UNIVERSIDADE DA CORUÑA

Facultade de Informática
Departamento de Electrónica e Sistemas

PROXECTO FIN DE CARREIRA
Enxeñería Técnica en Informática de Sistemas

Computación paralela en un entorno heterogéneo CPUs-GPUs

Alumno: Iago López Galeiras
Directores: Emilio José Padrón González
Bruno Raffin
Fecha: 1 de octubre de 2010

D. Emilio José Padrón González
Profesor Contratado Doctor
Departamento de Electrónica e Sistemas
Universidade da Coruña

CERTIFICA: Que la memoria titulada *"Computación paralela en un entorno heterogéneo CPUs-GPUs"* ha sido realizada por Iago López Galeiras bajo mi dirección y constituye su Proxecto Fin de Carreira da Enxeñería Técnica en Informática de Sistemas.

En A Coruña, a 1 de octubre de 2010

D. Emilio José Padrón González
Director del proyecto

Resumen:

La STL (*Standard Template Library*) es una de las herramientas más utilizadas por los programadores de C++ por su carácter genérico e inclusión de algoritmos y estructuras de uso frecuente. Por otra parte, el límite físico alcanzado en el aumento de frecuencias de trabajo de las CPUs ha provocado la aparición de procesadores multinúcleo, lo que ha extendido el procesamiento paralelo del ámbito específico de la computación de altas prestaciones a otro tipo de entornos más generales. Debido a este renovado interés por el paralelismo han aparecido las tecnologías GPGPU (*General-Purpose Computing on Graphics Processing Units*), que han permitido a las unidades de procesamiento de gráficos (GPUs) realizar computación de propósito general, aprovechando su un alto grado de paralelismo. En este contexto, CUDA es una de las tecnologías GPGPU más maduras, exitosas y eficientes de la actualidad.

El objetivo de este proyecto es implementar algoritmos de la STL de manera que aprovechen las capacidades de los entornos paralelos multiCPU y multiGPU con las tecnologías POSIX Threads y CUDA. De esta forma, proporcionando una implementación paralela eficiente de la STL se le ofrece al programador una interfaz ya conocida y muy generalizada para algoritmos y estructuras de uso común en programación que le abstrae de las características físicas de la plataforma paralela a explotar.

Palabras clave:

Procesamiento paralelo, Multinúcleo, GPGPU, CUDA, C++, STL, POSIX Threads, Boost.

Para Paula,
por apoyarme incondicionalmente tanto desde aquí como desde fuera.

Agradecimientos

Me gustaría agradecer a mis padres y a mis amigos sus ánimos y su fe inquebrantable en que acabaría este proyecto a tiempo. Por supuesto agradecer también a mis directores Bruno Raffin por ayudarme a enfocar el proyecto en la dirección que ha tomado y Emilio Padrón por sus acertadas correcciones y sugerencias, especialmente en el *sprint* final. Muchas gracias a todos.

Índice general

Índice de cuadros

Índice de figuras

Capítulo 1

Introducción

En este capítulo hablaremos brevemente sobre los objetivos marcados para este proyecto y la organización de esta memoria.

1.1. Objetivos

El objetivo final de este proyecto es proporcionar una biblioteca de programación que permita abstraer al programador de la complejidad de un sistema multiprocesador heterogéneo moderno, formado por procesadores multinúcleo (CPUs) y una o varias tarjetas gráficas con soporte para cálculo de propósito general (GPGPU). Para alcanzar este objetivo se ha adoptado la filosofía de partir de un API muy extendida, como es la de la biblioteca STL para C++. De esta forma, en este proyecto se ha realizado una implementación de los algoritmos **transform** y **accumulate** de la librería STL de forma que aprovechen las capacidades paralelas de los sistemas multiCPU y multiGPU. Para ello se usarán las tecnologías POSIX Threads para el manejo de hilos en CPU y CUDA para aprovechar las capacidades GPGPU de las GPUs de NVIDIA.

Como objetivos intermedios que ha sido necesario alcanzar para realizar esta implementación podemos enumerar: el aprendizaje de C++, uso e implementación de la librería STL, programación multi-hilo con POSIX Threads y programación paralela en CUDA.

1.2. Organización

Esta memoria está organizada de la siguiente manera:

- El Capítulo 2 hace una introducción al procesamiento paralelo: clasificación, tipos y

lenguajes de programación paralelos.

- El Capítulo 3 se centra en explicar brevemente la arquitectura de las GPUs, centrándose en la tecnología CUDA de NVIDIA: su modelo de programación y arquitectura hardware.
- El Capítulo 4 describe brevemente el lenguaje de programación C++ y sus características más interesantes para este proyecto y la STL, especialmente los algoritmos **transform** y **accumulate** por ser los elegidos para implementar en este proyecto.
- El Capítulo 5 explica con detalle la implementación realizada de los algoritmos de la STL.
- El Capítulo 6 muestra y analiza los resultados de rendimiento obtenidos por nuestra implementación.
- El Capítulo 7 expone las conclusiones de este proyecto y posibles trabajos futuros.

Capítulo 2

Procesamiento paralelo

En este capítulo daremos una visión general sobre el procesamiento paralelo: su razón de ser, complejidades y tipos de sistemas paralelos y su programación. De este modo situaremos en contexto las tecnologías empleadas en el proyecto.

2.1. Introducción

El procesamiento paralelo es una forma de proceso en la que varios cálculos se ejecutan de forma simultánea [12]. Para conseguirlo se apoya en el principio *divide y vencerás*, es decir, los problemas grandes se pueden descomponer en problemas más pequeños tratables de forma concurrente. De esta forma, aunque la suma del tiempo que todos los subproblemas que componen el problema utilizan sea la misma que el tiempo que el propio problema emplea, al ejecutarse de forma paralela el rendimiento es mayor. El límite físico alcanzado en el aumento de frecuencias de trabajo de los procesadores y los valores cada vez más altos de generación de calor y consumo ha extendido el procesamiento paralelo del ámbito de la computación de altas prestaciones (*High-Performance Computing* o *HPC*) a todo tipo de entornos: domésticos (principalmente en forma de procesadores multicore), servidores, etc.

La programación de sistemas paralelos es significativamente más difícil porque introduce varios problemas que no tiene la programación secuencial, como problemas de sincronización y comunicación entre las diferentes subtareas. No obstante, ofrece a cambio un aumento de rendimiento que, en general, se puede modelar de forma simple con la ley de Amdahl aplicada al paralelismo:

$$S = \frac{1}{(1-P) + P/N} \tag{2.1}$$

Donde S es la máxima aceleración que se puede conseguir, N es el número de procesadores o núcleos computacionales, P es la proporción paralelizable del programa y $(1-P)$ es la proporción no paralelizable. Así, la ley de Amdahl permite predecir la aceleración (*speedup*) máxima teórica que puede alcanzar un código paralelo en un sistema multiprocesador.

2.2. Clasificación de sistemas para el procesamiento paralelo

Existen varios enfoques para clasificar las distintas alternativas existentes dentro del mundo de las arquitecturas paralelas y la computación de altas prestaciones. A continuación describimos los que gozan de una mayor aceptación y nos permiten encuadrar el trabajo realizado en este proyecto.

2.2.1. Taxonomía de Flynn

Una de las formas tradicionales de clasificar las arquitecturas paralelas atiende a las diferentes combinaciones del lugar en el se aplique el paralelismo: en instrucciones o en datos. Esta clasificación se llama taxonomía de Flynn y fue uno de los primeros sistemas para clasificar los programas y ordenadores. Estas combinaciones, que se ejemplifican gráficamente en la Figura 2.1 son:

- **SISD** (*Single Instruction, Single Data stream*): No aprovecha el paralelismo, es un modelo secuencial tradicional. Es el caso de los procesadores de un solo núcleo.
- **SIMD** (*Single Instruction, Multiple Data stream*): Aplica una misma serie de instrucciones a un conjunto diferente de datos. Este es el caso de las unidades vectoriales de las CPUs en un sistema monoprocesador y también uno de los tipos de paralelismo presentes en las tarjetas gráficas (en adelante GPUs) modernas.
- **MISD** (*Multiple Instruction, Single Data stream*): Es poco común, aplica diferentes instrucciones al mismo conjunto de datos. Suele usarse para tolerancia a fallos, se comprueba que todas las aplicaciones de diferentes instrucciones devuelven los mismos datos.
- **MIMD** (*Multiple Instruction, Multiple Data stream*): Aplican diferentes instrucciones, simultáneamente en ejecución, a un conjunto diferente de datos. Entre este tipo

