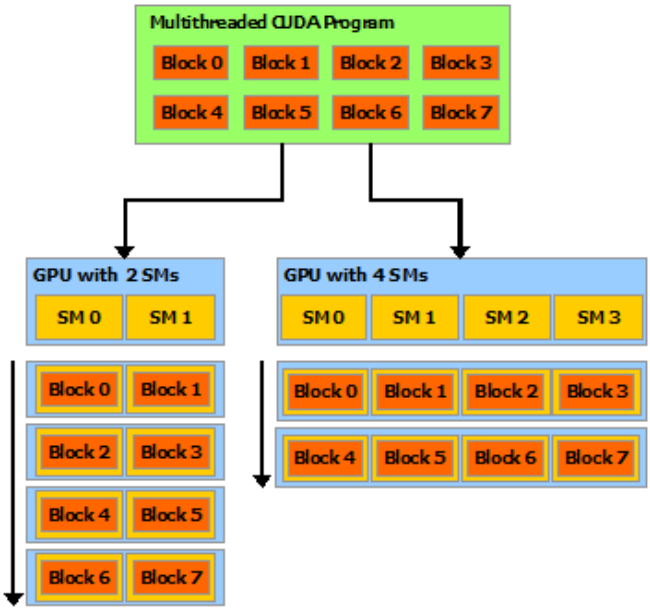

Figura 2.3: Escalabilidad [11]

Para entender totalmente la Figura 2.3, es necesario comprender cómo se comporta a nivel de hardware *CUDA* aunque sea a un nivel básico nos da una idea aproximada de como funciona internamente.

Una *GPU* está formada por un *array de Streaming Multiprocessors (SMs)*, un programa multihilo es partido en bloques de hilos que se ejecutan independientemente unos de otros, por eso una *GPU* con más multiprocesadores ejecutará automáticamente un programa en menos tiempo que una *GPU* con menos multiprocesadores.

La arquitectura de las *GPUs* de *NVIDIA* está formada por un *array* escalable de multihilos *SMs* como hemos comentado anteriormente. Cuando un programa en *CUDA* en el *host CPU* invoca un *kernel grid*, los bloques del *grid* son enumerados y distribuidos a los distintos multiprocesadores disponibles para ejecutar.

Un multiprocesador está diseñado para ejecutar cientos de hilos concurrentemente, para manejar tal tamaño de hilos se emplea una arquitectura denominada *SIMT (Single-instruccion, Multiple-Thread)* que junto al Hardware *Multithreading* forman parte de las características de ingeniería de los *SMs* y que son comunes a todas las *GPUs* de *NVIDIA*.

2.3 El modelo de programación

En esta sección se explicarán las nociones básicas así como los conceptos básicos para programar en *CUDA* siempre ligado a *C*, se explicará a través de ejemplos teóricos y de un caso práctico simple a forma de ejemplo sacado de la documentación de *CUDA* [12].

2.3.1. El Kernel

CUDA extiende C permitiendo al programador definir funciones de C, de nombre *kernels*, que cuando son llamadas son ejecutadas *N* veces en paralelo por *N* diferentes hilos *CUDA*.

Un *kernel* es definido usando la declaración __global__ y el número de hilos de *CUDA* que ejecuta ese kernel es especificado usando la construcción «....», a cada hilo que ejecuta el kernel se le da un valor único o *thread ID* que es accesible dentro del kernel a través de la variable *threadIdx* [13].

Como ejemplo el siguiente código suma dos vectores A y B de tamaño *N* y almacena el resultado en el vector C, el *main* se ejecutaría en el *host* y la función *VecAdd* se lanzaría en el *device*.

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

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Cada uno de los *N* hilos que ejecute `VecAdd()` realiza un par de sumas.

2.3.2. Jerarquía de hilos

Por conveniencia, *threadIdx* puede ser un vector de 3 dimensiones, se representa de este modo para poder identificar hilos en una dimensión, dos o tres que a su vez pueden formar bloques de hilos de una dos y tres dimensiones, así podemos movernos entre los distintos elementos de un vector, matriz o volumen.

Existen límites en el número de hilos por bloque, ya que todos los hilos de un bloque deberían estar en el mismo *core* del procesador y deben compartir los limitados recursos

de memoria de ese *core*, en las actuales *GPUs* un bloque de hilos puede contener hasta 1024 hilos.

Los bloques de hilos se organizan en *grid* de una dos y tres dimensiones como se ve en la Figura 2.4. El número de hilos por bloque en un *grid* depende normalmente de los datos que estemos procesando o del número de procesadores de la *GPU*.

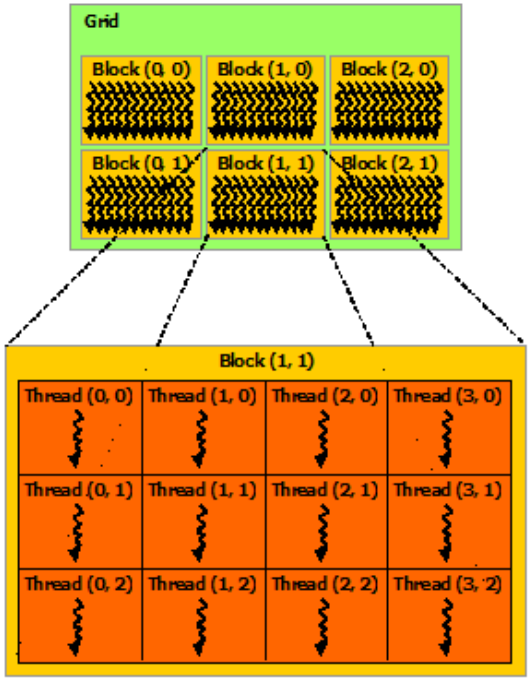

Figura 2.4: Grid de bloques de hilos [11]

El número de hilos por bloque y el número de bloques por *grid* están especificados en la sintaxis propia de *CUDA* y pueden ser de tipo *int* o *dim3*, de igual manera que el ejemplo anterior con *threadIdx*, el número de bloque puede variar de una dos o tres dimensiones, pudiendo acceder a ese dato con *blockIdx*

2.3.3. Jerarquía de memoria

Los hilos de *CUDA* pueden acceder a los datos de múltiples memorias durante su ejecución, cada hilo tiene su memoria local privada, cada bloque tiene una memoria compartida visible por todos los hilos del bloque y todos los hilos tienen acceso a la misma memoria global.

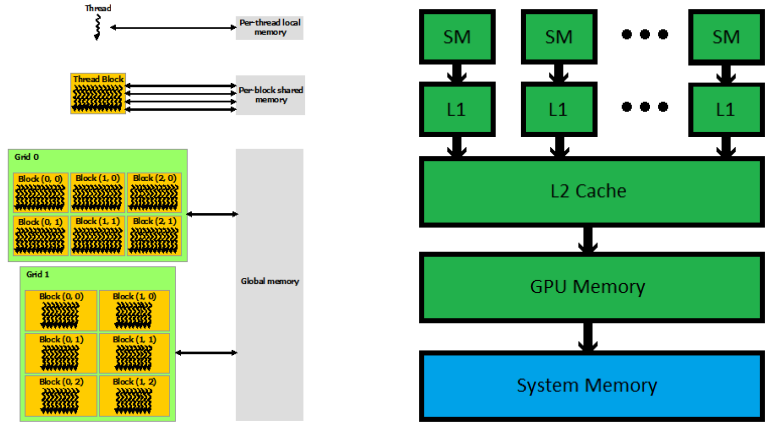

Figura 2.5: Jerarquía de memoria [10]

Este detalle parece obviable sino fuera porque la memoria compartida por bloque es mucho más rápida que la memoria global, utilizarla es vital si queremos conseguir un buen rendimiento en el programa. La capacidad de esta es pequeña (sobre *48kb*) y la latencia depende del modelo concreto en un rango de 3-4 ciclos.

Las *GPUs* modernas de *NVIDIA* (arquitectura *Fermi* o posterior) tienen una cache L2 mucho más pequeña que una típica cache de *CPU* de nivel L2 o L3, pero con un mayor ancho de banda.

La cache L1 de una *GPU* es más pequeña que la de una *CPU*, pero otra vez el ancho de banda es mucho mayor, las gráficas de alta gama de *NVIDIA* tienen varios *SMs*, como ya comentamos en los párrafos pertinentes de la sección 2.2, además cada una está equipada con su propia cache L1. La mayoría de las *CPUs* de gama media-alta tienen alrededor de 6-8 *cores* mientras que una *GPU* de *NVIDIA* puede tener hasta 16 *SMs*, otra aspecto a tener en cuenta es que la cache L1 en *CUDA* no es "coherente" estos significa que dos diferentes *SMs* que estén escribiendo o leyendo de la misma posición de memoria no hay garantía de que un *SM* verá inmediatamente los cambios de otro *SM*, esto es importante ya que es difícil de *debugear* y causa bastantes problemas [14, 15].

2.3.4. La reducción

El objetivo de utilizar *CUDA* para hacer un *reduce* es aprovechar todo el procesamiento paralelo que ofrece, necesitamos ser capaces de usar múltiples bloques de hilos para procesar *arrays* de grandes dimensiones, ésto se consigue de una manera eficiente si mantenemos todos los procesadores de la *GPU* ocupados y cada bloque de hilos reduce una parte del *array*

Uno de los principales problemas para realizar la tarea descrita en el párrafo anterior es que *CUDA* no tiene sincronización global, tendría un alto coste de implementación en el hardware de *GPUs* con una gran cantidad de procesadores, también forzaría al programador a ejecutarlo con menos bloques para evitar el *deadlock*, lo cual reducirá la eficiencia en general. Hay que evitar la sincronización global descomponiendo el cálculo en llamadas múltiples a los *kernels*, como se muestra en la Figura 2.6. Las reducciones tienen una intensidad aritmética muy baja por lo que deberíamos intentar aumentar el pico de ancho de banda [16].

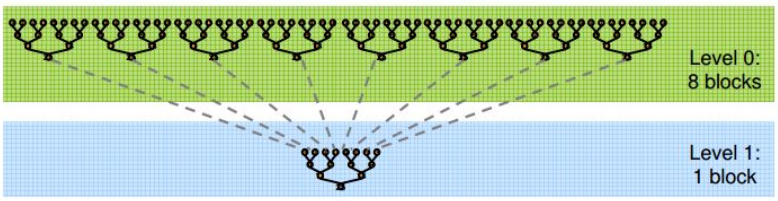

Figura 2.6: Multiple kernel launch [16]

A continuación se explicará los pasos básicos para desarrollar un algoritmo óptimo de *reduce* en *CUDA* siguiendo las recomendaciones de *NVIDIA*, se empezará con una versión muy básica de un *reduce* para paso a paso ir perfeccionándolo, seguir estas recomendaciones tiene un impacto importante en el rendimiento del algoritmo y son una parte destacada en el desarrollo del mismo.

```
// each thread loads one element from global to shared mem
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x
*blockDim.x + threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();
for (unsigned int s=1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0) {
      sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

En este ejemplo básico vemos que el problema reside en que las ramas altamente divergentes producen un rendimiento muy pobre. Podemos simplemente remplazar la parte de código que produce este resultado por una una rama no divergente, pero tendríamos conflictos en la memoria compartida.

```
for (unsigned int s=1; s < blockDim.x; s *= 2) {
    int index = 2 * s * tid;
    if (index < blockDim.x) {
      sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

Intentando mejorar el ejemplo básico podríamos sustituir con un bucle invertido y basarnos en un índice de *threadID*

```
if (tid < s) {
```

En este punto podría funcionar correctamente pero como consecuencia del *if* la mitad de los hilos estarían inactivos en la primera iteración del bucle, el siguiente paso para obtener un mejor rendimiento consistiría en dividir el número de bloques y remplazar la carga por dos cargas y la primera suma de la reducción para evitar este hecho.

```
// perform first level of reduction,
// reading from global memory, writing to shared memory
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
```

```
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

Pese a las mejoras implementadas en el algoritmo básico aún estamos lejos del pico de ancho de banda teórico que nos ofrece la tarjeta A.1, la siguiente estrategia para llegar a ello sería un *unroll* de los bucles, ligado al concepto de *warp* [1]. Esto es debido a que según avanza la reducción el número de hilos activos desciende, teniendo en cuenta que las instrucciones son *SIMD* sincronizadas con un *warp*, esto significa que cuando es menor a 32 no necesitamos sincronizar los hilos, por lo que podemos realizar el *unrolling*.

```
for (unsigned int s=blockDim.x/2; s>32; s>>=1)
{
    if (tid < s)
       sdata[tid] += sdata[tid + s];
    __syncthreads();
}
if (tid < 32)
{
    sdata[tid] += sdata[tid + 32];
    sdata[tid] += sdata[tid + 16];
    sdata[tid] += sdata[tid + 8];
    sdata[tid] += sdata[tid + 4];
    sdata[tid] += sdata[tid + 2];
    sdata[tid] += sdata[tid + 1];
}
```

Si pudiéramos saber el número de iteraciones en tiempo de compilación podríamos hacer un *unroll* de la reducción completa y no solo de los 32 últimos, dado que el tamaño del bloque está limitado por la *GPU* a 512 hilos en potencias de 2. La complicación ahora reside en como podemos hacer un *unroll* de los tamaños de los bloques que no conocemos en tiempo de compilación y ahí es donde entran los *templates*.