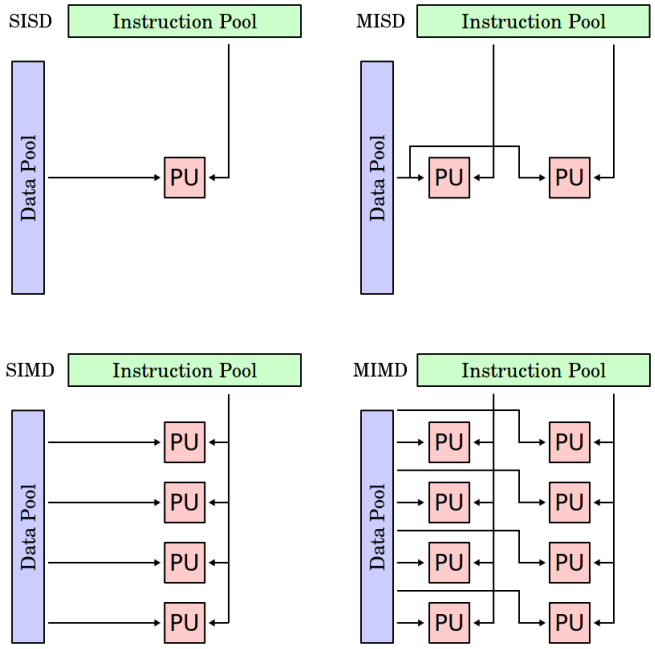

Figura 2.1: Taxonomía de Flynn. Cada PU es una unidad de procesamiento [4]

de sistemas encontramos, por ejemplo, arquitecturas multiprocesador homogéneas, tanto en configuraciones de memoria compartida como distribuida.

2.2.2. Tipos de paralelismo

Otra manera de clasificar el paralelismo es según su tipo. Paralelismo a nivel de bit, a nivel de instrucción, a nivel de datos y a nivel de tareas.

Paralelismo a nivel de bit Viene dado por el tamaño de palabra utilizado en el procesador: Cuanto más grande sea, menos instrucciones serán necesarias para realizar una operación. Por ejemplo, cuando un procesador de 8 bits tenía que hacer una operación de números de 16 bits la operación tenía que dividirse, mientras que uno de 16 bits podría hacerla en un solo paso.

Paralelismo a nivel de instrucción Un programa es, esencialmente, una serie de instrucciones que se ejecutan una detrás de otra. Estas instrucciones pueden dividirse en varias etapas para ser ejecutadas en forma de *tubería* o *pipeline* en el procesador de forma solapada; un procesador con un *pipeline* de N etapas puede ejecutar N instrucciones a

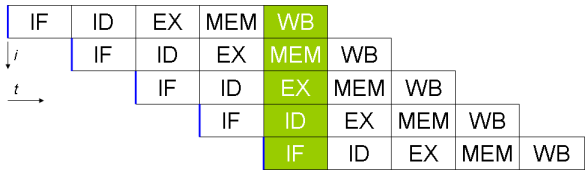

Figura 2.2: Pipeline de 5 etapas [3]

la vez, cada una en diferentes etapas como se puede ver en la Figura 2.2. Además del *pipelining* existen otras técnicas de paralelismo a nivel de instrucción como la ejecución superescalar, en la que se pueden emitir dos instrucciones a la vez gracias a la duplicación de unidades funcionales del procesador; la ejecución fuera de orden (ya sea dinámicamente o en tiempo de compilación), que se basa en cambiar de orden las instrucciones sin dependencias para mantener el procesador ocupado mientras se espera por los datos de alguna otra instrucción; o el renombrado de registros, usado para no tener que parar el *pipeline* si se reutilizan los mismos registros del procesador.

Paralelismo a nivel de datos Es el paralelismo que existe en los bucles, en los que los datos se pueden procesar en diferentes nodos siempre que no existan dependencias con iteraciones anteriores. También son paralelismo a nivel de datos las instrucciones vectoriales como SSE o Altivec o la computación en GPGPU, utilizada en este proyecto y de la que se habla en profundidad en el Capítulo 3.

Paralelismo a nivel de tareas Se centra en ejecutar cada hilo (*thread*) o proceso en un procesador (o unidad de procesamiento) diferente, compartiendo o no la misma memoria. Sus programas son independientes pero pueden comunicarse utilizando, por ejemplo, variables compartidas que deberán ser protegidas ante accesos concurrentes en el caso de memoria compartida, o mediante paso de mensajes en sistemas con memoria distribuida. Esta clase de procesamiento paralelo también es utilizada en este proyecto.

2.2.3. Clases de arquitecturas paralelas

Las arquitecturas paralelas pueden dividirse en *Uniform Memory Access* (UMA), en la que se puede acceder a cada elemento de la memoria principal con la misma latencia, y *Non-Uniform Memory Access* (NUMA), en la que las latencias cambian. Normalmente una arquitectura UMA se consigue con memoria compartida entre los elementos de la arquitectura y NUMA con memoria distribuída.

A su vez, estas arquitecturas pueden clasificarse de acuerdo al nivel de hardware que ofrece o expone el paralelismo:

Arquitecturas multinúcleo (*multicore*) Los procesadores multinúcleo son aquellos que contienen varias unidades de ejecución en el mismo chip. Suelen tener un bus y una cache compartida (además de uno o varios niveles cache locales en cada núcleo). Son responsables de la popularización actual del paralelismo, al ser la principal solución empleada por los fabricantes de procesadores para mantener el ritmo en el aumento del rendimiento ante la aparición de limitaciones físicas en la tecnología actual que han detenido los contínuos incrementos en la frecuencia de reloj de los procesadores de los últimos años.

Arquitecturas SMP Son similares a las multinúcleo, pero esta vez cada unidad es un procesador completo y se comunican entre sí a través de un bus además de compartir la memoria como las arquitecturas multinúcleo. Aunque han sido una de las arquitecturas paralelas clásicas, no son muy escalables debido a que tienen que competir por el bus de conexión. Hoy en día se complementan con arquitecturas multinúcleo.

Computación distribuida Es un sistema NUMA con memoria distribuída y los elementos se conectan entre sí a través de una red. Es una arquitectura muy escalable y tolerante a fallos. La programación de estos sistemas suele ser más compleja que la de sistemas con memoria compartida.

Arquitecturas paralelas especializadas Son arquitecturas utilizadas para problemas paralelos concretos. Este es el caso de los de procesadores vectoriales (ya en desuso) y de la distintas tecnologías GPGPU (*General Purpose Computing on Graphics Processing Units*), centradas en el uso de las GPUs para la computación de propósito general.

En este proyecto abordamos la paralelización de sistemas paralelos heterogéneos, compuestos por procesadores multinúcleo con acceso a una memoria compartida y una o más tarjetas gráficas con capacidades GPGPU.

2.3. Lenguajes de programación paralelos

Las soluciones existentes para la programación y aprovechamiento de los sistemas multiprocesador todavía se encuentran a cierta distancia en cuanto a sencillez y robustez de los paradigmas y herramientas existentes para la programación de sistemas puramente secuenciales. No obstante, a la espera de disponer algún día de buenas herramientas para

la paralelización automática, existen en la actualidad bastantes alternativas para lograr implementaciones paralelas eficientes.

La mayor parte de las soluciones existentes para la programación de sistemas multiprocesador ofrecen un API con funciones y directivas pensada para ser utilizada desde un lenguaje de programación de propósito general como C/C++, Fortran, Java, etc. Las características de la plataforma y el tipo de paralelismo a explotar determinan lo adecuado de las distintas soluciones.

Para la programación de plataformas de memoria distribuida suelen utilizarse alternativas basadas en paralelismo a nivel de tareas con paso de mensajes, como PVM o, sobre todo, MPI, que se ha convertido en un estándar de facto para la programación de máquinas de memoria distribuida. Se trata de una biblioteca multiplataforma, con una especificación estándar definida y libre [7] que tiene como objetivos permitir una elevada portabilidad del código y obtener implementaciones eficientes incluso en arquitecturas heterogéneas. Además, existen múltiples implementaciones disponibles, tanto libres (OpenMPI, MPICH2...) como de los principales fabricantes de computadores (Intel MPI, HP MPI...).

En el caso de sistemas de memoria compartida, como es el caso que nos ocupa en la parte CPU del proyecto, una programación multi-hilo permite explotar de una forma más simple que con el paso de mensajes la presencia de múltiples unidades de procesamiento. Alternativas para esto serían los APIs OpenMP 3.0 [6] e Intel Threading Building Blocks (TBBs) [10]. Ambas son dos bibliotecas de alto nivel que permiten abstraer el procesamiento paralelo con múltiples hilos, ofreciendo tanto paralelismo a nivel de tareas como a nivel de datos.

Para este proyecto, sin embargo, nos hemos decantado por el uso de la biblioteca POSIX Threads o Pthreads [5], estándar POSIX para la programación multihilo. Se trata de una biblioteca independiente de la plataforma que permite un acercamiento de bajo nivel al control de los hilos en ejecución, lo que nos ofrece la posibilidad de una gran flexibilidad a costa de, quizás, una mayor dificultad de programación. Así, la biblioteca definie un API con operaciones para manejo de hilos (creación, bloqueo...), regiones de exclusión mútua con mutexes (para evitar accesos simultáneos a variables compartidas o regiones críticas) y sincronización mediante barreras y bloqueos de lectura/escritura.