```
Template <unsigned int blockSize>
__global__ void reduce5(int *g_idata, int *g_odata)
```

```
if (blockSize >= 512) {
    if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads();
}
if (blockSize >= 256) {
    if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads();
[..]
}
if (tid < 32)
{
    if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
[..]
}
```

Por último debemos reemplazar la carga y la suma de dos elementos visto anteriormente con un bucle *while* y las sumas, dando lugar a un código libre de *invervaling addressing*, hilos "parados" en la primera iteración, y manteniendo la *coalescing*[2] quedando el código de la siguiente manera:

[1] Se denomina warp al conjunto de 32 hilos

[2] Hilos que se ejecutan simultáneamente acceden a bloques contiguos de memoria

```
template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n)
{
extern __shared__ int sdata[];
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockSize*2) + tid;
unsigned int gridSize = blockSize*2*gridDim.x;
sdata[tid] = 0;
while (i < n) { sdata[tid] += g_idata[i] + g_idata[i+blockSize];
i += gridSize; }
__syncthreads();
if (blockSize >= 512) { if (tid < 256) { sdata[tid] += sdata[tid + 256]; }
    __syncthreads(); }
if (blockSize >= 256) { if (tid < 128) { sdata[tid] += sdata[tid + 128]; }
    __syncthreads(); }
if (blockSize >= 128) { if (tid < 64) { sdata[tid] += sdata[tid + 64]; }
    __syncthreads(); }
if (tid < 32) {
    if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
    if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
[..]
```

2.4 Ejemplo didáctico en CUDA

El modelo de programación de *CUDA* asume que los hilos de *CUDA* se ejecutan en un dispositivo (*device*) distinto que opera como coprocesador en cuanto al *host* que es el que ejecuta el programa en sí, es decir, un código englobado en un programa de C, el *main* por ejemplo, se lanzaría en el *host* y cuando se llamara a la función *MatAdd* propia de *CUDA* esta se lanzaría en la *GPU*. Una posible ejecución sería la de la Figura 2.7.

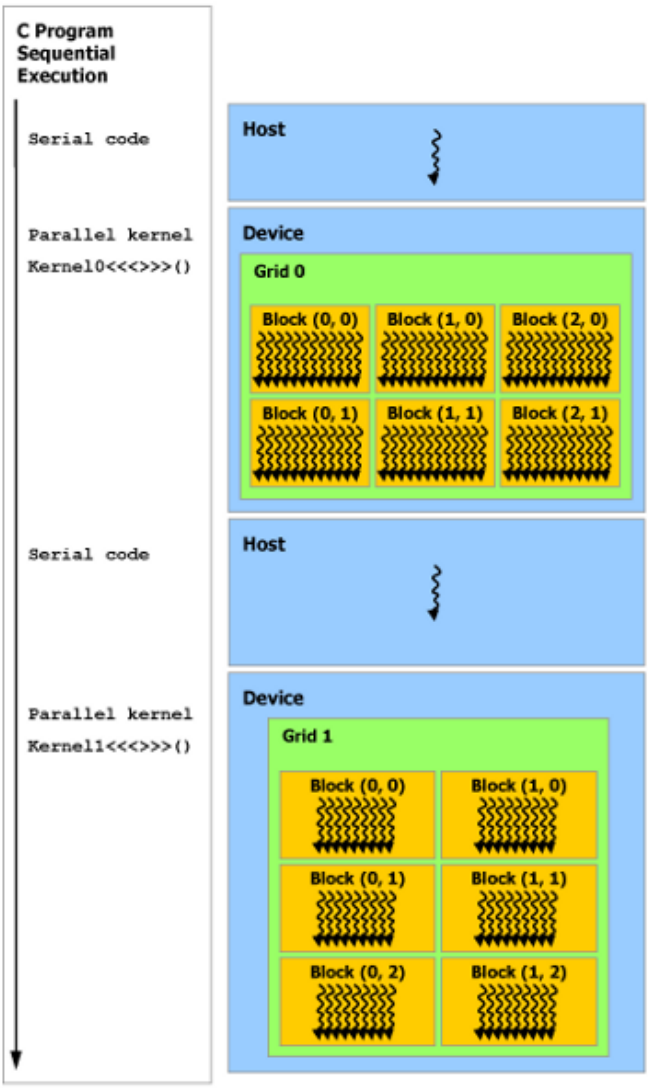

Figura 2.7: Ejecución código host-device

Para ello necesitamos ciertas funciones para reservar,limpiar y copiar memoria del *device* como también para transferir datos entre la memoria del *host* y del *device*, *cudaMalloc*, *cudaFree* y *cudaMemcpy* respectivamente.

2.4.1. Multiplicación de matrices sin memoria compartida

```
// [..]

// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc(&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size,
```

```
13              cudaMemcpyHostToDevice);
14    [..]
15
16    // Allocate C in device memory
17    [..]
18
19    // Invoke kernel
20    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
21    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
22    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);
23
24    // Read C from device memory
25    cudaMemcpy(C.elements, Cd.elements, size,
26              cudaMemcpyDeviceToHost);
27
28    // Free device memory
29    cudaFree(d_A.elements);
30   [..]
31 }
32
33 // Matrix multiplication kernel called by MatMul()
34 __global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
35 {
36    // Each thread computes one element of C
37    // by accumulating results into Cvalue
38    float Cvalue = 0;
39    int row = blockIdx.y * blockDim.y + threadIdx.y;
40    int col = blockIdx.x * blockDim.x + threadIdx.x;
41    for (int e = 0; e < A.width; ++e)
42        Cvalue += A.elements[row * A.width + e]
43                * B.elements[e * B.width + col];
44    C.elements[row * C.width + col] = Cvalue;
45 }
```

En este ejemplo de *CUDA* vamos a multiplicar dos matrices A y B, guardando el resultado en C, parecido a como operaríamos con el lenguaje de programación C, primero debemos reservar memoria para los datos que vamos a manejar pero en este caso en el *device*, reservamos memoria con *cudaMalloc* para A para posteriormente copiar del *host* al *device* la matriz A con *cudaMemcpy*, de igual manera operamos con B. Al almacenar el resultado en C debemos también reservar memoria en el *device* para esa matriz.

Llamamos al *kernel* con *MatMulKernel*, el cual está definido en la linea 34, cada hilo lee una fila de A y una columna de B y computa el correspondiente elemento de C, por lo que A es leída *B.width* veces desde la memoria global y B es leída *A.height* veces. Esto resulta importante ya que una de las principales ventajas de *CUDA* y a su vez desventajas es como está diseñada la jerarquía de memoria en una *GPU* tal como explicábamos en la sección 2.3.3, de una forma parecida a una *CPU*, una *GPU* tiene acceso a distintas memorias que tienen distintas velocidades de lectura/escritura, el programa expuesto anteriormente accede continuamente a una memoria global de la *GPU* que si bien es más rápida que acceder a la memoria del *host* para copiar datos queda por detrás de la memoria compartida.

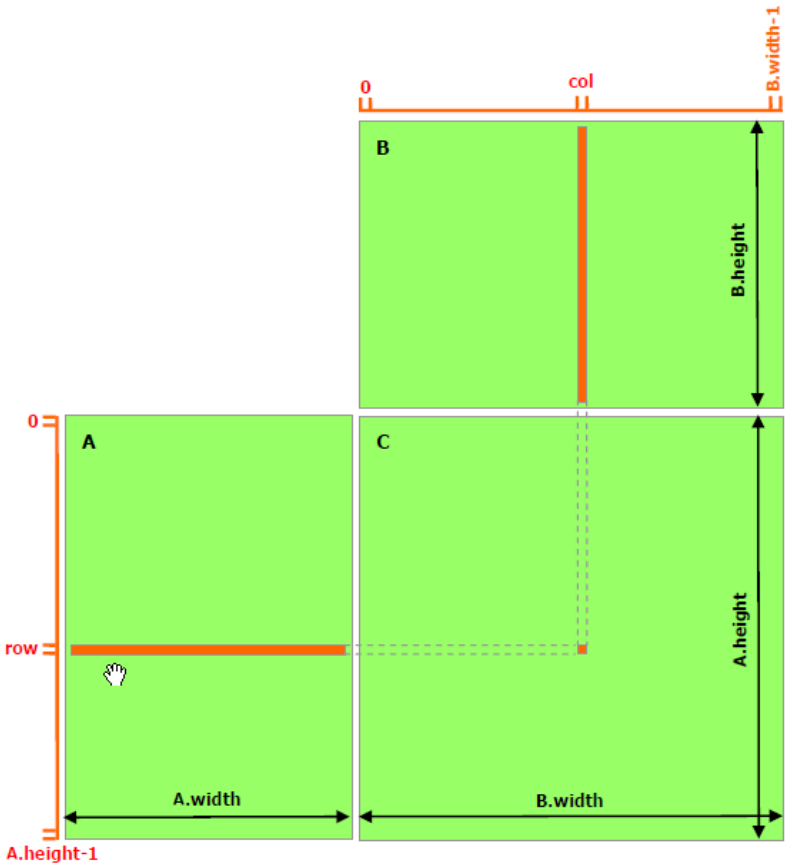

Figura 2.8: Multiplicación sin memoria compartida

En la Figura 2.8 vemos una representación gráfica de como realizaría la ejecución el código expuesto anteriormente sin el uso de memoria compartida.

2.4.2. Multiplicación de matrices con memoria compartida

En la implementación con memoria compartida, cada bloque es responsable de calcular una pequeña porción cuadrada de la sub matriz C_s y cada hilo del bloque calcula un elemento de esa sub matriz cuadrada C_s. C_s es igual al producto de dos matrices rectangulares, la sub matriz de dimensión A, que tiene los mismos índices de fila que C_s y la sub matriz de B que tiene los mismos índices de columna que C_s, para alojar estas matrices en el *device*, estas dos matrices rectangulares son divididas en cuantas matrices cuadradas de dimensión *block_size* necesarias para que finalmente C_s sea calculada como la suma de los productos de esas matrices cuadradas [17].

Cada uno de estos productos se realiza primero cargando las dos matrices cuadradas correspondientes de la memoria global a la memoria compartida donde cada hilo carga un elemento de cada matriz para que cada hilo calcule un elemento del producto, cada hilo acumula el resultado de estos productos en un registro que vuelca esta información en la memoria global.

Haciéndolo de esta manera, A es sólo leída *B.width/block_size* veces desde la memoria global y B es leída *A.height /block_size* veces.

```
1 [..]
2 // Matrix multiplication kernel called by MatMul()
3  __global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
4 {
5     // Block row and column
```

```
6     int blockRow = blockIdx.y;
7     int blockCol = blockIdx.x;
8
9     // Each thread block computes one sub-matrix Csub of C
10    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
11
12    // Each thread computes one element of Csub
13    // by accumulating results into Cvalue
14    float Cvalue = 0;
15
16    // Thread row and column within Csub
17    int row = threadIdx.y;
18    int col = threadIdx.x;
19
20    // Loop over all the sub-matrices of A and B that are
21    // required to compute Csub
22    // Multiply each pair of sub-matrices together
23    // and accumulate the results
24    for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
25
26        // Get sub-matrix Asub of A
27        Matrix Asub = GetSubMatrix(A, blockRow, m);
28
29        // Get sub-matrix Bsub of B
30        Matrix Bsub = GetSubMatrix(B, m, blockCol);
31
32        // Shared memory used to store Asub and Bsub respectively
33        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
34        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
35
36        // Load Asub and Bsub from device memory to shared memory
37        // Each thread loads one element of each sub-matrix
38        As[row][col] = GetElement(Asub, row, col);
39        Bs[row][col] = GetElement(Bsub, row, col);
40
41        // Synchronize to make sure the sub-matrices are loaded
42        // before starting the computation
43        __syncthreads();
44
45        // Multiply Asub and Bsub together
46        for (int e = 0; e < BLOCK_SIZE; ++e)
47            Cvalue += As[row][e] * Bs[e][col];
48
49        // Synchronize to make sure that the preceding
50        // computation is done before loading two new
51        // sub-matrices of A and B in the next iteration
52        __syncthreads();
53    }
54
55    // Write Csub to device memory
56    // Each thread writes one element
57    SetElement(Csub, row, col, Cvalue);
```

```
58 }
```

Sólo se ha puesto la parte que muestra la memoria compartida, existen más funciones para la ejecución de este código como por ejemplo, *GetSubMatrix*, *SetElement*, *GetElement*, etc, pero no se ha considerado ponerlas ya que son funciones que simplemente hacen los cálculos menores para que el kernel "principal" del programa funcione, aunque se disponen de ellos en la página oficial de *NVIDIA* ya que es un ejemplo que se usa de forma práctica [12]. En la Figura 2.9 vemos gráficamente lo explicado en esta subsección de forma gráfica

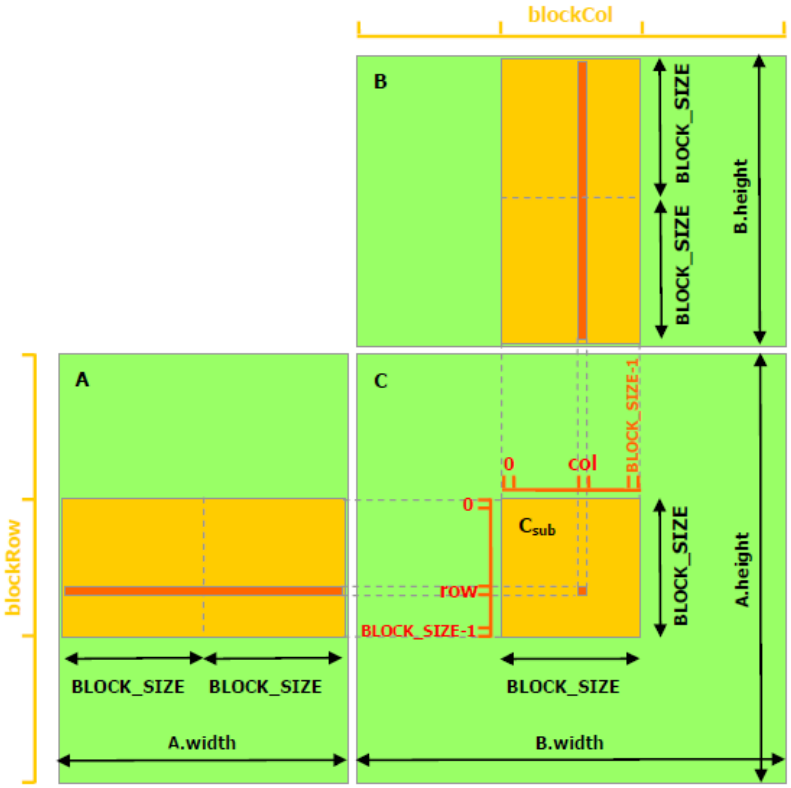

Figura 2.9: Multiplicación con memoria compartida

CAPÍTULO 3
Algoritmo EM

El algoritmo *EM* o de *Expectation–Maximization* es un método iterativo para encontrar el *maximun likelihood* estimado de parámetros en modelos estadísticos, donde el modelo depende de variables latentes no observadas. El procedimiento consiste en definir una esperanza o expectativa en particular y luego maximizarla, el procedimiento como ya se ha mencionado es iterativo, iniciándose en un cierto valor inicial de los parámetros y actualizando los valores en cada iteración,paso *E* y paso *M*, los parámetros actualizados en cada iteración son los valores que maximizan la expectativa en esa iteración particular. El procedimiento fue introducido por Dempster, Laird y Rubin en 1977 como un mecanismo para manejar información perdida o ausente, sin embargo es aplicable de forma mucho más general y se ha utilizado con éxito en muchos campos de la estadística [18].

3.1 Maximum-likelihood

Hay dos aplicaciones principales para el algoritmo EM, la primera ocurre cuando en los datos faltan valores por problemas o limitaciones con el proceso de observación de los datos. La segunda se da al intentar optimizar la función de *likelihood*.

En los modelos tradicionales, el cálculo del máximo de la función *log verosimilitud* o LL se vuelve más complejo a medida que intentamos obtener un modelo más realista y flexible, debido en gran medida al aumento de parámetros, que origina mucho más tiempo de cálculo, por ejemplo en el procedimiento de *Newton-Rhapson* o *NR* se utiliza el vector gradiente multiplicado por el negativo de la inversa del *hessiano* (una matriz cuadrada de *nxn*, que contiene las segundas derivadas parciales) esto debe ser calculado y luego invertido, este cálculo con un gran número de parámetros puede llegar a ser complejo. []

En la Figura 3.1 se aprecia como por el método mencionado, si la pendiente es positiva, se incrementa “Avanza” y si es negativa “Retrocede”, en la Figura 3.2 si la curvatura es grande significaría que la pendiente cambia rápidamente como en el primer dibujo probablemente estemos cerca del máximo así que daría un paso pequeño, por el contrario si la curvatura es pequeña significaría que la pendiente no está cambiando mucho, por lo que daríamos un paso más grande ya que el máximo probablemente esté más lejos, hay otros métodos como el *BHHH* , *BHHH-2* , *DFP y BGGS* entre otros pero no es el objetivo de este proyecto el analizarlos, se nombra el *NR* por su sencillez técnica para demostrar el *porqué* del uso del algoritmo *EM* habiendo otros algoritmos [19].

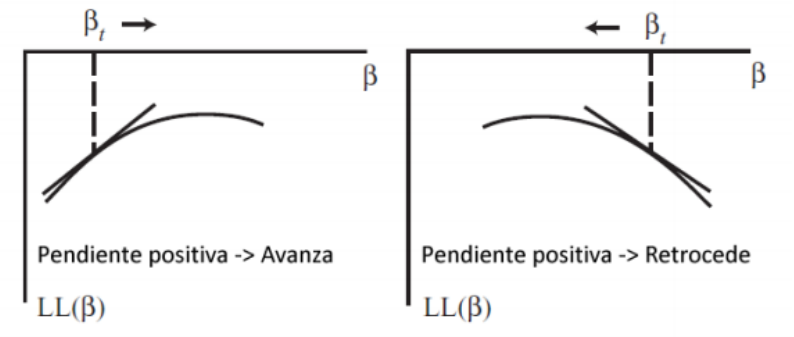

Figura 3.1: La dirección del paso según la pendiente

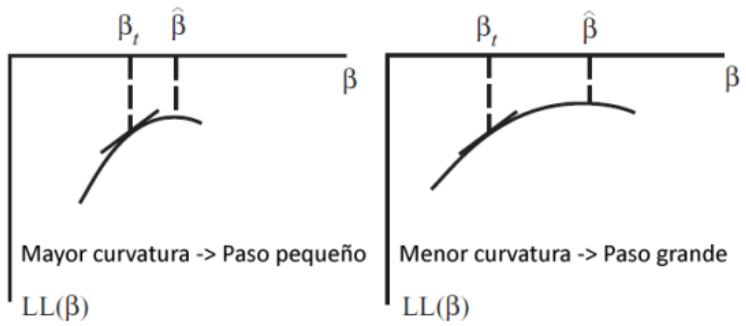

Figura 3.2: Paso grande o pequeño según curvatura

La función *LL* para los modelos mencionados en el párrafo anterior es a menudo aproximadamente cuadrática, de manera que estos métodos operan de manera efectiva, sin embargo, cuando el modelo se vuelve más complejo, la función *LL* generalmente se vuelve menos cuadrática, al menos en algunas regiones del espacio de parámetros. Este problema se manifiesta de dos formas, el procedimiento iterativo puede quedarse "enganchado" en áreas no cuadráticas de la función *LL*, dando pasos pequeños como vimos anteriormente que no representarían apenas mejora en la *LL*, o el procedimiento puede "pasar de largo" el máximo repetidas veces.

3.1.1. Maximum-likelihood formulación

Supongamos que disponemos de un *training set* $\mathcal{X} = \{x_1, x_2, \ldots, x_N\}$, donde cada x_i puede ser unidimensional o multidimensional.

Para simplificar la formulación consideraremos el caso unidimensional que corresponde a distribuciones normales univariadas, también asumiremos que todas las muestras han sido generadas aleatoriamente a partir de una distribución normal o de Gauss que desconocemos:

Media:

$$\mathbb{E}[x] = \int_{-\infty}^{\infty} \mathcal{N}(x \mid \mu, \sigma^2)\, x \, \mathrm{d}x = \mu$$

Momento de segundo orden:

$$\mathbb{E}[x^2] = \int_{-\infty}^{\infty} \mathcal{N}(x \mid \mu, \sigma^2)\, x^2 \, \mathrm{d}x = \mu^2 + \sigma^2$$

Varianza:

$$\text{var}[x] = \mathbb{E}[x^2] - \mathbb{E}[x]^2 = \sigma^2$$

Normal multivariada

$$\mathcal{N}(\mathbf{x} \mid \mu, \mathbf{\Sigma}) = \frac{1}{(2\pi)^{d/2}\ |\mathbf{\Sigma}|^{1/2}}\ e^{-\frac{1}{2}(\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu)}$$

$\mu \in \mathbb{R}^d$ es el vector media y $\mathbf{\Sigma} \in \mathbb{R}^{d\times d}$ es la matriz de varianzas-coavarianzas.

Las muestras que han sido generadas de manera independiente a partir de una misma distribución diremos que son *independently and identically distributed* (i.i.d.). El objetivo es estimar los valores de μ y σ (*mean* y *variance* respectivamente) a partir de los datos disponibles

Si la probabilidad conjunta de dos eventos independientes es el producto de sus probabilidades marginales, entonces, si $\mathcal{X}$ es i.i.d. la verosimilitud del *training set* es:

$$p(\mathcal{X} \mid \mu, \sigma^2) = \prod_{n=1}^{N} \mathcal{N}(x_n \mid \mu, \sigma^2) = \prod_{n=1}^{N} \frac{1}{(2\pi\sigma^2)^{(1/2)}}\ e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$$

Concretando, el objetivo es estimar los valores de μ y σ que maximizan la verosimilitud. Si tomamos el logaritmo

$$\ln p(\mathcal{X} \mid \mu, \sigma^2) = -\frac{1}{2\sigma^2}\sum_{n=1}^{N}(x_n - \mu)^2 - \frac{N}{2}\ln\sigma^2 - \frac{N}{2}\ln(2\pi)$$

Maximizando con respecto a μ tenemos

$$\frac{\partial \ln p(\mathcal{X} \mid \mu, \sigma^2)}{\partial\mu} = \frac{\partial \frac{-1}{2\sigma^2}\sum_{n=1}^{N}(x_n-\mu)^2 - \frac{N}{2}\ln\sigma^2 - \frac{N}{2}\ln(2\pi)}{\partial\mu} = -\frac{1}{\sigma^2}\sum_{n=1}^{N}(x_n - \mu)$$

Igualando a cero tenemos que

$$\frac{1}{\sigma^2}\sum_{n=1}^{N}(x_n-\mu) = 0 \quad \Rightarrow \quad \sum_{n=1}^{N} x_n = \sum_{n=1}^{N}\mu = N * \mu \quad \Rightarrow \quad \mu = \frac{1}{N}\sum_{n=1}^{N} x_n$$

Esto se conoce como la media muestral, que en algunos casos denotaremos como μ_{ML}, por *Maximum Likelihood*.

Maximizando con respecto a σ^2 tenemos

$$\frac{\partial \ln p(\mathcal{X} \mid \mu, \sigma^2)}{\partial\sigma^2} = \frac{\partial \frac{-1}{2\sigma^2}\sum_{n=1}^{N}(x_n-\mu)^2 - \frac{N}{2}\ln\sigma^2 - \frac{N}{2}\ln(2\pi)}{\partial\sigma^2} = \frac{1}{2\sigma^4}\sum_{n=1}^{N}(x_n-\mu)^2 - \frac{N}{2}\frac{1}{\sigma^2}$$

Igualando a cero tenemos $\frac{1}{2\sigma^4}\sum_{n=1}^{N}(x_n-\mu)^2 = \frac{N}{2}\frac{1}{\sigma^2} \quad \Rightarrow \quad \sum_{n=1}^{N}(x_n-\mu)^2 = N\cdot\sigma^2$

Que es la varianza muestral obtenida por *Maximum Likelihood* respecto de la media muestral y que expresaremos como sigue:

$$\sigma^2_{ML} = \frac{1}{N}\sum_{n=1}^{N}(x_n - \mu_{ML})^2$$

Por cuestiones teóricas esta aproximación estima a la baja la varianza, se corrige dividiendo por $N-1$ en lugar de por N. El cálculo de la media es correcto.

3.2 GMM - EM

En esta sección se explicará como usar el algoritmo *EM* utilizando para ello *GMM* tal y como se ha hecho con el código desarrollado.

Las mixturas de distribuciones normales o de Gauss, o también conocidas como *Gaussian Mixture Models GMM*, son una de las mejores técnicas *clustering* para obtener una estimación de la densidad de un conjunto de muestras utilizadas para el aprendizaje y se aplican al *training set* sin etiquetar, es una técnica de aprendizaje no supervisado. Esta técnica se utiliza a menudo en las primeras fases del estudio para conocer el proceso sobre el que se va a trabajar, se pueden descubrir relaciones en los datos que puedan ayudar a descubrir patrones que se repiten más porque se concentran o presentan mayor densidad en ciertas regiones del espacio multidimensional.

Se proporcionan un número de puntos N en un espacio dimensional d, lo que queremos en este caso es encontrar un conjunto de K distribuciones Gaussianas que mejor represente la distribución de los puntos, K es fijada a priori pero las medias y las covarianzas de las distribuciones son desconocidas. En este ejercicio "sin superversar" no se nos proporciona cual de los N puntos de los datos pertenece a que K Gaussiana. Se desea también de igual modo estimar la probabilidad de que cada punto n venga de la distribución k, esta probabilidad es $P(k|n)$, esta matriz también se conoce como *reponsibilitiy matrix*..

- μ_k La media k, cada vector es de longuitud d
- σ_k La matriz de covarianza k, cada una de tamaño dxd
- $P(k \mid n)$ Las probabilidades k para cada n punto

De hecho $\mathcal{L}$ es la clave para todo el problema, es definida como proporcional a la probabilidad de un *dataset* por lo que encontraremos los mejores valores para los parámetros maximizando el *likelihood* $\mathcal{L}$,

Como los puntos de los datos se asumen independientes $\mathcal{L}$ es el producto de las probabilidades de encontrar un punto en cada posición x_n.

$$L = \prod_n p(x_n) \tag{3.1}$$

Podemos dividir $p(x_n)$ en sus contribuciones para cada una de las k Gaussianas:

$$p(x_n) = \sum_k N(x_n \mid \mu_k, \mathbf{\Sigma_k}) P(k) \tag{3.2}$$

$p(x)$ es a menudo llamada la mixtura de pesos de los puntos x_n, podemos dividir $p(x_n)$ en sus K contribuciones individuales, dándole sus probabilidades individuales.

$$p_{nk} = P(k \mid n) = \frac{N(x_n \mid \mu_k, \mathbf{\Sigma_k}) P(k)}{p(x_n)} \tag{3.3}$$

Las ecuaciones 3.1 y 3.3 son necesarias para calcular $\mathcal{L}$, el p_{nk} y los valores de μ_k y σ_k que es lo consideramos en el algoritmo *EM* como un paso E.

La siguiente cuestión reside en como encontramos los valores de μ_k, Σ, y $P(k)$, como solo conocemos la probabilidad de que un punto en particular de los datos este en cierta

Gaussiana solo deberíamos tener en cuenta la fracción de p_{nk} de cada punto, eso deriva en la siguiente estimación del *maximun likelihood*.

$$\hat{\mu}_k = \frac{\sum_n p_{nk} x_n}{\sum_n p_{nk}} \tag{3.4}$$

$$\hat{\Sigma}_k = \frac{\sum_n p_{nk}(x_n - \hat{\mu_k}) \otimes (x_n - \hat{\mu_k})}{\sum_n p_{nk}} \tag{3.5}$$

De manera similar también obtenemos $\hat{P}(k)$

$$\hat{P}(k) = \frac{1}{N} \sum_n p_{nk} \tag{3.6}$$

Los "sombreros" denotan estimaciones, las ecuaciones 3.4, 3.5 y 3.6 son el paso M en el algoritmo *EM*. El potencial del algoritmo *EM* reside en un teorema de mucho más calado matemático, el cual no es el objetivo de este proyecto explicar pero que se puede resumir de manera que: empezando con cualquier valor de los parámetros, una iteración del paso E seguida de un paso M incrementará el valor del *likelihood* $\mathcal{L}$ y en sucesivas iteraciones convergerán en un máximo.

- Las inicializaciones para los valores de μ_k, Σ_k y las fraciones $P(k)$
- Ejecutamos un paso E para conseguir nuevos valores de p_{nk} seguido de un paso M para calcular los nuevos valores de μ_k, Σ_k y las fraciones $P(k)$
- Paramos cuando el valor de $\mathcal{L}$ ya no cambia

Un punto importante sobre estos pasos descritos es que a menudo los valores de la función Gaussiana normalmente son muy pequeños que tienden a cero, por ello es importante trabajar con logaritmos para estas densidades.

$$\log N(x \mid \mu, \Sigma) = -\frac{1}{2}(x-\mu)\ \Sigma^{-1}\ (x-\mu) - d/2\ log(2\pi) - \frac{1}{2}\ log\ det(\Sigma) \tag{3.7}$$

El problema reside en la ecuación 3.1 cuando debemos de sumar todas las cantidades, estas podrían ser demasiado pequeñas para ser sumadas, pero si lo reconstruimos con su logaritmo el problema se resuelve, la solución a este problema es la conocida como formula *log-sum-exp*.

$$log(\sum_i exp(z_i)) = z_max + log((\sum_i exp(z_i - z_{max})) \tag{3.8}$$

3.2.1. Ejemplo didáctico de GMM-EM

En este ejemplo vamos a ver por encima un ejemplo práctico de como el algoritmo EM funciona usando GMMs, en este caso se ha utilizado un algoritmo k-means para inicializar las medias para evitar inicializarla al azar. Consideremos a un *GMM* con los siguientes parámetros:

$$\mu_1 = \begin{bmatrix} 0 \\ 4 \end{bmatrix}, \mu_2 = \begin{bmatrix} -2 \\ 0 \end{bmatrix}, \Sigma_1 = \begin{bmatrix} 3 & 0 \\ 0 & 1/2 \end{bmatrix}, \Sigma_2 = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix},$$

Y de pesos relativos

$$w_1 = 0{,}6 \; ; \; w_2 = 0{,}4$$

La Figura 3.3 nos muestra sus densidades, así como también 1000 muestras dibujadas, las muestras de la primera y segunda componentes están marcadas en rojo y azul respectivamente.

Ejecutamos el algoritmo *k-means* en las 1000 muestras y usamos los *centroides* de los dos *k-means* como las estimaciones iniciales de las medias:

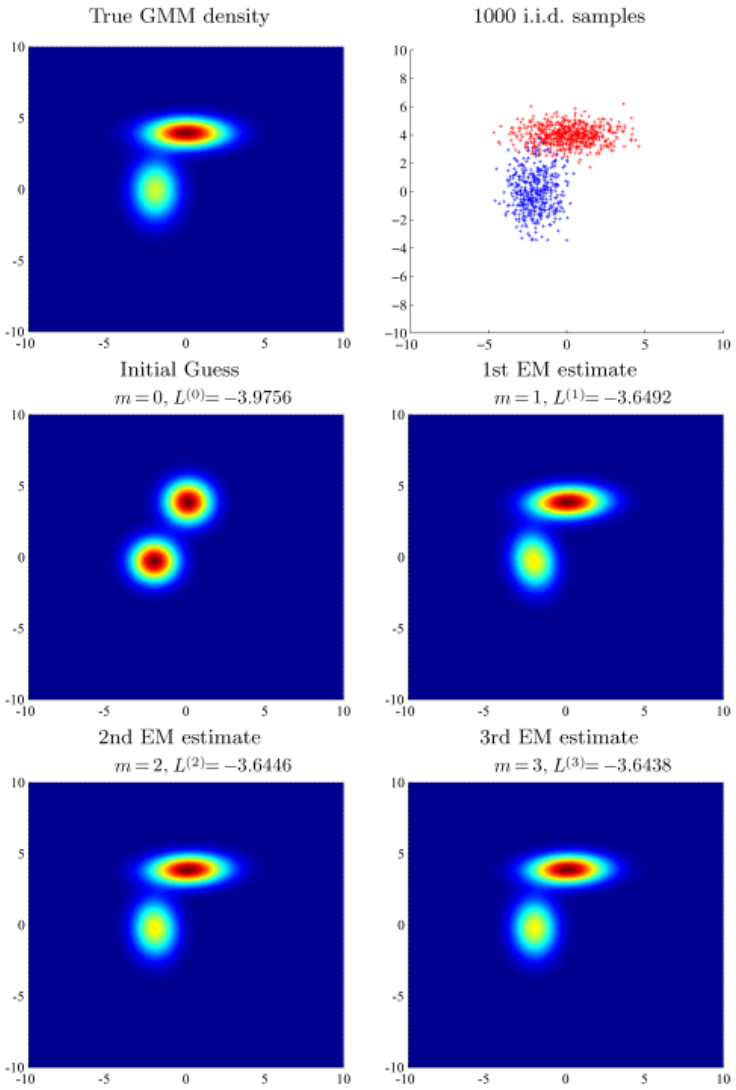

Figura 3.3: Ejecución algoritmo EM ejemplo visual

La densidad correspondiente a estas estimaciones iniciales se ve en la primera imagen de la Figura 3.3, en la segunda imagen de la Figura 3.3 vemos impresos los 1000 puntos indicados anteriormente de dos colores, cargamos la inicializaciones de las variables en la tercera imagen *Initial Guess*, en las tres sucesivas imágenes vemos como el algoritmo *EM* converge en solo tres iteraciones. No se exponen los datos numéricos ya que solo se pretende que sirva como un ejemplo gráfico de como funciona el algoritmo *EM*, pero están disponibles si se desean [20].

CAPÍTULO 4
Código desarrollado

Habiendo sentado las bases teóricas del proyecto, pasamos ahora a analizar el código desarrollado.

A continuación, veremos la implementación del algoritmo en *C++*. Esta primera versión se ejecuta completamente en *CPU* como un único hilo. Seguidamente describiremos las premisas con las que se abordó el proceso de paralelización y las decisiones de diseño que se tomaron. Por último, estudiaremos las diferencias entre el código final, implementado en *C++* y *CUDA*, y la versión original.

4.1 Decisiones de diseño para la versión *CPU*

Dado el objetivo del proyecto, el primer paso en su desarrollo fue la implementación de los algoritmos vistos anteriormente directamente en *C++*. No se hace uso de ningún tipo de paralelización ni se toma ventaja de los múltiples núcleos que se pueden encontrar en *CPUs* actuales. Para esta primera versión se tomó como base el código que puede encontrarse en [21].

En esta sección, analizaremos algunas de las decisiones de diseño que se tomaron a la hora de implementar los algoritmos y desarrollar el programa.

4.1.1. Indexación de *arrays*

Una de los tipos de datos básicos de los lenguajes de programación modernos es el *array*. Este tipo de datos describe una colección de elementos indexados por un número entero positivo.

En *C++*, un *array* de 5 elementos de tipo `double` se definiría de la siguiente manera:

```
1 double foo[5];
```

Este tipo de datos es perfecto para representar vectores o matrices en algoritmos que tratan directamente con estructuras matemáticas. Tal es el caso de nuestro programa. Un vector sería un *array* unidimensional, mientras que una matriz sería un *array* bidimensional.

En *C++*, podemos definir un *array* bidimensional de la siguiente forma:

```
1 double var[5][8];
```

Sin embargo, nos encontramos con un problema a la hora de tratar con *arrays* multidimensionales en *C++*. Solo la primera dimensión puede ser definida en tiempo de ejecución. Si se tienen más dimensiones, se deberá definirlas en tiempo de compilación. Dado que normalmente no conocemos el tamaño de los *arrays* hasta el momento de la ejecución, esto significa que no podremos usar *arrays* multidimensionales.

Existen diversas soluciones a este problema, siendo la más versátil el uso de *arrays* unidimensionales en los que guardamos una matriz aplanada. Esto es, guardar cada fila a continuación de la anterior, como se ve en el siguiente ejemplo:

```
1 // Matriz de 3 filas por 5 columnas
2 double var[15];
3 var[0 * 5 + 4] = 3.2;
4 var[2 * 5 + 1] = 7.6;
```

Como se puede observar, si tenemos una matriz 3x5, definimos un *array* de tamaño $3 \times 5 = 15$. Para acceder al elemento `[0][4]`, usamos el índice `[0 * 5 + 4]`, mientras que el elemento `[2][1]`, se indexa como `[2 * 5 + 1]`. Esto tiene una ventaja importante, que es el hecho de que realmente en la memoria los datos siempre se guardan de esta forma, lo que más adelante nos permitirá realizar optimizaciones a la hora de acceder a memoria.

4.1.2. Datos de entrada y salida

En el proceso de diseño de todo programa se debe decidir qué datos se requieren como entrada y qué será lo que se devuelva al usuario como salida.

Datos de entrada

Nuestra aplicación se ejecutará como un programa de texto en una terminal. Requiere dos parámetros de entrada, como se puede ver a continuación.

```
1 $ ./cpu datos/test3.data resultados.txt
```

El primer parámetro es el fichero de datos con las muestras que queremos clasificar. Debe contener una cabecera indicando la cantidad de muestras, así como el número de dimensiones en las que están representadas. A continuación, le seguirán las muestras; una por línea, y sus elementos separados por espacios. Veamos un ejemplo de un fichero de datos:

```
1 500 2 0
2 2.0756 5.71966
3 30.4059 25.7652
4 7.76758 0.647905
5 23.1863 28.0345
6 5.0597 1.50914
7 ...
```

El segundo parámetro requerido por el programa es el fichero de salida, en el que escribiremos los resultados obtenidos.

Datos de salida

La aplicación provee datos al usuario de dos formas distintas: imprimiendo información en la consola a medida que se ejecuta el algoritmo, y escribiendo los resultados finales en el fichero de salida.

A continuación se muestra un ejemplo del estado de la consola mientras se ejecuta el programa:

```
...

**************************************
Ejecutando algoritmo usando 2 gaussianas..

Converge en 11 iteraciones. Verosimilitud: -2.42592e+03
Tiempo de ejecucion: 1 ms.

**************************************
Ejecutando algoritmo usando 5 gaussianas.............

Converge en 134 iteraciones. Verosimilitud: -1.85602e+03
Tiempo de ejecucion: 26 ms.

**************************************
Ejecutando algoritmo usando 10 gaussianas.......

Converge en 62 iteraciones. Verosimilitud: -1.80013e+03
Tiempo de ejecucion: 25 ms.

...
```

Por último, tenemos el fichero de salida mencionado anteriormente, en el que cada línea muestra el resultado de una ejecución completa del algoritmo *EM*, como se puede ver a continuación:

```
1,0,3,-2431.23
2,1,11,-2425.92
5,26,134,-1856.02
10,25,62,-1800.13
15,71,136,-1737.74
20,137,191,-1710.05
...
```

El primer número indica la cantidad de gaussianas utilizadas, seguido por el tiempo de ejecución en ms, el número de iteraciones realizadas y el valor de la verosimilitud en la última iteración.

4.1.3. Estructura del código

Uno de los principales objetivos que se buscaron a la hora de desarrollar el proyecto fue claridad del código. Esto se traduce, entre otras cosas, en elegir cuidadosamente los nombres de ficheros, funciones y variables. Se consideró también importante ceñirse a las buenas prácticas habituales del lenguaje.

Estos puntos proveen el enfoque general por el cual se llegó al diseño de la estructura del código que vemos a continuación.

- *main.cpp*
  - `main(int, char*)`
  - `numeroAleatorio(unsigned int)`
- *gmm.h*
- *gmm.cpp*
  - `resultado_type estimar(double*, size_t, size_t)`
  - `void calcular()`
  - `void cholesky(size_t)`
  - `void resolver(double*)`
  - `double logaritmoDeterminante()`
  - `void imprimir(bool, unsigned int)`

4.2 Versión *CPU*

Teniendo en cuenta las decisiones de diseño vistas en la sección anterior, pasamos ahora a realizar una descripción exhaustiva del código desarrollado para la versión *CPU*.

4.2.1. *main.cpp*

Toda aplicación escrita en *C++* debe contener necesariamente una función global `main(int, char*)`. Dado que es el punto de ingreso del programa y por lo tanto una parte fundamental del mismo, es habitual llamarle al fichero que la contiene *main.cpp*.

Inicialización del programa

La función `main(int, char*)` empieza comprobando los parámetros con los que ha sido llamado el programa son válidos. Una vez realizado esto, se procede a leer el fichero de datos almacenando las muestras en un *array* llamado `double *datos`. A continuación se definen varios parámetros necesarios para la ejecución del programa, como la precisión de parada o el número máximo de iteraciones del algoritmo *EM*, así como el acceso en modo escritura al fichero de salida que se utilizará para los resultados.

Una de las tareas más importantes que se llevan a cabo en `main(int, char*)` es el cálculo del vector media de todas las muestras y la matriz de varianzas-covarianzas.