Específicamente, el código CPU del proyecto hace uso de Boost.Thread, abstracción construida por encima de la biblioteca POSIX Threads. Boost es un conjunto de bibliotecas para C++ que cubren un gran abanico de funciones de todo tipo. En concreto, Boost.Thread [1] es una abstracción portable de Pthreads más adecuada para su utilización en lenguajes orientados a objetos, ya que el API de Pthreads fue creada pensando en

el lenguaje C y la programación estructurada.

Para la programación de las GPUs presentes en el sistema hemos utilizado CUDA, que es la tecnología GPGPU de nVIDIA y es descrita en el Capítulo 3.

Capítulo 3

GPUs

En este capítulo hablaremos sobre las GPUs: sus capacidades de cálculo, su arquitectura y por qué es interesante trabajar con ellas. Nos centraremos en la arquitectura Tesla de NVIDIA en particular y en el entorno CUDA por ser lo utilizado en este proyecto.

3.1. Introducción

La GPU es un procesador especializado en gráficos que libera al procesador principal de las principales tareas relacionadas con la salida gráfica de un ordenador y, además, acelera esta clase de cálculos. El principal motivo para este mayor rendimiento en cuestiones gráficas es, obviamente, una mera cuestión de diseño: mientras que la CPU tiene que adaptarse a cualquier algoritmo, es decir, ser de propósito general, la GPU solo tiene que realizar una serie muy concreta de tareas, por lo tanto será más eficiente. Resultando de esta especialización su mayor eficiencia.

Las GPUs comenzaron acelerando el renderizado de polígonos y el mapeado de texturas, más tarde añadieron unidades para hacer cálculos geométricos y finalmente se volvieron programables, permitiendo realizar operaciones en los vértices y las texturas que hasta ese momento sólo se podían hacer en la CPU. La mayoría de estas operaciones se hacen sobre matrices o vectores, por lo que se empezó a considerar el uso de las GPUs para otro tipo de cálculos, no sólo para gráficos.

La GPUs programables actuales son procesadores altamente paralelos, multinúcleo y multitarea. Tienen una gran capacidad de cálculo en coma flotante (llegando a cientos de gigaflops) y mucho ancho de banda, como se puede ver en la Figura 3.1. La razón de esta diferencia vuelve a estar en la especialización de la GPU, al ejecutar la misma instrucción para cada dato de entrada (paralelismo a nivel de datos, descrito en la Sección 2.2.2), no

hay necesidad de dedicar tantos recursos al control de flujo como las CPUs. Por otro lado, dada la alta intensidad aritmética de cálculo (la cantidad de operaciones aritméticas es mucho mayor que la cantidad de de operaciones de memoria) el acceso a la memoria se puede enmascarar en los cálculos en lugar de en grandes caches, como en el caso de la CPU. La Figura 3.2 ilustra estas diferencias.

Este modelo de programación, típico de la informática gráfica, datos con alto paralelismo y alta intensidad aritmética, puede ser aprovechado por multitud de aplicaciones: desde renderizado 3D (la razón de ser de las GPUs), posprocesado de imágenes, codificación de vídeo hasta proceso de señales o simulaciones físicas. Así nace la GPGPU (*General-Purpose Computing on Graphics Processing Units*), para aprovechar la potencia de cálculo de las GPUs en aplicaciones de propósito general.

El aprovechamiento efectivo de todo el potencial que las tarjetas gráficas actuales nos proporcionan requiere, sin embargo, de herramientas y paradigmas de programación que permitan una eficaz abstracción del hardware, proporcionando una interfaz que favorezca al diseño de aplicaciones complejas que puedan hacer uso de forma eficiente de las características de las GPUs modernas. Un paradigma de programación que se ajusta bastante bien a los recursos computacionales que nos ofrecen las GPUs con capacidades GPGPU es el clásico *Stream Processing*, que proporciona una manera sencilla de explotar el paralelismo a nivel de datos. Su aplicación efectiva presenta, no obstante, muchos retos a resolver para conseguir implementaciones eficientes en la GPU. Así, una primera aproximación a la implementación de un algoritmo sobre la GPU puede no ofrecer el rendimiento esperado y requerir un gran trabajo de optimización por parte del programador para un buen aprovechamiento de los recursos de la GPU.

Hace ya aproximadamente una década de la aparición de las primeras GPUs programables con *shaders*, tecnología que motivó el desarrollo de los primeros intentos de lenguajes de programación para GPUs como GLSL (*OpenGL Shading Language*) o NVIDIA Cg (*C for Graphics*). Estas tecnologías eran todavía APIs orientadas exclusivamente a tareas gráficas, que exigían una programación de bajo nivel al exponer la estructura y las limitaciones del *pipeline* gráfico directamente al programador. No obstante, en los últimos años han aparecido varias propuestas para el cálculo de propósito general en GPUs, ya bajo el paraguas de la denominación GPGPU, como OpenCL, propuesta que trata de ofrecer una solución multiplataforma para la programación paralela en sistemas heterogéneos compuestos por procesadores multinúcleo y una o varias GPUs, pero cuyas primeras implementaciones se encuentran todavía lejos de obtener buenos rendimientos. Nos centraremos en CUDA, tecnología GPGPU elegida para este proyecto.

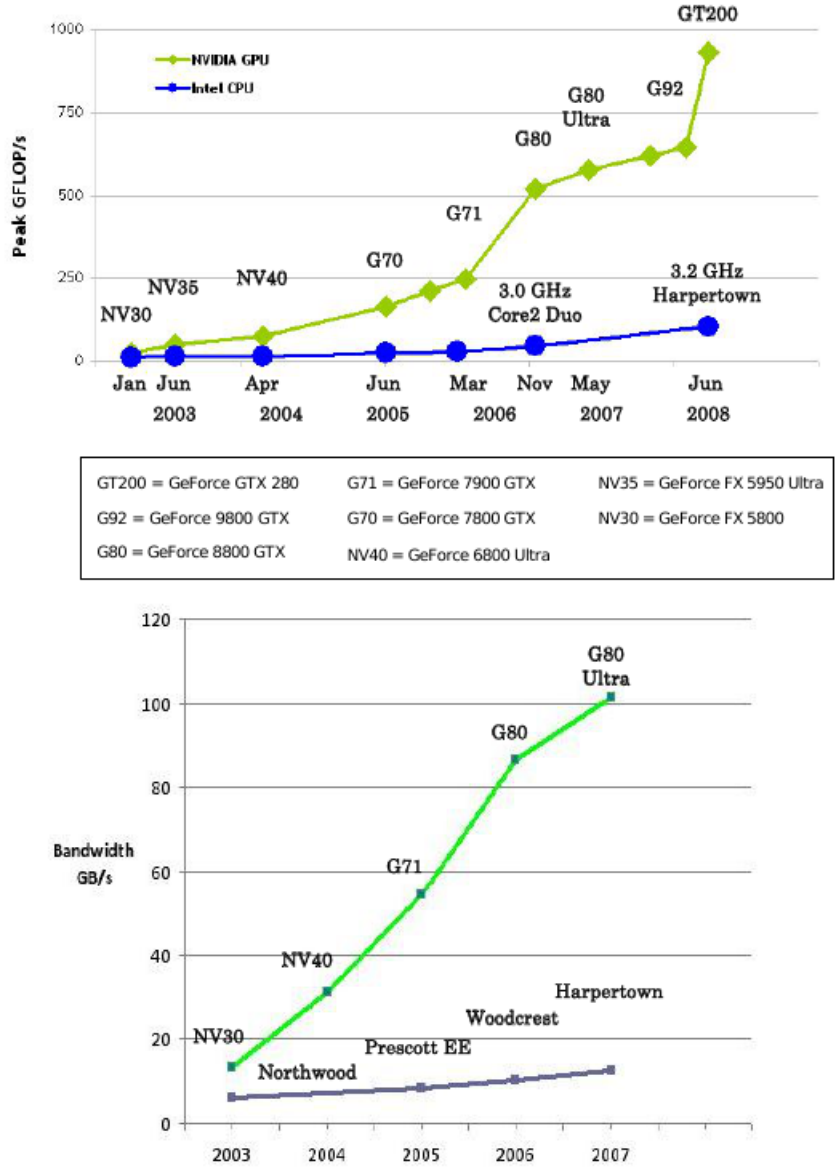

Figura 3.1: Operaciones en coma flotante por segundo y ancho de banda de CPU y GPU [9]

Figura 3.2: Dedicación de transistores en CPU y GPU [9]

3.2. Arquitectura de las GPUs

Tanto las GPUs como, en menor medida, las CPUs utilizan la técnica SIMD (*Single Instruction, Multiple Data*). Como ya hemos comentado en la Sección 2.2, consiste en aplicar la misma instrucción (*Single Instruction*) a una cantidad grande de datos (*Multiple Data*).