```
1 for (size_t j = 0; j < numDimensiones; j++) {
2    suma = 0.0;
3
4    for (size_t i = 0; i < numMuestras; i++) {
5       suma += datos[i * numDimensiones + j];
6    }
7
8    media_muestras[j] = suma / numMuestras;
9 }
```

```
10
11 for (size_t j = 0; j < numDimensiones; j++) {
12    for (size_t h = j; h < numDimensiones; h++) {
13       suma = 0.0;
14
15       for (size_t i = 0; i < numMuestras; i++) {
16          suma +=
17             (datos[i * numDimensiones + j] - media_muestras[j])
18             * (datos[i * numDimensiones + h] - media_muestras[h]);
19       }
20
21       matriz_covarianzas[j * numDimensiones + h] =
22          suma / (numMuestras - 1);
23
24       matriz_covarianzas[h * numDimensiones + j] =
25          matriz_covarianza[j * numDimensiones + h];
26    }
27 }
```

El primer bloque `for` se encarga de iterar a través de las muestras calculando el vector media, al que almacenaremos en `double *media_muestras`. El siguiente bloque `for` genera la matriz de varianzas-covarianzas del conjunto de datos de entrada. Una de las características de estas matrices es que es simétrica respecto a su diagonal principal, por lo que únicamente hará falta calcular una de las mitades. Como se puede observar en la línea 12, el segundo `for` empieza su recorrido en la diagonal. Finalmente, en la línea 24 se copia el valor calculado a la posición opuesta de la matriz.

Bucle principal

Llegado este punto, todo está preparado para comenzar la estimación de las mixturas. Como se explicó anteriormente, una de las decisiones de diseño fue que el programa debía crear múltiples mixturas a fin de encontrar la mejor aproximación a los datos muestrales.

```
1 size_t numGaussianasLista[] = { 1, 2, 5, 10, 15, 20 };
2
3 for (size_t z = 0;
4     z < sizeof(numGaussianasLista) / sizeof(*numGaussianasLista);
5     z++) {
6     ...
```

Como se puede observar, el bucle `for` principal se encarga de iterar a través del *array* `size_t numGaussianasLista[]`, en el que se ha definido una lista de tamaños para las distintas mixturas que deseamos comprobar.

A continuación, se procede a la inicialización de los parámetros de cada una de las gaussianas.

```
1 for (size_t k = 0; k < numGaussianas; k++) {
2    pesos[k] = 1.0 / numGaussianas;
3
4    size_t muestra =
5        (size_t) random_int (numMuestras);
```

```
6
7     for (size_t i = 0; i < numDimensiones; i++) {
8        medias[k * numDimensiones + i] =
9           datos[muestra * numDimensiones + i];
10
11       for (size_t j = 0; j < numDimensiones; j++) {
12          covarianzas[k * numDimensiones * numDimensiones
13          + i * numDimensiones + j] =
14             matriz_covarianzas[i * numDimensiones + j];
15       }
16    }
17 }
```

Siguiendo la teoría explicada previamente, se asigna a cada gaussiana un peso sobre la mixtura inicialmente idéntico. Seguidamente se les asigna a cada una un vector media distinto, elegidos aleatoriamente a partir de las muestras mediante la función `unsigned int random_int( unsigned int rango )`. El proceso de inicialización de los parámetros finaliza creando una copia para cada gaussiana de la matriz de varianzas-covarianzas calculada anteriormente.

```
1  unsigned int numeroAleatorio( unsigned int rango )
2  {
3      random_device  rd;
4      mt19937 generator( rd() );
5      uniform_int_distribution<unsigned int> ud( 0, rango-1 );
6
7      return ud(generator);
8  }
9
10
11 static random_device  _rd_;
12 static mt19937 _generator_( _rd_() );
13 static uniform_real_distribution<double> _ud_;
14
15 unsigned int random_int( unsigned int rango )
16 {
17     return (unsigned int)( rango * _ud_(_generator_) );
18 }
```

Esta función hace uso de un generador pseudoaleatorio Mersenne Twister, el cual es capaz de producir números de 32 bits con un período de $2^{19937} - 1$ [22]. Su valor de retorno es un número entero sin signo elegido a partir de una distribución uniforme de los números naturales hasta `rango - 1`.

Una vez inicializadas las gaussianas, solo nos queda comenzar la ejecución del algoritmo *EM* con los parámetros que se han ido definiendo.

```
1  high_resolution_clock::time_point tInicio =
2     high_resolution_clock::now();
3
4  GMM gmm(numGaussianas, pesos, medias, covarianzas,
5     maxIteraciones, precision);
6
7  auto resultados = gmm.estimar(datos, numMuestras, numDimensiones);
8
```

```
9  high_resolution_clock::time_point tFin =
10    high_resolution_clock::now();
```

Se hace uso de un reloj de alta resolución en las líneas 1 y 8. Ésto nos permitirá llevar a cabo el estudio de los resultados obtenidos de forma precisa. Además, se puede ver en la línea 6 que el método `resultado_type estimar(double*, size_t, size_t)` devuelve una serie de resultados que se imprimiran en el fichero de salida, junto con el número de gaussianas de la mixtura y el tiempo de ejecución del algoritmo *EM*.

Con esto finalizamos la aproximación de una mixtura. Como se indicó anteriormente, este proceso se ejecutará iterativamente para cada una de las mixturas de distintos tamaños que se deben comprobar.

Por último, llegado el fin del programa se procede a realizar las tareas de limpieza habituales, cerrando el fichero de salida y desbloqueando la memoria utilizada por la aplicación.

4.2.2. *gmm.h*

Como es habitual en aplicaciones desarrolladas en *C++*, el programa hace uso de un fichero de cabecera. En este fichero se declaran clases, así como las firmas de sus métodos u otras funciones auxiliares. También se pueden definir variables y valores constantes de los que se hará uso en el programa.

Lo más destacable de este fichero es la definición de una constante que se utilizará en varios puntos del programa y la estructura de datos que usaremos como valor de retorno del algoritmo *EM*.

```
1  const double MENOS_INFINITO = -numeric_limits<double>::max();
2
3  struct resultado_type {
4    size_t iteracion;
5    double verosimilitud;
6  };
```

En la primera línea se hace uso de una técnica común en *C++* cuando se requieren límites superiores o inferiores en un rango. En este caso definimos `const double MENOS_INFINITO` como el mayor número representable con una variable `double` con signo negativo. A continuación, se puede ver la estructura `struct resultado_type` con miembros `size_t iteracion` y `double verosimilitud`. Estos son los datos que retornaremos como resultado de la ejecución del algoritmo.

Por último, tenemos la definición de la clase `GMM`. En ella, como indican las buenas prácticas, se definen sus propiedades y métodos, asignando a cada uno un modificador de acceso, ya sea público o privado, a fin de mantener un buen encapsulamiento de sus miembros.

4.2.3. *gmm.cpp*

Este es el fichero donde se implementa el algoritmo *EM* y, por tanto, la parte más interesante del código.

Inicialización de la clase

Lo primero que nos encontramos es el constructor de la clase `GMM`.

```
1  GMM::GMM(size_t numGaussianas, double* pesos,
2         double* medias, double* covarianzas,
3         unsigned int maxIteraciones = 250,
4         double precision = 1e-5)
5  {
6     ...
7
8     this->numGaussianas = numGaussianas;
9
10    this->pesos = pesos;
11    this->medias = medias;
12    this->covarianzas = covarianzas;
13
14    this->datos = NULL;
15    this->maxIteraciones = maxIteraciones;
16    this->precision = precision;
17
18    this->verosimilitud = MENOS_INFINITO;
19 }
```

Se han omitido las validaciones que se realizan sobre los parámetros de entrada. El trabajo de este método es la creación de una instancia de la clase. Concretamente, inicializa los *arrays* de pesos, medias y covarianzas, así como la precisión de parada y el número máximo de iteraciones del algoritmo *EM*. Obsérvese, las inicializaciones `double verosimilitud` , `MENOS_INFINITO`, como se ha visto en el capítulo de teoría.

En contraste con el constructor, nos encontramos al destructor de la clase `GMM`. Su trabajo es liberar correctamente la memoria utilizada por las instancias de la clase.

Método principal

Llegado este punto, la instancia ha sido inicializada, las muestras están en memoria, las gaussianas tienen sus parámetros establecidos. Solo resta comenzar el proceso de iteración entre los pasos *E* y *M* del algoritmo. Para ello tenemos el método `resultado_type estimar(double*, size_t, size_t)`.

```
1  resultado_type GMM::estimar(double *datos,
2                       size_t numMuestras,
3                       size_t numDimensiones)
4  {
5     ...
6
7     double ultimaVerosimilitud = verosimilitud;
8
9     this->resp = new double[numMuestras * numGaussianas];
10    this->L = new double[numDimensiones * numDimensiones];
11    this->v = new double[numDimensiones];
```

Se han vuelto a omitir las validaciones realizadas sobre los parámetros de entrada del método. En la línea 7 se observa la definición de la variable `double ultimaVerosimilitud`

que utilizaremos para determinar el momento en el que el algoritmo converge. Seguidamente, asignamos la memoria necesaria para las matrices `resp` y `L`, así como el vector `v`.

```
1  for (i = 1; i <= maxIteraciones; i++) {
2     ...
3
4     calcular();
5
6     if (abs((verosimilitud - ultimaVerosimilitud) / verosimilitud)
7        <= precision) {
8
9        finalizado = true;
10
11       break;
12    }
13
14    ultimaVerosimilitud = verosimilitud;
15 }
```

Aquí vemos el bucle `for` en el que se basa el algoritmo *EM*. Al comienzo del bloque se imprime información sobre la iteración a uno de los *streams* estándar, lo que se ha omitido. Le sigue una llamada a `void calcular()`, método que veremos más adelante, y llegamos a la condición de parada en la línea 6.

La condición de parada comprueba la diferencia entre las verosimilitudes calculadas en la iteración actual y la anterior para dividirlo nuevamente por la verosimilitud. Si dicho cálculo se encuentra dentro del rango de precisión deseado, finalizamos el proceso iterativo. En caso contrario, se actualiza la variable `double ultimaVerosimilitud` en preparación para la siguiente iteración.

```
1     if (!finalizado) i--;
2
3     imprimir(finalizado, i);
4
5     resultado_type resultados;
6
7     resultados.iteracion = i;
8     resultados.verosimilitud = verosimilitud;
9
10    return resultados;
11 }
```

Cuando se sale del bucle `for`, ya sea porque se cumple la condición de parada o se ha llegado al límite de iteraciones del algoritmo, imprimimos el resultado de la última iteración ejecutada en el *stream* estándar. Una pequeña consideración a tener en cuenta es que si el bucle llegó al límite de iteraciones (`!finalizado`), tendremos que decrementar la variable contador `i` en una unidad para poder utilizarla como información para el usuario.

Solo resta crear el objeto `resultados`, asignar los datos obtenidos y devolverlo como valor de retorno del método.

Paso E

En el apartado anterior se vio el bucle principal del algoritmo *EM*. En él, cada iteración realiza una llamada al método `void calcular()`, cuyo código analizaremos en estos dos últimos apartados de la sección.

```
1  void GMM::calcular()
2  {
3      double *logDets = new double[numGaussianas];
4      double *u = new double[numDimensiones];
5      double suma, verosimilitudParcial,
6         maxContribucion, sumaProbabilidades;
```

El método comienza definiendo las variables necesarias para realizar los cálculos que se vieron en la sección de teoría, las cuales se irán explicando a medida que se utilicen. A continuación, tenemos el primero de los dos bucles `for` que componen el paso *E* del algoritmo.

```
1  for (size_t k = 0; k < numGaussianas; k++) {
2     cholesky(k);
3     logDets[k] = logaritmoDeterminante();
4
5     for (size_t i = 0; i < numMuestras; i++) {
6        for (size_t j = 0; j < numDimensiones; j++) {
7           u[j] = datos[i * numDimensiones + j]
8              - medias[k * numDimensiones + j];
9        }
10
11       resolver(u);
12
13       suma = 0.0;
14
15       for (size_t j = 0; j < numDimensiones; j++) {
16          suma += v[j] * v[j];
17       }
18
19       resp[i * numGaussianas + k] = -0.5
20          * (suma + logDets[k]) + gsl_sf_log(pesos[k]);
21    }
22 }
```

Este primer bucle `for` se encarga de calcular los datos necesarios para determinar la verosimilitud de la mixtura, dados los parámetros que se tienen en esta iteración del algoritmo *EM*. Se calculan todos los datos para cada una de las gaussianas antes de continuar.

El primer paso es calcular la factorización de Cholesky de la matriz de varianzas-covarianzas. El resultado de esta operación es la matriz triangular inferior L, que se guardará en la variable de clase `double *L`, y que usaremos en varios puntos del paso *E*. A continuación, se calcula el logaritmo del determinante de esta matriz L.

El objetivo del bucle `for` de la línea 5 es el cálculo de la ecuación 3.7. Para ello, se necesita obtener el resultado de $(x - \mu)$, que se guardará en el *array* `double *u`. Una vez hecho esto, llamamos al método `void resolver(double*)` que calcula el vector v, del cual necesitamos el cuadrado. Finalmente, en la línea 19 somos capaces de seguir la ecuación 3.7.