En el caso de las CPUs podemos encontrar esta técnica en las instrucciones vectoriales que incluyen, como por ejemplo las diferentes SSE (*Streaming SIMD Extensions*) de Intel o 3DNow! de AMD. Consisten en varios registros de un tamaño de varias palabras en los que se introducen los números y se opera con ellos de forma vectorial, permitiendo así realizar varias operaciones tradicionales en un solo ciclo de reloj.

En el caso de las GPUs, este paralelismo a nivel de datos se lleva a una escala más grande permitiendo muchas más operaciones simultáneas al disponer de muchos núcleos con varias ALUs por núcleo. Cada secuencia de instrucciones (*stream*) se ejecuta en todos los núcleos en paralelo en el marco de un contexto de ejecución.

Otra diferencia con las CPUs es la forma de afrontar los bloqueos, que se producen cuando un procesador no puede emitir la siguiente instrucción en el *stream* debido a una dependencia. Para evitar los bloqueos provocados por las grandes latencias de memoria, las CPUs utilizan las memorias cache; en cambio, las GPUs mantienen más contextos de ejecución de los que pueden manejar a la vez para emitir instrucciones de otros contextos cuando hay uno bloqueado. Las GPUs tienen caches más pequeñas que utilizan para aprovechar que cuando se aplica un *shader*, se hace sobre fragmentos próximos de la pantalla y se aprovecha esa localidad espacial en 2D al cargar las texturas.

Como ejemplo de la diferencia de rendimiento entre CPUs y GPUs, una NVIDIA GeForce GTX 480 puede alcanzar un pico de unos 1300 GFLOPS mientras que un Intel Core

i7 980 XE puede como mucho llegar a 107.55 GFLOPS.

3.3. CUDA

CUDA (*Compute Unified Device Architecture*) es la arquitectura de computación paralela de NVIDIA. Compatible en menor o mayor medida con todas sus tarjetas gráficas de los últimos años, está basada en un *driver* y un *runtime*, que junto a una biblioteca de funciones permiten mediante una extensión de C aplicar *Stream Processing* para explotar el paralelismo presente en la GPU, que se expone mediante una abstracción de su arquitectura. El objetivo principal de esta abstracción y los mecanismos que CUDA nos proporciona para su explotación es permitir el desarrollo de aplicaciones que escalen su rendimiento de forma transparente con un mayor número de núcleos computacionales en la GPU. Otro objetivo es ofrecer facilidad de aprendizaje para programadores familiarizados con el lenguaje C. La base de la abstracción de la GPU que CUDA ofrece se basa en tres pilares: una jerarquía de grupos de hilos (que pueden ser miles), memorias compartidas y sincronización por barreras. Estas abstracciones proporcionan paralelismo de grano fino y a nivel de hilo dentro del paralelismo grueso y a nivel de tarea. Esta descomposición permite al programador dividir el problema en varios subproblemas y permite escalabilidad transparente, ya que CUDA se encarga de repartir las tareas.

3.3.1. Modelo de programación

CUDA proporciona dos APIs diferentes y mútuamente exclusivas para la programación de la GPU. La primera de ellas es el API del *driver* CUDA. Es de bajo nivel, ofreciendo un gran control y flexibilidad de los trabajos en la GPU, aunque con una mayor dificultad de programación. El API del *driver* permite a las funciones cargar los kernels como módulos binarios o ensamblador de CUDA, por lo tanto, se pueden compilar los kernels por separado y llamarlos desde el programa que los utilice. Es necesario utilizarlo para programar en un lenguaje distinto de C, por lo que será el API utilizado en este proyecto. El segundo API es el del *runtime* CUDA, está construida sobre la anterior y facilita las tareas de programación.

En el API del *runtime*, CUDA utiliza una extensión del lenguaje C llamada *C for CUDA* para definir *kernels* que se ejecutarán N veces en paralelo por N hilos de CUDA. Un kernel se define usando la palabra clave **__global__** y el número de hilos en cada llamada se especifica utilizando la nueva sintaxis <<< ... >>>. Además, cada hilo tiene un identificador único accesible mediante la variable **threadIdx** (que es un vector de 3 componentes por comodidad del programador).

Sirva el siguiente fragmento de código como ejemplo de programación e invocación de un kernel CUDA:

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

int main()
{
    ...
    // Invocación del kernel
    VecAdd<<<1, N>>>(A, B, C);
}
```

En este ejemplo, cada hilo que ejecute VecAdd() sumará un elemento del vector A con otro del B.

Cada grupo de hilos forma un bloque de una, dos o tres dimensiones (según las dimensiones que tenga el **threadIdx**). El índice de cada hilo y su identificador (*thread ID*) se relacionan el uno con el otro de forma sencilla: Para un vector de una dimensión son el mismo, para uno de dos dimensiones de tamaño (D_x, D_y) el *thread ID* de un índice de hilo (x, y) es $(x + yD_x)$ y para un vector de tres dimensiones de tamaño (D_x, D_y, D_z) el *thread ID* de un índice de hilo (x, y, z) es $(x + yD_x + zD_xD_y)$. El número máximo de hilos por bloque depende de la memoria del núcleo, en la actualidad es 512 hilos. Los hilos dentro de un bloque pueden cooperar entre ellos utilizando la memoria compartida (*shared memory*, ver 3.3.1.1) y sincronizando puntos de ejecución con **__syncthreads()** para coordinar los accesos a memoria.

Además, el conjunto de bloques está organizados en una rejilla (*grid*) de una o dos dimensiones (Figura 3.3) y para identificar un bloque en una rejilla se utiliza la variable **blockIdx** y la dimensión del bloque se obtiene consultando la variable **blockDim**.

Con todo lo anterior en cuenta, así quedaría una suma de matrices en CUDA:

```
// Definición del kernel
__global__ void MatAdd(float* A, float* B, float* C)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
```

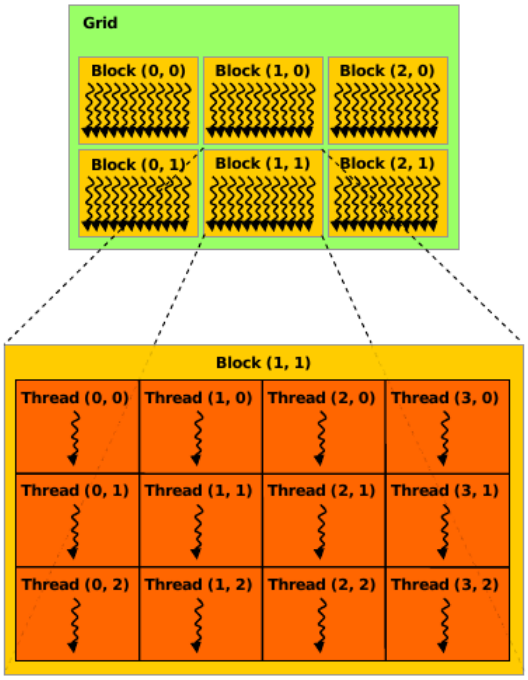

Figura 3.3: Jerarquía de hilos [9]