```
1  verosimilitud = 0.0;
2
3  for (size_t i = 0; i < numMuestras; i++) {
4     maxContribucion = MENOS_INFINITO;
5
6     for (size_t k = 0; k < numGaussianas; k++) {
7        if (resp[i * numGaussianas + k] > maxContribucion) {
8           maxContribucion = resp[i * numGaussianas + k];
9        }
10    }
11
12    suma = 0.0;
13
14    for (size_t k = 0; k < numGaussianas; k++) {
15       suma += exp(resp[i * numGaussianas + k] - maxContribucion);
16    }
17
18    verosimilitudParcial = maxContribucion + gsl_sf_log(suma);
19
20    for (size_t k = 0; k < numGaussianas; k++) {
21       resp[i * numGaussianas + k] =
22          exp(resp[i * numGaussianas + k] - verosimilitudParcial);
23    }
24
25    verosimilitud += verosimilitudParcial;
26 }
```

Habiendo calculado la ecuación 3.7 para cada gaussiana, solo nos queda estimar la verosimilitud para la iteración actual del algoritmo *EM* (ecuación 3.1).

A tal fin, comenzamos iterando a través de las muestras en el bucle `for` de la línea 3. Para empezar, debemos determinar a qué gaussiana pertenece la muestra y cuál es el grado de contribución, que guardaremos en la variable `double maxContribucion`. Este proceso se lleva a cabo en el bucle `for` de la línea 6.

En la línea 14, nos encontramos otro bucle `for`, encargado de calcular $(z_i - z_{max})$. Utilizaremos este resultado para obtener el valor de la ecuación 3.8 en la línea 18.

Finalmente, en el bucle `for` de la línea 20 calculamos las probabilidades p_{nk} que utilizaremos en el paso *M*, y acabamos incrementando la variable `double verosimilitud` con el valor de la verosimilitud parcial estimada.

Paso M

Una vez realizado el paso *E*, solo resta volver a calcular los parámetros de la mixtura en el paso *M*.

```
1  for (size_t k = 0; k < numGaussianas; k++) {
2     sumaProbabilidades = 0.0;
3
4     for (size_t i = 0; i < numMuestras; i++) {
5        sumaProbabilidades += resp[i * numGaussianas + k];
6     }
7
8     pesos[k] = sumaProbabilidades / numMuestras;
9
```

```
10      for (size_t j = 0; j < numDimensiones; j++) {
11          suma = 0.0;
12
13          for (size_t i = 0; i < numMuestras; i++) {
14              suma += resp[i * numGaussianas + k]
15                  * datos[i * numDimensiones + j];
16          }
17
18          medias[k * numDimensiones + j] = suma / sumaProbabilidades;
19
20          for (size_t h = 0; h < numDimensiones; h++) {
21              suma = 0.0;
22
23              for (size_t i = 0; i < numMuestras; i++) {
24                  suma += resp[i * numGaussianas + k]
25                      * (datos[i * numDimensiones + j]
26                          - medias[k * numDimensiones + j])
27                      * (datos[i * numDimensiones + h]
28                          - medias[k * numDimensiones + h]);
29              }
30
31              covarianzas[k * numDimensiones * numDimensiones
32                  + j * numDimensiones + h] = suma / sumaProbabilidades;
33          }
34      }
35  }
```

De la misma forma que en el paso E, comenzamos el proceso con un bucle `for` que iterará a través de la lista de gaussianas, calculando todos sus parámetros.

El bucle `for` de la línea 4 ejecuta la operación $\sum_n p_{nk}$, guardando el resultado en la variable `double sumaProbabilidades` que se utilizará para el cálculo del resto de parámetros. A continuación, en la línea 8 se puede ver la ecuación 3.6, que actualiza los pesos de la mixtura. Nótese el uso del vector `double *resp` calculado en el paso E.

Para el cálculo de las medias se necesita realizar un sumatorio para todas las muestras, $\sum_n p_{nk} x_n$. Esta operación se lleva a cabo en el bucle `for` de la línea 13. Teniendo este resultado, solo queda dividirlo entre `sumaProbabilidades` (línea 18), completando la ecuación 3.4.

Por último, nos falta actualizar la matriz de varianzas-covarianzas. Similarmente al cálculo de las medias, esto requiere un sumatorio para todas las muestras. En este caso, $\sum_n p_{nk}(x_n - \hat{\mu_k}) \otimes (x_n - \hat{\mu_k})$, que se ejecuta en el bucle `for` de la línea 23. Una vez hecho esto, finalizamos el proceso en la línea 31 siguiendo la ecuación 3.5.

```
1  delete [] logDets;
2  delete [] u;
```

Llegado este punto, solo falta liberar la memoria asignada a las variables utilizadas en el método. Queda, así, finalizada la implementación del algoritmo *EM*.

4.3 Decisiones de diseño para la versión GPU

4.3.1. Expresiones lambda y funciones *reduce*

C++11 introdujo al lenguaje la capacidad de crear expresiones lambda. Esto significa que ahora se pueden crear métodos que aceptan funciones como parámetros, lo que permite un grado de expresividad que anteriormente no existía. A continuación, se puede ver un ejemplo del uso de una función lambda.

```
1 void func(std::vector<int>& v) {
2   std::for_each(v.begin(), v.end(), [](int) { /* código */ });
3 }
```

Obsérvese que el tercer parámetro del método `for_each` es una función, donde los corchetes definen las variables del alcance de `func` que se desean inyectar, entre paréntesis se encuentran sus parámetros de entrada y entre llaves el código.

Nosotros haremos uso de este tipo de funciones para escribir de forma genérica los métodos *reduce* que necesitaremos.

Función *reducir*

Empezaremos analizando la función `void reducir(...)`, la cual usaremos de forma extensiva en el programa. Es la parte más complicada del código, ya que se mezclan varios conceptos, además de implementar la lógica vista en el capítulo de teoría.

```
1  __device__ void reducir(Predicate valor, Predicate2 direccionResultado,
2      Predicate3 reduccionFinal, const size_t n,
3      volatile double *sharedData, const size_t numBloques)
4  {
5     __shared__ bool esUltimoBloque;
6
7     const size_t tid = threadIdx.x;
8     const size_t gridSize = (blockSize * 2) * gridDim.x;
9
10    size_t i = blockIdx.x * (blockSize * 2) + threadIdx.x;
```

`Predicate`, `Predicate2` y `Predicate3` indican que se esperan funciones lambda como parámetros de entrada. El modificador `volatile` indica al compilador que no intente optimizar el acceso a esa variable guardando elementos en registros, lo cual es importante ya que necesitamos que todos los hilos ejecutando este código accedan siempre a la información actualizada.

Empezamos definiendo la variable `bool esUltimoBloque`, con modificador `__shared__` que indica que todos los hilos de un bloque comparten su valor. Utilizaremos esta variable para asegurar que todos los hilos del último bloque ejecutan un bloque de código que veremos más adelante.

A continuación, definimos los índices `tid`, `gridSize` e `i` que se utilizarán para acceder a las variables en memoria, tal como se vio en el capítulo de teoría.

```
1     double suma = 0.0;
2
```

```
3     while (i < n) {
4        suma += valor(i);
5
6        if (i + blockSize < n) {
7           suma += valor(i+blockSize);
8        }
9
10       i += gridSize;
11    }
12
13    reducirBloque<blockSize>(sharedData, suma, tid);
```

El bucle `while` es una modificación de los algoritmos *reduce* vistos en teoría. El bucle se ejecutará mientras el identificador del hilo actual sea menor que el límite del *array* que estamos reduciendo.

Es aquí donde nos encontramos el primer uso de una expresión lambda. El valor que añadiremos a la variable `double suma` está determinado por la expresión lambda `double valor(size_t)`, la cual aceptamos como parámetro. Esto quiere decir que cada vez que utilicemos la función `void reducir(...)`, podremos hacer que reduzca sumas de expresiones radicalmente distintas. En la próxima sección veremos ejemplos de llamadas a esta función y quedará clara la gran utilidad de las expresiones lambda.

El condicional `if` de la línea 6 se asegura de que no nos pasemos al hacer la segunda suma. El hecho de hacer dos sumas es un tipo de *unroll* básico que nos permite distribuir mejor los elementos entre los bloques disponibles. Se hará referencia a esto más adelante cuando veamos las llamadas a los *kernels*.

Finalmente, llamamos a la función `void reducirBloque(volatile double*, double, const size_t)` para que sume los subtotales de cada hilo del bloque, dejando el resultado en la memoria compartida. Este método es idéntico al `reduce6` visto en teoría.

```
1     if (tid == 0) {
2        *(direccionResultado()) = sharedData[0];
3
4        __threadfence();
5
6        unsigned int ticket = atomicInc(&contadorBloques, numBloques);
7        esUltimoBloque = (ticket == numBloques - 1);
8     }
9
10    __syncthreads();
11
12    if (esUltimoBloque) {
13       reduccionFinal();
14
15       if (tid == 0) {
16          contadorBloques = 0;
17       }
18    }
19 }
```

Llegado este punto, tenemos un número de subtotales igual al de bloques del kernel. Dado que lo que nos interesa es hacer una suma completa y acabar con un solo resultado, tendremos que hacer una última reducción de estos subtotales.

Por la naturaleza de *CUDA*, es imposible sincronizar el estado de los hilos de distintos bloques, por lo que no podremos esperar a que todos acaben las sumas parciales para luego distribuir el trabajo de hacer la última reducción entre todos los bloques. Por tanto, la única forma de realizar esta última reducción es limitar el trabajo a un único bloque.

Para determinar cuál es el último bloque en ejecutar el código, usaremos el concepto de tickets. Para empezar, hacemos que el primer hilo de cada bloque guarde el subtotal del bloque en la memoria global. Obsérvese el uso de otra expresión lambda para determinar la dirección de memoria. A continuación, utilizando el método `__threadfence()`, aseguramos que todas las escrituras en memoria se han realizado antes de continuar.

Finalmente, incrementamos en uno el contador de tickets, utilizando una suma atómica para evitar condiciones de carrera. Si después de realizar el incremento, la variable `ticket == numBloques - 1`, eso significa que somos el último bloque en acabar. Solo resta ejecutar la función lambda `void reduccionFinal()` y reiniciar el contador de tickets.

Dicha función indicará el código que se desea ejecutar una vez realizada la reducción inicial, típicamente llamar a la función `void reducirFinal(...)`, la cual es esencialmente idéntica al bucle `while` de la función `void reducir(...)`, salvo que accede a la memoria compartida en vez de acceder a la memoria global, y su forma de indexar depende solamente del identificador del hilo, ya que será ejecutada por un solo bloque.

4.3.2. Estructura del código

Siguiendo la mismas directrices de diseño que se utilizó para la versión *CPU*, llegamos a la estructura del código que vemos a continuación.

- *main.cpp*
  - `main(int, char*)`
  - `numeroAleatorio(unsigned int)`
- *gmm.h*
- *gmm.cu*
  - `resultado_type estimar(double*, size_t, size_t)`
  - `void calcular()`
  - `void inicializarGPU()`
  - `void copiarDesdeGPU()`
  - `void limpiarGPU()`
  - `void imprimir(bool, unsigned int)`
- *kernels.cu*
  - `void paso_e_cholesky(double*, double*, const size_t)`
  - `void paso_e(double*, double*, double*, double*, double*, double*, const size_t, size_t const)`
  - `void paso_e2(double*, double*, const size_t, const size_t)`
  - `void paso_e_verosimilitud(double*, double*, const size_t)`
  - `void paso_m(double*, double*, double*, const size_t)`
  - `void paso_m2(double*, double*, double*, double*, const size_t)`

  - `void paso_m_covarianzas(double*, double*, double*, double*, const size_t, const size_t)`
  - `void paso_m_covarianzas_final(double*, double*, const size_t)`

- *auxiliares.cu*
  - `double logaritmoDeterminante(double*, const size_t, const size_t)`
  - `void reducirBloque(volatile double*, double, const size_t)`
  - `void reducirFinal(Predicate, Predicate2, volatile double*, size_t)`
  - `void reducir(Predicate, Predicate2, Predicate3, const size_t, volatile double*, const size_t)`

4.4 Versión GPU

Una vez vista la versión *CPU* del proyecto, pasamos a la parte central del trabajo. Una versión del algoritmo paralelizada utilizando la plataforma de desarrollo CUDA, en combinación con el lenguaje C++. Dado que el proceso se partió del código de la versión *CPU*, en esta sección se analizarán las diferencias entre las dos versiones.

4.4.1. *main.cpp* y *gmm.h*

La gran ventaja del diseño modular que se tomó desde el principio es que permite cambiar la implementación de la clase `GMM` sin tener que modificar el resto de ficheros del programa. El código de estos dos archivos es idéntico al de la versión *CPU*, por lo que no será necesario ningún comentario adicional.

4.4.2. *gmm.cu*

Este es el fichero donde se implementa el algoritmo *EM*, equivalente a *gmm.cpp* de la versión *CPU*.

Inicialización de la clase

El constructor de la clase `GMM` es idéntico al de la versión *CPU*. En el destructor encontramos una pequeña diferencia; una llamada al método `void limpiarGPU()`.

```
1 void GMM::limpiarGPU()
2 {
3     cudaFree(g_resp);
4     cudaFree(g_datos);
5
6     ...
7 }
```

Como se vio en el apartado de teoría, `cudaError_t cudaFree(void*)` se utiliza para liberar la memoria asignada a una variable. Se han omitido otras varias llamadas al mismo método.

Método principal

El método `resultado_type estimar(double*, size_t, size_t)` se mantiene prácticamente intacto respecto a la versión *CPU*. Solo se añade el código que se ve a continuación.

```
...

numRespuestasGPU = numMuestras / (BLOCK_SIZE << 1);

if (numMuestras % (BLOCK_SIZE << 1)) {
   numRespuestasGPU++;
}

inicializarGPU();

for (i = 1; i <= maxIteraciones; i++) {
    ...
```

En estas nuevas líneas de código se llevan a cabo dos operaciones importantes.

Por un lado, definimos la variable `size_t numRespuestasGPU`, que utilizaremos para determinar el número de hilos necesarios para los kernels más adelante. Como se puede observar, será igual al número de muestras dividido el doble del tamaño de bloque. `const size_t BLOCK_SIZE` se define en el fichero *auxiliares.cu* y depende de la arquitectura en la que se ejecute el programa. Su uso se explicará con más detalle más adelante.

La segunda operación añadida es la inicialización de las variables en la memoria de la tarjeta gráfica. Una vez hecho esto, tenemos el bucle `for` que ya veíamos en la versión *CPU*.

A continuación, se puede ver el código del método `void inicializarGPU()`.

```
void GMM::inicializarGPU()
{
    cudaMalloc(&g_datos,
        numMuestras * numDimensiones * sizeof(double));

    cudaMemcpy(g_datos, datos,
        numMuestras * numDimensiones * sizeof(double),
        cudaMemcpyHostToDevice);


    ...
}
```

Este método consiste de múltiples llamadas a `cudaError_t cudaMalloc(void**, size_t)`, función que reserva memoria global de la tarjeta gráfica y la asigna a un puntero que acepta como parámetro. También se hace una llamada a `cudaError_t cudaMemcpy(void*, const void*, size_t, enum cudaMemcpyKind)`, a fin de copiar el contenido del *array* `double *datos` a la memoria global de la tarjeta gráfica. Se han omitido el resto de llamadas.

Paso E

En estos dos últimos apartados de la sección analizaremos el método `void calcular()`. Dado que el proceso de paralelización es similar para los distintos bucles del algoritmo, se omitirán algunos trozos de poco interés.

```
1 void GMM::calcular()
2 {
3     dim3 dimBlock(BLOCK_SIZE, 1, 1);
4     size_t arraySize = BLOCK_SIZE * sizeof(double);
```

Empezamos definiendo los tamaños por defecto de bloque y de memoria compartida que utilizarán los *kernels*. Como se comentó previamente, la constante `const size_t BLOCK_SIZE` se define en el fichero *auxiliares.cu* y depende de la arquitectura en la que se ejecute el programa. Esto significa que, por defecto, los bloques tendrán `BLOCK_SIZE` hilos y un *array* en memoria compartida de `BLOCK_SIZE` *doubles*.

```
1 {
2    dim3 dimGrid(1, 1, numGaussianas);
3    dim3 dimBlock(1, 1, 1);
4    paso_e_cholesky<<<dimGrid, dimBlock, arraySize>>>(g_covarianzas, g_L,
 ↪  numDimensiones);
5 }
```

Aquí vemos la primera llamada a un *kernel*. Igual que en la versión *CPU*, la primera operación del paso *E* es calcular la descomposición de Cholesky de la matriz de varianzas-covarianzas.

La diferencia es que en vez de estar dentro de un bucle `for` que itera para todas las gaussianas, ahora creamos un *grid* de bloques cuyo tamaño es igual al número de gaussianas. Nótese que sobrescribimos el tamaño de bloque, definiendo un solo hilo por bloque. Claramente no se está sacando provecho de la paralelización masiva que provee *CUDA*, pero esto es debido a que la factorización de Cholesky es difícil de paralelizar y es un proceso increíblemente barato con respecto al resto de cálculos del programa.

De la misma forma que en la versión *CPU*, ahora se calculan los logaritmos de los determinantes de la matriz triangular *L*, seguidos de las ecuaciones 3.7 y 3.8. Se omite el código de estos cálculos ya que sus modificaciones resultan relativamente simples.

```
1 {
2    dim3 dimGrid(numRespuestasGPU, 1, 1);
3    paso_e_verosimilitud<<<dimGrid, dimBlock,
 ↪  arraySize>>>(g_verosimilitudParcial, g_verosimilitud, numMuestras);
4 }
```

El último cálculo del paso *E* es la suma de las verosimilitudes parciales resultantes de la ejecución del *kernel* anterior a este. En el *kernel* `void paso_e_verosimilitud(...)` se utiliza por primera vez la función `void reducir(...)` vista anteriormente.

Para ello, se define un *grid* de `numRespuestasGPU` bloques. Como vimos anteriormente, `numRespuestasGPU = numMuestras / (BLOCK_SIZE << 1)`.

Esto es, si tenemos 10000 muestras y `BLOCK_SIZE` es 128, `numRespuestasGPU` será igual a 40. Por tanto, el *kernel* se ejecutaría en un *grid* de 40 bloques de 128 hilos cada uno. Nótese que $40 \times 128 = 5120$, ya que dividimos entre el doble de `BLOCK_SIZE`. Esto se debe a que, como se mencionó en la sección de expresiones lambda, la función `void reducir(...)` hace un pequeño *unroll* por el que se suman dos elementos directamente, permitiéndonos reducir el tamaño del *grid*.

A continuación, veremos el código del *kernel* `void paso_e_verosimilitud(...)`.

```
1  __global__ void paso_e_verosimilitud(double *g_verosimilitudParcial,
2     double *g_verosimilitud, const size_t n)
3  {
4     extern __shared__ double sharedData[];
5
6     reducir<BLOCK_SIZE>(
7        [&] (size_t i) -> double { return g_verosimilitudParcial[i]; },
8        [&] () -> double* { return &g_verosimilitud[blockIdx.x]; },
9        [&] () -> void { reducirFinal<BLOCK_SIZE>(
10          [&] (size_t tid) -> double* { return &g_verosimilitud[tid]; },
11          [&] () -> double* { return &g_verosimilitud[0]; },
12          sharedData, gridDim.x); },
13       n, sharedData, gridDim.x * gridDim.y * gridDim.z);
14 }
```

Empezamos definiendo el *array* `double sharedData[]`. El modificador `__shared__` indica que reside en la memoria compartida para todos los hilos de un bloque.

En la línea 6, tenemos nuestra primera llamada al método `void reducir(...)`. Como vimos anteriormente, los tres primeros parámetros han de ser expresiones lambda.

La primera, en la línea 7, indica la expresión que queremos sumar. En este caso, será simplemente el *array* de verosimilitudes parciales. Nótese que lo indexamos con una variable `size_t i` que será inyectada con el identificador del hilo al ejecutarse.

La segunda expresión lambda, en la línea 8, indica la dirección en memoria global donde queremos guardar el resultado parcial del bloque.

Finalmente, la tercera expresión, en la línea 9, indica lo que se debe hacer al finalizar la reducción. En este caso, queremos llamar a la función `void reducirFinal(...)` para obtener un único resultado final. Esta función espera, a su vez, otras dos expresiones lambda. La primera siendo el valor a sumar, que vemos que es la misma dirección donde habíamos guardado los resultados parciales, y la segunda es la dirección de memoria donde vamos a guardar el resultado final.

Esto es un ejemplo perfecto de la complejidad que supone el desarrollo de aplicaciones con *CUDA*, debido a la necesidad de optimizar a nivel de acceso a memoria.

Paso M

Dado que en el paso *M* todos los *kernels* simplemente hacen llamadas al método `void reducir(...)`, solo veremos uno de ellos como ejemplo, `void paso_m2(...)`. Este *kernel* se encarga de calcular el sumatorio $\sum_n p_{nk} x_n$.

```
1  {
2     dim3 dimGrid(numRespuestasGPU, numDimensiones, numGaussianas);
3     paso_m2<<<dimGrid, dimBlock, arraySize>>>(g_resp, g_datos,
   ↪  g_sumaProbabilidades, g_medias, numMuestras);
4  }
```

Similarmente al resto de *kernels*, empezamos definiendo un *grid* de bloques. En este caso, utilizamos un *grid* tridimensional, donde la primera dimensión es de tamaño `numRespuestasGPU`, la segunda `numDimensiones`, y la tercera `numGaussianas`.

Con esto conseguimos distribuir los tres bucles `for` anidados que teníamos en la versión *CPU* entre todos los núcleos de la tarjeta gráfica. A continuación, se muestra el código del *kernel*.

```
1  __global__ void paso_m2(double *g_resp, double *g_datos,
2     double *g_sumaProbabilidades, double *g_medias, const size_t n)
3  {
4     extern __shared__ double sharedData[];
5
6     const size_t j = blockIdx.y;
7     const size_t k = blockIdx.z;
8
9     const size_t numGaussianas = gridDim.z;
10    const size_t numDimensiones = gridDim.y;
11
12    reducir<BLOCK_SIZE>(
13       [&] (size_t i) -> double { return g_resp[k * n + i] * g_datos[j * n +
   ↪  i]; },
14       [&] () -> double* { return &g_medias[k * numDimensiones * gridDim.x +
   ↪  j * gridDim.x + blockIdx.x]; },
15       [&] () -> void {
16          for (size_t a = 0; a < numGaussianas; a++) {
17             for (size_t b = 0; b < numDimensiones; b++) {
18                reducirFinal<BLOCK_SIZE>(
19                   [&] (size_t tid) -> double* { return &g_medias[a *
   ↪  numDimensiones * gridDim.x + b * gridDim.x + tid]; },
20                   [&] () -> double* { return &g_medias[a * numDimensiones +
   ↪  b]; },
21                   sharedData, gridDim.x);
22                if (threadIdx.x == 0) g_medias[a * numDimensiones + b] /=
   ↪  g_sumaProbabilidades[a];
23             }
24          }
25       }, n, sharedData, gridDim.x * gridDim.y * gridDim.z);
26 }
```

Para empezar, en las líneas 6 y 7, vemos la definición de los índices necesarios para acceder a los *arrays* en la memoria global.

En la línea 12, tenemos la llamada a la función `void reducir(...)`. Aquí vemos la utilidad de las expresiones lambda. En el primer argumento, indicamos que queremos sumar el producto `g_resp[k * n + i] * g_datos[j * n + i]` para todas las muestras. A continuación, indicamos la dirección de retorno en memoria global, como ya se había visto anteriormente.

De nuevo, en la línea 15, vemos la flexibilidad que proveen las expresiones lambda. En este tercer parámetro estamos indicando el proceso que se ha de ejecutar una vez realizada la reducción inicial. Concretamente, pasamos un bloque de código con dos bucles `for` anidados, que se encargarán de llamar a la función `void reducirFinal(...)` para obtener el resultado final de la media de cada dimensión para cada gaussiana. Por último, calculamos la división entre `g_sumaProbabilidades[a]`, tal y como hacíamos en la versión *CPU*.

CAPÍTULO 5
Resultados

Los resultados expuestos a continuación han sido obtenidos en la misma máquina con el mismo número de procesos en segundo plano y en el mismo estado, es decir, exclusivamente ejecutando el algoritmo, la máquina se puede consultar en el apéndice A.1, el *dataset* utilizado se compone de un fichero de entrada de $3 * 10^4$ datos y 23 dimensiones, los resultados aquí representados son la media de 20 ejecuciones.

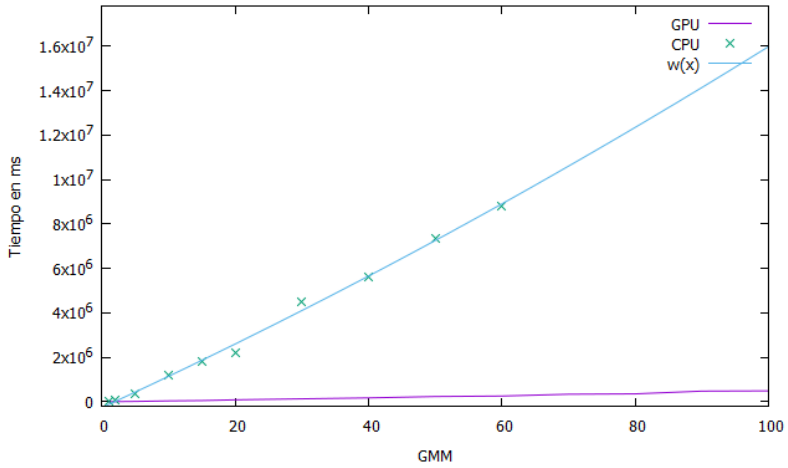

Figura 5.1: Tiempo GPU vs CPU

La gráfica 5.1 representa de forma visual los tiempos obtenidos del algoritmo de *CPU* y *GPU* en términos de tiempo, en ella vemos como poniendo en valor una comparación entre ambos la gráfica muestra un crecimiento mucho mayor en *CPU*.

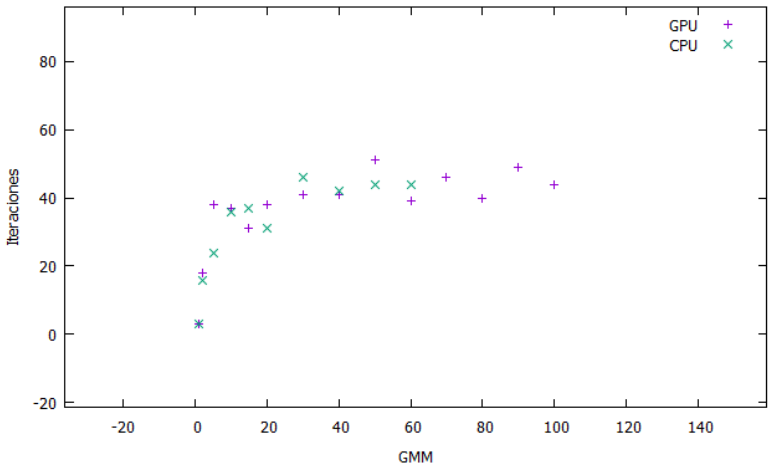

Figura 5.2: Iteraciones CPU vs GPU

En la Figura 5.2 podemos observar una comparativa entre el número de iteraciones necesarias para que el algoritmo converja, como vemos el número de iteraciones son "similares" para ambas implementaciones.

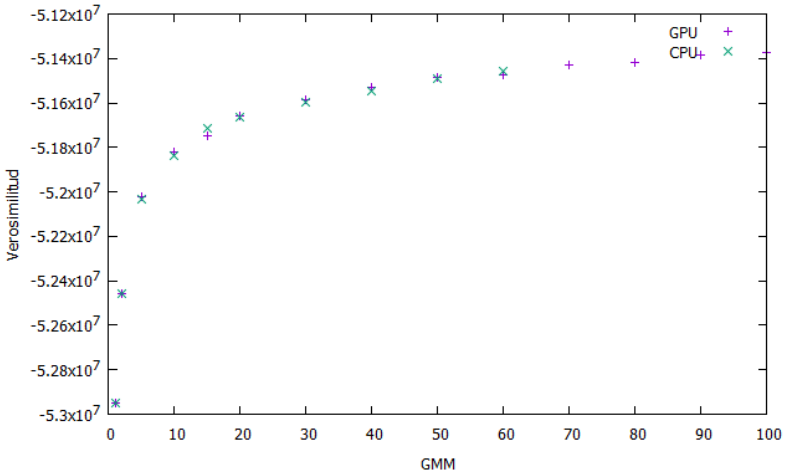

Figura 5.3: Verosimilitud CPU vs GPU

En la Figura 5.3 encontramos una comparación de la verosimilitud encontrada al converger el algoritmo, como vemos los resultados muy parejos.

Es importante poner en contexto los resultados expuestos hasta ahora, ya que un algoritmo X el cual convergiera prácticamente de inmediato arrojando un resultado en muy pocas iteraciones y en poco tiempo, pero el valor que devolviera de verosimilitud no fuera coherente sería un algoritmo que no estaría calculando el algoritmo *EM* adecuadamente, en estos resultados vemos como la verosimilitud devuelta por ambas implementaciones es casi idéntica, esto indica que el algoritmo calcula la misma verosimilitud pero en un tiempo mucho menor en el caso de la *GPU*.

Obtenidos los tiempos de las distintas ejecuciones, podemos saber que el *speed-up* conseguido se encuentra entre un $30x$ y $35x$ una vez se estabiliza con valores medios-altos de componentes de *GMM*, con valores más contenidos la diferencia es menor entorno a 14-$27x$, como se puede ver en la tabla 5.

GMM	Speed-up
1	13,86
2	12,76
5	23,56
10	27,26
15	33,12
20	29,98
30	33,88
40	32,88
50	31,20
60	34,25

Se han realizado diversas pruebas en las mismas condiciones redactadas al principio de esta Sección con distintos *datasets* de $5x10^4$, $2x10^4$ elementos y 23 dimensiones, 10^5 elementos y 23/21/12 dimensiones, así como también en 50000 elementos y 12 dimensiones, todos los resultados arrojan *speed-up* parecidos en torno a 32-35x, siempre con valores de la verosimilitud correctos, se ha observado que cuanto menor es el tamaño del *dataset* es necesario un mayor número de *GMMs* para que alcance ese *speed-up* y viceversa, aunque no se ha querido recargar este apartado exponiendo todas las pruebas y resultados llevados a cabo, debido a que arrojan resultados muy parejos todos ellos.

La siguiente Figura 5.4 si bien no es exactamente un resultado en cuanto a tiempo o verosimilitud si que es interesante en cuanto a lo que muestra. Existe una herramienta de *NVIDIA* que analiza algunos aspectos interesantes de la ejecución de un programa en *CUDA*, es el denominado *NVIDIA Visual Profiler* [23], el cual resulta fundamental entre otras opciones para *debugear* el código, aunque hay que poner siempre en contexto los resultados ofrecidos por dicha herramienta. En la Figura 5.4 se advierte la importancia/ocupación de los distintos *kernels* durante la ejecución, es decir, cual de los *kernels* consume más tiempo de ejecución.

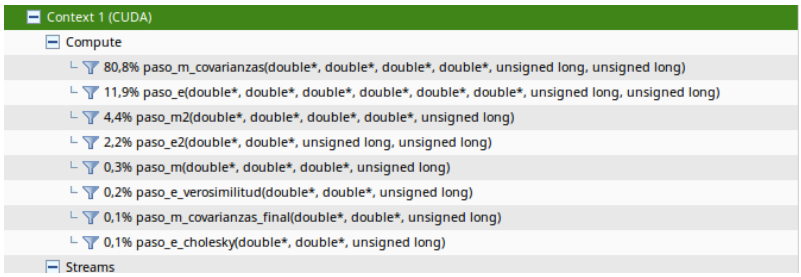

Figura 5.4: Porcentaje de carga en los distintos kernels

Como se ve claramente el proceso más "caro" computacionalmente del algoritmo es el paso M donde se calculan las covarianzas, el cual merece un análisis más detallado.

Figura 5.5: Análisis Visual Profiler

El análisis de *NVIDIA Visual Profiler* nos indica, como vemos en las imágenes de la Figura 5.5, que la carga de los distintos *SM* es del 100 %, y el número de hilos utilizados es el máximo, 1024 hilos.

CAPÍTULO 6
Conclusiones

CUDA es una extensión de C++ muy interesante de cara a la programación paralela sobre *GPUs NVIDIA*. La curva de aprendizaje es rápida al inicio gracias a que se dispone de muchas librerías que incluyen implementaciones de la mayoría de funciones que necesitamos, como por ejemplo *cuBLAS* [24]. También hay disponible una detallada documentación. Sin embargo, el problema aparece cuando las librerías disponibles no ofrecen lo que requerimos, bien porque la implementación disponible no se ajusta exactamente a nuestras necesidades, o bien porque directamente no existe dicha implementación.

El reto aparece cuando se necesita profundizar en *CUDA* para programar un algoritmo del cual no existen implementaciones disponibles. La complejidad a la que se enfrenta el programador es muy alta debido a la estructura de los distintos tipos de memoria dentro de las GPUs, cuyo detalle es de difícil comprensión. Otro punto a tener en cuenta es la gran dificultad para *depurar* código *CUDA*, pues en un programa *CUDA* hay multitud de hilos ejecutándose simultáneamente, lo que puede dar lugar a múltiples condiciones de carrera.

En lo relativo exclusivamente a este proyecto, hemos visto como con un hardware de hace 2 años cuyo coste fue de 100 euros aproximadamente, se pueden obtener grandes mejoras en cuanto a *speed-up*. No obstante, debemos decir que el *speed-up* de $30x$ obtenido ha sido con respecto a un core de la *CPU*, dado que nuestro código para *CPUs* no está adaptado para aprovechar todos los *cores* disponibles, en cuyo caso estaríamos hablando de una ganancia inferior a $30x$ utilizando *GPUs*.

Todo hardware tiene un límite, y no siempre será posible ubicar todo un *dataset* en la memoria de la *GPU*. Por lo general, los *datasets* se componen de gran número de muestras, y éstas suelen ser vectores de características con muchas componentes, en algunos casos incluso miles. Por tanto, no siempre será posible utilizar nuestra solución, tal como se ha implementado será útil para *datasets* relativamente pequeños (número de muestras en $\mathbb{R}^d \leq 10^6$, con $d \leq 50$) y *GMMs* de hasta 100 Gaussianas.

Las posibles mejoras se enfocarían a lo comentado en el apartado 5 de Resultados, en lo referente al *kernel* que más tiempo de cómputo consume, alrededor del 80 % del total. Bien repartiendo los cálculos de dicho *kernel* entre varias *GPUs*, o bien tratar de optimizar el código de dicho *kernel*.

Bibliografía

[1] Duke University. *Computational Statistics in Python*. Jul. de 2016. URL: https://people.duke.edu/~ccc14/sta-663/EMAlgorithm.html.

[2] Shu-Ching Chang y Hyung Jin Kim. *EM Algorithm*. Jul. de 2016. URL: http://homepage.divms.uiowa.edu/~kcowles/s166_2007/chang166.pdf.

[3] Chris McCormick. *Gaussian Mixture Models Tutorial and MATLAB Code*. Jul. de 2016. URL: http://mccormickml.com/2014/08/04/gaussian-mixture-models-tutorial-and-matlab-code/.

[4] "Zilla". *EM.java*. Jul. de 2016. URL: http://scribblethink.org/Computer/Javanumeric/EM.java.

[5] Dirk Eddelbuettel. *Implementing an EM Algorithm for Probit Regressions*. Jul. de 2016. URL: http://gallery.rcpp.org/articles/EM-algorithm-example/.

[6] Ernesto Insua-Suárez y Marlis Fulgueira-Camilo. *Paralelización del Algoritmo Expectación-Maximización Utilizando OpenCL*. Sep. de 2014.

[7] Mortaza Jamshidian y Robert I. Jenrrich. *Acceleration of the EM algorithm by Using Quasi-Newton Methods*. Jul. de 2016. URL: http://www.quretec.com/u/vilo/edu/2003-04/DM_seminar_2003_II/ver1/P10/artiklid/JamshidianJennrich.pdfs.

[8] Wikipedia. *CUDA*. Jun. de 2016. URL: https://en.wikipedia.org/wiki/CUDA.

[9] NVIDIA. *Aplicaciones para la GPU*. Mar. de 2016. URL: http://www.nvidia.es/object/gpu-computing-applications-es.html.

[10] NVIDIA. *Programming guide*. Feb. de 2016. URL: http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz4HzKEADvl.

[11] Andres Rondan. *CUDA: MODELO DE PROGRAMACIÓN*. Mar. de 2016. URL: http://dis.um.es/~domingo/apuntes/AlgProPar/0910/exposicion1/RondanAndres-CUDA.pdf.

[12] NVIDIA. *Ejemplos prácticos CUDA*. Enero de 2016. URL: http://docs.nvidia.com/cuda/cuda-samples.

[13] NVIDIA. *An Easy Introduction to CUDA C and C++*. Mayo de 2016. URL: https://devblogs.nvidia.com/parallelforall/easy-introduction-cuda-c-and-c/.

[14] NVIDIA. *CUDA C BEST PRACTICES GUIDE*. Mayo. NVIDIA, 2011.

[15] PGI Insider. *Understanding the CUDA Data Parallel Threading Model*. Mayo de 2016. URL: https://www.pgroup.com/lit/articles/insider/v2n1a5.htm.

[16] Ramesh Sridharan. *Optimizing Parallel Reduction in CUDA*. Feb. de 2016. URL: http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf.

[17] Robert Hochberg. *Matrix Multiplication with CUDA — A basic introduction to the CUDA programming model*. Mayo de 2016. URL: https://www.shodor.org/media/content/petascale/materials/UPModules/matrixMultiplication/moduleDocument.pdf.

[18] Wikipedia. *Expectation–maximization algorithm*. Feb. de 2016. URL: https://en.wikipedia.org/wiki/Expectation%E2%80%93maximization_algorithm.

[19] Wikipedia. *Estimación, Maximación numérica*. Mar. de 2016. URL: http://eml.berkeley.edu/books/choice2nd/C8.pdf.

[20] Maya R. Gupta y Yihua Chen. *Theory and Use of the EM Algorithm*. 2.ª ed. now, 2010.

[21] William H. Press y col. *Numerical Recipes The art of Scientifc Computing*. 846.ª ed. Cambridge, 2010.

[22] Wikipedia. *Mersenne Twister*. Jun. de 2016. URL: https://en.wikipedia.org/wiki/Mersenne_Twister.

[23] NVIDIA. *Profiler User's Guide*. Jun. de 2016. URL: https://developer.nvidia.com/nvidia-visual-profiler.

[24] NVIDIA. *cuBLAS*. Mar. de 2016. URL: https://developer.nvidia.com/cublas.

APÉNDICE A
Configuración del sistema

A.1 Especificaciones técnicas

La configuración del equipo donde se han realizado las pruebas y se han medido los tiempos es el siguiente:

- AMD FX-6300 3,5GHz
- Asrock 970 Extreme R2.0
- Gigabyte GeForce GTX 750 OC 1GB GDDR5:
  - GTX 750 GPU Engine Specs:
  - 512CUDA Cores
  - 1020Base Clock (MHz)
  - 1085Boost Clock (MHz)
  - Compute Capability 5.0
- GTX 750 Memory Specs:
  - 5.0 GbpsMemory Clock
  - 1024 MBStandard Memory Config
  - GDDR5Memory Interface
  - 128-bitMemory Interface Width
  - 80 Memory Bandwidth (GB/sec)