```
6       if (i < N && j < N)
7         C[i][j] = A[i][j] + B[i][j];
8   }
9
10  int main()
11  {
12      ...
13      // Invocación del kernel
14      dim3 dimBlock(16, 16);       // tamaño arbitrario de bloque
15      dim3 dimGrid((N + dimBlock.x - 1) / dimBlock.x,
16                   (N + dimBlock.y -1) / dimBlock.y);
17      VecAdd<<<dimGrid, dimBlock>>>(A, B, C);
18  }
```

3.3.1.1. Jerarquía de memoria

Cada hilo de CUDA puede acceder a varios espacios de memoria:

- **Local Memory** (Memoria local): Es una región de la memoria normal de la GPU

(*device memory*) privada de cada hilo.

- **Shared Memory** (Memoria compartida): Se trata de una memoria mucho más rápida, pero que hay en mucha menor cantidad. Se comparte entre los hilos de un bloque.
- **Global Memory** (Memoria global): Espacio de memoria normal de la GPU (*device memory*) que comparten todos los hilos.
- **Constant Memory** (Memoria constante): Es de solo lectura y es accesible para todos los hilos.
- **Texture Memory** (Memoria de texturas): Es de solo lectura y es accesible para todos los hilos. Es eficiente cuando hay localidad espacial.

3.3.2. Implementación hardware

La arquitectura de CUDA consiste en una serie de *Streaming Multiprocessors* (SMs) escalables (Figura 3.4) que se corresponden con un bloque en el modelo de programación de CUDA: cuando se ejecuta un programa CUDA, éste invoca a un kernel que distribuye los bloques del *grid* entre los SMs disponibles. Cuando un bloque termina su ejecución, un nuevo bloque ocupará el SM que queda vacío.

Un *Streaming Multiprocessor* se compone de ocho *Scalar Processors* (SPs), dos unidades especiales para números trascendentes, una unidad de instrucciones multihilo y memoria compartida. El SM se encarga de crear, manejar y ejecutar hilos concurrentes por hardware sin introducir ninguna sobrecarga o coste adicional. Además, implementa la sincronización por barrera **__syncthreads()** en una sola instrucción. Todo esto hace posible un paralelismo de grano fino eficiente, permitiendo asignar un hilo a cada elemento de datos, por ejemplo, un píxel en una imagen.

Para controlar cientos de hilos el SM utiliza una arquitectura llamada SIMT (*Single Instruction, Multiple Thread*): se asigna cada hilo a un SP y cada uno se ejecuta de forma independiente con sus propios registros. El SM crea, maneja, planifica y ejecuta los hilos en grupos de 32 llamados *warps*, Cuando le llegan uno, o varios bloques, los divide en *warps* que se planifican por la unidad SIMT. Un *warp* ejecuta una instrucción en común a la vez, así que la mayor eficiencia se alcanza cuando todos los *warps* siguen el mismo camino. Si hay divergencias por una instrucción condicional entre los hilos de un *warp*, se ejecuta en serie cada camino, con la consiguiente pérdida de eficiencia.

Como se puede ver en la Figura 3.5, cada SM tiene cuatro tipos de memoria:

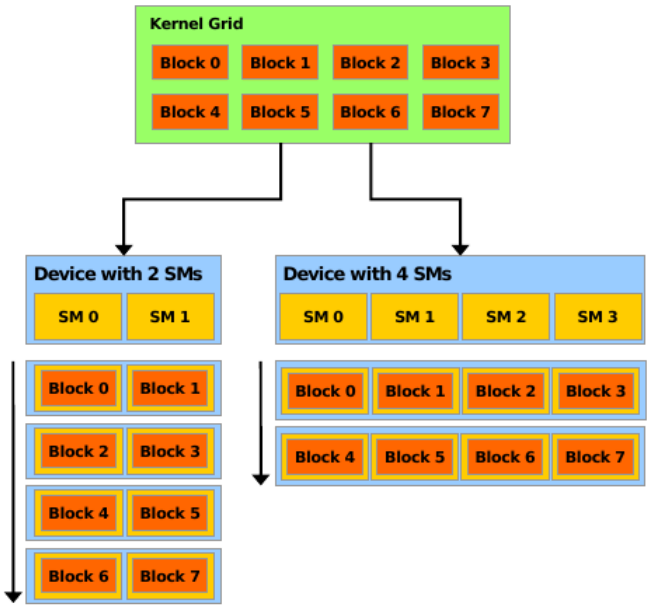

Figura 3.4: Escalabilidad automática [9]

- Un conjunto de **registros** de 32 bits por procesador.
- Una **memoria compartida** por los SPs, es decir, por el bloque de hilos que es donde reside la *shared memory*.
- Una **cache constante** de solo lectura compartida por los SPs que acelera las lecturas de la *constant memory*.
- Una **cache de texturas** de solo lectura compartida por los SPs que acelera las lecturas de la *texture memory*.

Las GPUs compatibles con CUDA se clasifican según su Capacidad Computacional, desde Capacidad 1.0 hasta 1.3, por el momento. Cuanta más Capacidad más potencia y menos restricciones para el programador.

3.3.3. Rendimiento

Para obtener el mayor rendimiento de las aplicaciones CUDA existen una serie de recomendaciones a seguir a la hora de escribir el código. Aquí se describirán brevemente las más importantes (para una descripción detallada de todas las recomendaciones, consultar [8]).

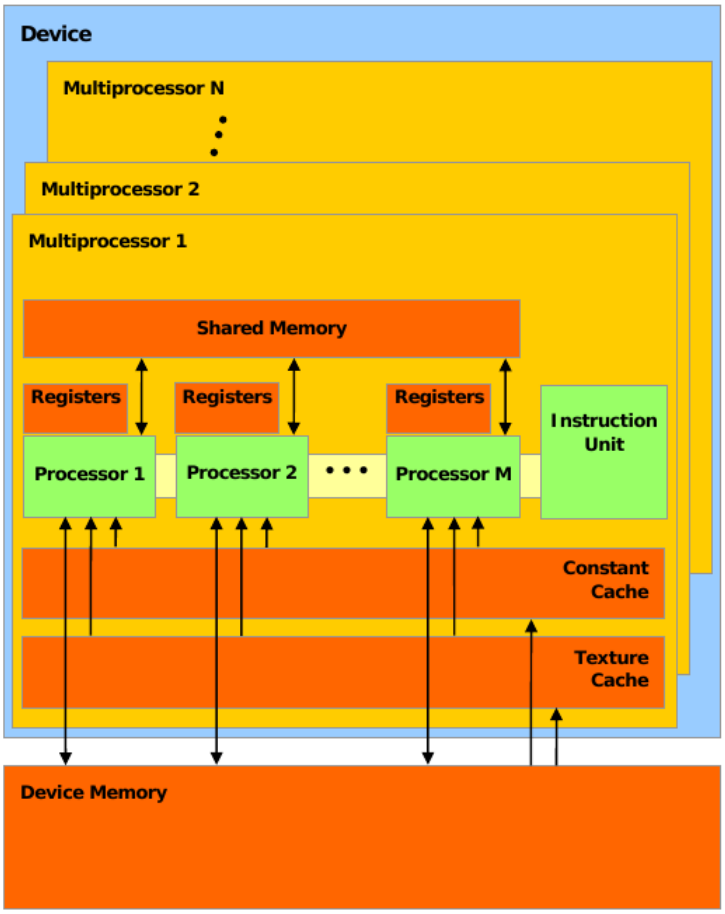

Figura 3.5: Arquitectura hardware [9]

Lo primero que hay que hacer para adaptar nuestro código a CUDA es buscar formas de **convertir el código secuencial en paralelo** para aumentar el posible beneficio de utilizar CUDA (ver Sección 2.1), de otra forma no obtendríamos ningún beneficio o incluso perderíamos rendimiento. El código que no podemos paralelizar debería ejecutarse en la CPU.

El ancho de banda entre la memoria de la GPU y la propia GPU (141GBps en una NVIDIA GeForce GTX 280, por ejemplo) es mucho más grande que entre la RAM del sistema y la GPU (8GBps en el bus PCI Express x16 Gen2), por lo tanto, es muy importante **minimizar la transferencia entre la RAM del sistema y la GPU**. Todas las estructuras de datos intermedias deberían crearse en la memoria de la GPU y es importante agrupar las transferencias pequeñas de memoria en una grande, debido al coste de cada transferencia.

Otra medida importante para mejorar el rendimiento es **utilizar la memoria compartida** para almacenar datos intermedios compartidos entre los hilos de un bloque ya que su latencia es aproximadamente 100 veces menor que la latencia de la memoria global. La memoria compartida está dividida en bancos de memoria que pueden ser accedidos simultáneamente, por lo tanto, es aconsejable **evitar conflitos de bancos**. Para evitar los conflictos hay que tener en cuenta que los bancos están organizados de forma que palabras sucesivas de 32 bits son asignadas a bancos sucesivos.

Sin duda, una de las recomendaciones más importante es **asegurarse de que los accesos a memoria sean *coalesced*** (término que indica que varios accesos se fusionan en un único acceso) siempre que sea posible.

Las lecturas y escrituras de memoria global de hilos de medio *warp* (16 hilos) pueden ser realizadas en una sola transacción (o dos si se trata de palabras de 128 bits) si se cumplen ciertas condiciones. Para entender estos requisitos, la memoria debe verse como una serie de segmentos de 16 y 32 palabras alineados. La Figura 3.6 ayuda a explicar el *coalescing* de medio *warp* de palabras de 32 bits, como los float. Cada fila representa un segmento de 64 bytes (16 floats) y cada fila del mismo color representa un segmento de 128 bytes. En la parte inferior se puede ver medio *warp* de hilos que accede a la memoria global.

Ahora veremos los requisitos para el *coalescing* según la Capacidad Computacional de la GPU:

- En GPUs con Capacidad Computacional 1.0 o 1.1, el hilo k-ésimo de medio *warp* tiene que acceder a la palabra k-ésima en un segmento alineado a 16 veces el tamaño de los elementos accedidos; sin embargo, no todos los hilos tienen que participar.
- Para GPUs con Capacidad Computacional 1.2 y superior, se puede alcanzar el *coales-*

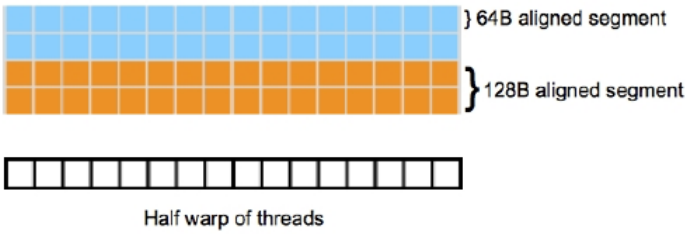

Figura 3.6: Segmentos de memoria global y medio *warp* de hilos [8]

cing con cualquier patrón de accesos que encaje en un segmento de 32 bytes para palabras de 8 bits, 64 bytes para palabras de 16 bits o 128 bytes para palabras de 32 y 64 bits. También se pueden realizar transacciones más pequeñas para no malgastar ancho de banda si sólo se utiliza la parte superior o inferior de los segmentos de 64 y 128 bytes.

Capítulo 4

C++ y STL

En este capítulo hablaremos brevemente sobre el lenguaje de programación C++, sus características más relevantes y las principales ventajas que ofrece frente a otras alternativas, y comentaremos la *Standard Template Library* (STL). Además, describiremos los algoritmos **std::transform** y **std::accumulate** implementados en la STL, que son los utilizados en este proyecto.

4.1. C++

C++ es un lenguaje de programación de de alto nivel, multiplataforma y de uso general, aunque con un cierto sesgo hacia la programación de sistemas que, a grandes rasgos:

- es un C mejorado (de hecho, es en general un superconjunto del lenguaje C),
- soporta abstracción de datos,
- soporta programación orientada a objetos y
- soporta programación genérica. [11]

Por lo tanto, se puede decir que C++ es un lenguaje de programación multiparadigma. Además, es muy versátil: incluye facilidades tanto para programar en alto nivel como en bajo. A pesar de estas características, uno de los objetivos de su diseño es ofrecer un buen rendimiento en las aplicaciones con él programadas.

Comentaremos ahora algunas de sus características diferenciadoras de otros lenguajes de programación.

4.1.1. Operadores

Como cualquier otro lenguaje de programación de alto nivel, C++ incluye un gran conjunto de operadores que permiten realizar operaciones aritméticas, lógicas, comparaciones, etc. Lo interesante es que casi todas esos operadores se pueden sobrecargar para definir el comportamiento deseado por el programador. De esta forma, cuando define sus propios tipos, puede definir además sus operadores y conseguir que los nuevos tipos se usen exactamente igual que los propios del lenguaje.

En este proyecto se utiliza la sobrecarga del operador() para encapsular en un objeto una función de CPU y una referencia a un kernel GPU (detalles en el Capítulo 5).

4.1.2. Plantillas

Las plantillas (*templates*) es la forma que tiene C++ de implementar la programación genérica.

Una plantilla es una clase o función parametrizada escrita sin conocer los argumentos específicos usados para instanciarlas. Después de la instanciación el resultado es un código equivalente al escrito de forma específica para los argumentos pasados [2]. Además, esta genericidad se consigue sin pérdida de rendimiento, ya que esta sustitución se realiza en tiempo de compilación.

En este proyecto se utilizan las plantillas para conseguir independencia en los tipos pasados a las funciones transform y accumulate (detalles en el Capítulo 5).

4.2. STL

La *Standard Template Library* (STL) es una librería para C++ que es una colección de clases y funciones diseñadas para facilitar la programación, al incluir estructuras y algoritmos de uso frecuente. Parte de esta librería se ha incluído en la librería estándar de C++ y es ampliamente utilizada por su carácter genérico y abstracción.

Entre otras cosas, la STL incluye una serie de contenedores genéricos como vectores, listas o conjuntos en los que se pueden introducir tipos de datos de C++ y cualquier tipo definido por el programador. Además, para esos contenedores se incluyen también: iteradores para recorrerlos, algoritmos para aplicarles diversas operaciones como búsquedas u ordenaciones y functores (clases que sobrecargan el operador()) para, por ejemplo, especificar el orden en el algoritmo de ordenación que son independientes del tipo de contenedor sobre el que se aplican. Toda esta independencia se consigue a través del uso de plantillas

que, como ya se ha dicho, no conllevan pérdida de rendimiento.

4.2.1. Transform

El algoritmo **std::transform** lleva a cabo una operación específica sobre cada elemento de un contenedor. Por ejemplo, podría hacer el cuadrado de cada elemento del contenedor.

Este es su prototipo:

```
template <class InputIterator, class OutputIterator, class
   UnaryFunction>
OutputIterator transform(InputIterator first, InputIterator last,
                         OutputIterator result, UnaryFunction op);
```

Recibe dos iteradores de entrada sobre un contenedor que definen los límites entre los cuales se aplica la función, uno de salida (que puede ser el mismo que el primero de entrada para modificar "al vuelo" el contenedor) que especifica en qué contenedor se colocará el resultado y un functor unario (puede ser una función tradicional o un functor propio) que define la operación que se va a aplicar. Los tipos de los iteradores y el functor se infieren a partir de los argumentos pasados a la función debido a que están definidos como plantillas.

Existe también otro prototipo que aplica una función binaria, pero no se utiliza en este proyecto.

4.2.2. Accumulate

El algoritmo **std::accumulate** calcula y devuelve la suma (o alguna otra operación binaria) de *init* y todos los elementos en el rango [first,last).

Este es el prototipo de la versión que suma:

```
template <class InputIterator, class T>
T accumulate(InputIterator first, InputIterator last, T init);
```

Y este es el prototipo de la versión en la que se le pasa otra operación binaria:

```
template <class InputIterator, class T, class BinaryFunction>
T accumulate(InputIterator first, InputIterator last, T init,
             BinaryFunction binary_op);
```

Capítulo 5

Implementación

En este capítulo hablaremos sobre la implementación realizada de los algoritmos **std::transform** y **std::accumulate** para aprovechar características multiGPU y multiCPU. El objetivo, como ya se ha comentado en capítulos anteriores, es aprovechar todos los recursos computacionales de un sistema heterogéneo con múltiples nucleos CPU y múltiples GPUs con capacidades GPGPU.

En las siguientes secciones describiremos los distintos aspectos de nuestra implementación, haciendo hincapié en las soluciones propuestas a los distintos problemas que se nos han planteado.

5.1. Interfaz

La idea de nuestra implementación de las funciones de la STL es que sea intercambiable con la implementación original. Para ello, se han definido utilizando plantillas para abstraer los tipos y la misma interfaz que las funciones originales excepto el tipo de las operaciones, que se ha cambiado por nuestros tipos.

5.1.1. Definición de operaciones

Para definir las operaciones que tendrán que realizar los algoritmos **transform** y **accumulate** se han creado clases abstractas que definen la interfaz de cada operación. Estas clases incluyen una referencia al kernel de GPU (su nombre) que se asigna en el constructor y además, obligan a implementar el operador() a las clases derivadas, que es donde irá la implementación en CPU.

La clase que define la interfaz de accumulate obliga también a definir una identidad para

realizar la reducción en CPU.

Clase abstracta de operaciones de transform:

```
template <typename T> class Transform_op {
protected:
    const char *_name;
public:
    Transform_op(const char* name):_name(name){}
    virtual int operator()(T const& n) const = 0;
    const char* getName()
    {
        return _name;
    }
};
```

Clase abstracta de operaciones de accumulate:

```
template <typename T> class Accumulate_op {
protected:
    const char *_name;
public:
    Accumulate_op(const char* name):_name(name){}
    virtual T operator()(T const& acc, T const& n) const = 0;
    virtual T identity() const = 0;
    const char* getName()
    {
        return _name;
    }
};
```

Dado que nuestra implementación está escrita en C++, es necesario utilizar el API del *driver* de CUDA y compilar por separado los kernels con **nvcc**. Esto, además de añadir complejidad a la hora de tratar con la GPU ocasiona (como ha sido comentado en la Sección 3.3) un problema al tratar con las plantillas en CUDA.

Desde la versión 3.0, lanzada en marzo de 2010, los kernels CUDA tienen soporte para plantillas, de modo que utilizando *C for CUDA*, el kernel puede llamarse de esta forma:

```
kernel<Plantilla><<<dimGrid,dimBlock>>>(parámetros);
```

Sin embargo, al utilizar el API del *driver* no es posible pasar el parámetro de plantilla al kernel, ya que a la función que llama al kernel solo se le pasa el nombre:

```
1  cudaLaunch("nombre_de_kernel");
```

Para solucionar este problema, los nombres de los kernels seran de la forma **nombre_*tipo*** donde *tipo* es el valor devuelto por la función **typeid()** para el tipo aceptado por las entradas del kernel. Así, cuando el nombre del kernel y el parámetro de plantilla lleguen a la función que se encarga de llamar el kernel, se concatenarán y se ejecutará el kernel adecuado.

La firma de las funciones para transform es la siguiente:

```
1  extern "C" __global__ void operacion_tipo(TIPO * const vector,
       const int size)
```

Para accumulate:

```
1  extern "C" __global__ void operacion_tipo(const TIPO * const d_in,
       TIPO * const res, const int size)
```

5.1.2. Interfaz de las funciones

Transform:

```
1  template < typename InputIterator, typename OutputIterator,
       typename T >
2  OutputIterator transform ( InputIterator first1, InputIterator
       last1, OutputIterator result, Transform_op<T> & op );
```

Accumulate con la operación por defecto:

```
1  template <class InputIterator, class T>
2      T accumulate(InputIterator first, InputIterator last, T init)
```

Accumulate:

```
1  template <class InputIterator, class T>
2      T accumulate(InputIterator first, InputIterator last, T init,
3                   Accumulate_op<T> & binary_op);
```

Como nuestros tipos definen la operación(), pueden ser utilizados para llamar a las funciones originales (Sección 4.2).

5.2. Manejo de los hilos

Este proyecto requiere el uso de varios hilos de ejecución en CPU por dos razones: para explotar la presencia de CPUs con varios núcleos y para que las diferentes GPUs trabajen en paralelo; así, cada implementación concreta tiene su propio hilo. Para el manejo de los hilos se utilizan las funcionalidades básicas que ofrece la clase Thread de la librería Boost (Sección 2.3) y el patrón de diseño "Adquirir Recursos es Inicializar", en adelante, RAII (*Resource Acquisition Is Initialization*).

5.2.1. RAII

RAII es un patrón de diseño utilizado en varios lenguajes de programación orientados a objetos como C++ o ADA para tratar el problema de la liberación de recursos.

La idea principal de RAII es simple: las clases deben manejar los recursos. Cuando se instancia una clase, el constructor inicializa los recursos necesarios (de ahí el nombre del patrón) y cuando el objeto se destruye, el destructor de la clase se encarga de liberar los recursos. Así, el tiempo de vida del objeto coincide con el tiempo de vida del recurso.

El patrón RAII ofrece las siguientes ventajas:

- Evita que el programador tenga que preocuparse de liberar los recursos de forma manual. Por ejemplo, en el caso del manejo de la memoria dinámica, el programador tiene que preocuparse de liberarla cuando ya no es necesaria (realizando un **delete** después de cada **new** en C++); esto puede ocasionar fugas de memoria (*memory leaks*) ya que es común olvidarse de su liberación. Con RAII cuando el objeto sale del espacio de nombres el destructor se ejecuta y reclama el recurso automáticamente.

- En el caso de C++, al lanzarse una excepción, se garantiza la ejecución es el de los destructores de los objetos en la pila, por lo que gracias al patrón RAII, se reclaman los recursos asociados a esos objetos. Esto evita escribir código para reclamar los recursos al tratar la excepción, que estaría disperso y duplicado.

5.2.2. Implementación de los hilos

Se ha decidido utilizar el patrón RAII para el manejo de los diferentes hilos. Cada hilo se representa con un objeto que está formado por un constructor que crea un hilo nuevo, una función que incluye el código a ejecutar en el nuevo hilo y un destructor que espera a que el hilo termine la ejecución.

Un problema que surge con el uso de los hilos es qué hacer cuando se produce un error en un hilo que no es el principal. En caso de error en uno de los hilos no se aplica la operación su parte del dato de entrada, por lo tanto el resultado que devuelve la operación es incorrecto y el usuario debería ser informado. El problema reside en que el hilo que detecta el error lanza una excepción pero ésta no puede propagarse directamente al hilo principal porque son hilos diferentes. La solución adoptada fue incluir en la clase que maneja el thread una variable del tipo de la excepción que puede lanzar ese hilo. Así, cuando el hilo secundario recoge la excepción, se copia a esta variable y en el destructor del objeto (cuyo código se ejecuta en el hilo principal) se comprueba si se ha producido una excepción y, en caso afirmativo, se relanza.

5.3. Transform

El algoritmo transform simplemente aplica una operación a cada elemento del vector de entrada y escribe el resultado en otro vector o en el propio vector de entrada. Al tratarse de una operación unaria no hay dependencias entre los datos de entrada.

5.3.1. Reparto de trabajo

El vector de entrada se reparte en dos partes, una para las CPUs y otra para las GPUs. Mediante un factor pasado como parámetro a la implementación , se ajusta el tamaño trabajado por cada una de las partes y, finalmente, la fracción asignada a cada parte se divide entre su número de unidades de procesamiento (núcleos en el caso de las CPUs y tarjetas gráficas en el caso de las GPUs). En el Capítulo 6.2 se comentarán los resultados obtenidos para varias pruebas realizadas con diferentes factores.

5.3.2. Implementación CPU

La implementación CPU es trivial: un bucle que recorre el vector de entrada y le va aplicando la operación a cada elemento escribiendo en el vector de salida.

5.3.3. Implementación GPU

La implementación GPU también es sencilla: se pasa el vector de entrada a la memoria global y en el kernel se le aplica la operación a cada elemento. Un ejemplo con la operación raíz cuadrada:

```
1 extern "C" __global__ void sqrt_f(float * const vector, const int
     size)
2 {
3     int i = (blockIdx.y * gridDim.x * blockDim.x) + blockIdx.x *
         blockDim.x + threadIdx.x;
4     if (i < size)
5         vector[i] = sqrt(vector[i]);
6 }
```

De esta forma, cada hilo de CUDA realizará la operación a su elemento correspondiente en paralelo con el resto. No es conveniente utilizar la memoria compartida ya que cada hilo lee y escribe una sola vez a memoria global y trayéndolo a memoria compartida se haría el mismo número de accesos a memoria global mas otros dos accesos a memoria compartida.

5.4. Accumulate

El algoritmo accumulate (también conocido como reduction) calcula la suma (o alguna otra operación binaria) de todos los elementos del vector de entrada.

5.4.1. Reparto de trabajo

El vector a reducir se reparte a partes iguales entre las GPUs y los resultados parciales obtenidos se operan en un hilo de CPU. En el Capítulo 6.2 se justificará el porqué de no utilizar varios hilos de CPU.

5.4.2. Implementación CPU

La implementación CPU es trivial: se inicializa un acumulador con la identidad de la operación a realizar y se recorre el vector de entrada con un bucle acumulando el resultado de operar el acumulador con el elemento correspondiente en cada iteración del bucle y, como resultado final, se devuelve el acumulador.

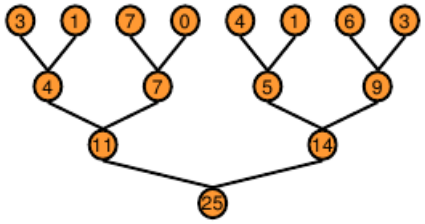

Figura 5.1: Árbol de reducción

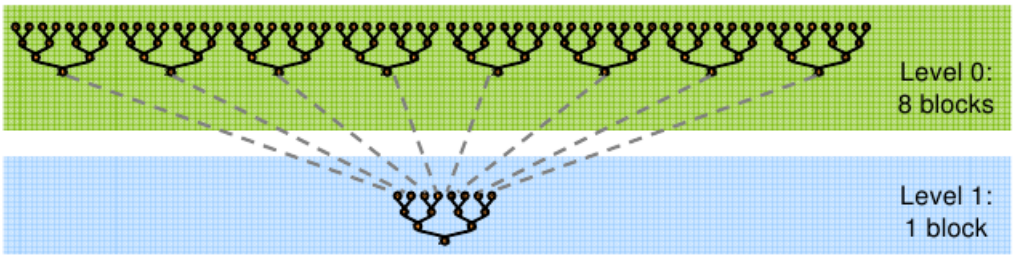

Figura 5.2: Aplicación recursiva del kernel

5.4.3. Implementación GPU

En la implementación GPU no tiene sentido utilizar algo parecido a la implementación GPU de transform porque no se aprovecharía el paralelismo, cada hilo necesita datos que corresponderían a otros hilos.

En esta implementación se utilizará un enfoque paralelizable en forma de árbol (Figura 5.1). Cada bloque reducirá un trozo del vector de entrada y obtendrá un resultado parcial, después se aplicará el mismo kernel de forma recursiva hasta que solo quede el resultado final (Figura 5.2)

Para la implementación en GPU se han probado varios kernels para conseguir más rendimiento. En este caso sí se utiliza la memoria compartida, ya que habrá que iterar varias veces por bloque para ir descendiendo en el árbol de reducción.

La primera versión del código puede verse ejemplificada en la Figura 5.3. Es una traducción directa del árbol de reducción a código:

```
1 __global__ void reduce0(int *g_idata, int *g_odata) {
2     extern __shared__ int sdata[];
3 
4     // cada hilo carga un elemento de la memoria global a la
        compartida
```

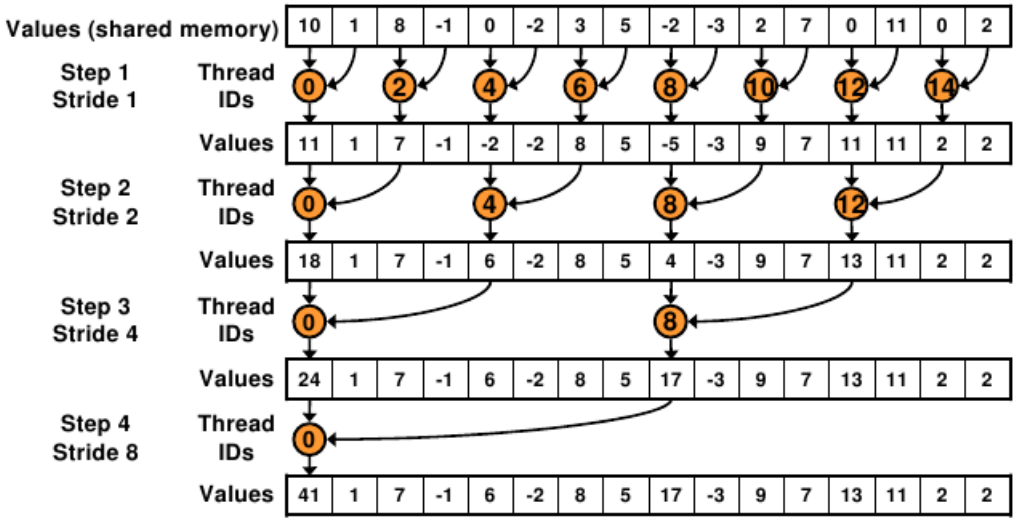

Figura 5.3: Direccionamiento intercalado 1

```
5       unsigned int tid = threadIdx.x;
6       unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
7       sdata[tid] = g_idata[i];
8       __syncthreads();
9
10      // reducir en memoria compartida
11      for(unsigned int s=1; s < blockDim.x; s *= 2) {
12          if (tid % (2*s) == 0) {
13              sdata[tid] += sdata[tid + s];
14          }
15          __syncthreads();
16      }
17
18      // escribir resultado en memoria global
19      if (tid == 0)
20          g_odata[blockIdx.x] = sdata[0];
21  }
```

Este código tiene dos problemas: la operación de módulo es muy lenta en la GPU y, el utilizar solo los hilos con *threadID* par, es muy ineficiente porque los *warps* son muy divergentes (ver Sección 3.3.2).

Para evitar esa divergencia y la utilización de la operación de módulo se puede cambiar

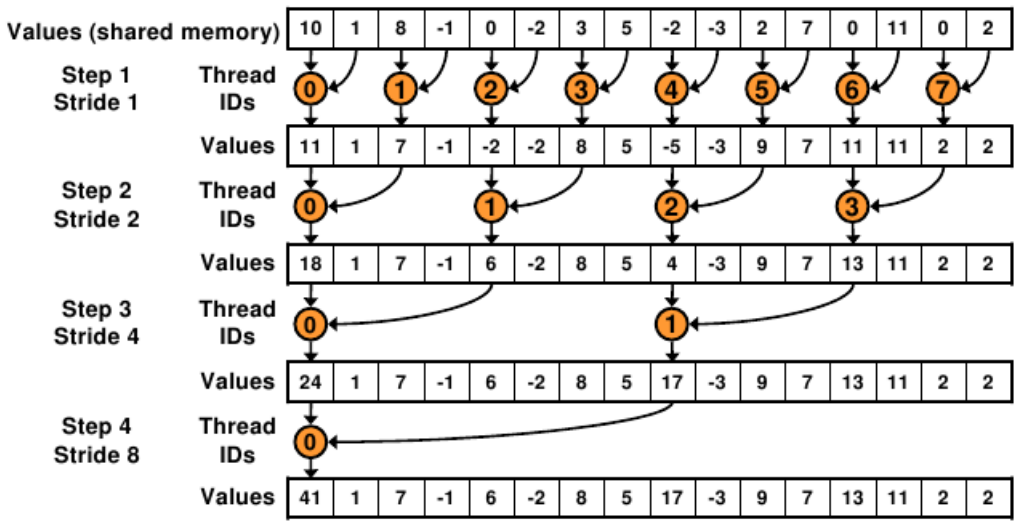

Figura 5.4: Direccionamiento intercalado 2

el bucle **for** por este otro:

```
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();
}
```

Estas modificaciones están ejemplificadas en la Figura 5.4.

El problema de este último código (que también estaba presente en la primera versión ya comentada) es que genera conflictos de banco al no acceder a los elementos secuencialmente en la memoria compartida. Este problema puede evitarse con otro cambio en el bucle, concretamente:

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }

```

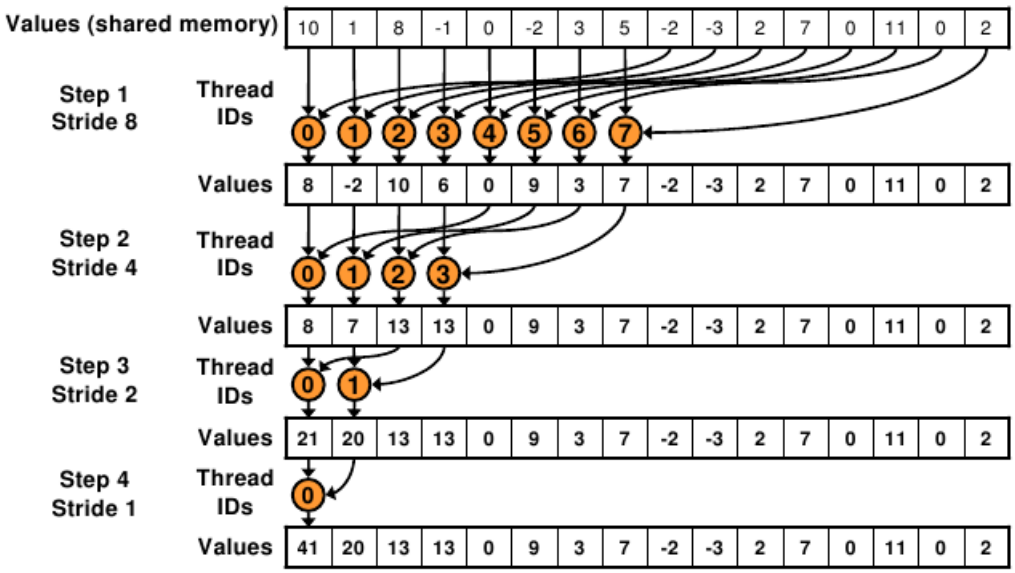

Figura 5.5: Direccionamiento secuencial

```
6       __syncthreads();
7   }
```

Como puede verse en la Figura 5.5, con el nuevo patrón de acceso a la memoria compartida desaparecen los conflictos de banco al acceder secuencialmente todos los hilos de cada *warp*.

Aquí está la versión final que se ha implementado:

```
1   extern "C" __global__ void reduction_add_j(const unsigned int *
        const d_in,
2    unsigned int * const res, const int size)
3   {
4       extern __shared__ int s_data[];
5
6       int idxOffset = blockDim.x * blockIdx.x;
7       int idx = idxOffset + threadIdx.x;
8       int tx = threadIdx.x;
9
10      s_data[tx] = (idx < size)? d_in[idx]: 0;
11
12      __syncthreads();
13
```

```
14      int stride;
15
16      for (stride=blockDim.x/2; stride > 0; stride>>=1) {
17          if (tx < stride) {
18              s_data[tx] += s_data[tx + stride];
19          }
20          __syncthreads();
21      }
22      if (tx == 0)
23          res[blockIdx.x] = s_data[0];
24  }
```

Capítulo 6

Resultados Experimentales

En este capítulo se mostrarán los resultados de rendimiento obtenidos con nuestra implementación original de los algoritmos **transform** y **accumulate** y se compararán con los obtenidos por la implementación de la STL.

6.1. Configuración de pruebas

El equipo de pruebas está formado por dos GPUs NVIDIA GeForce 9800 GT, una CPU Intel Core 2 Quad a 2.66Ghz y 3GB de RAM. En lo que respecta al apartado software, el sistema operativo del equipo es un Debian GNU/Linux con kernel 2.6.32. Se ha usado la versión 4.3 del compilador de C/C++ de gnu (gcc y g++) junto con la versión 1.42 de la librería Boost. La versión de CUDA instalada en la máquina es la 3.0.

Debido a la necesidad de tener un hilo para gestionar cada GPU, el total de hilos utilizados para el cálculo en CPU es 2.

Para las pruebas de **transform** se utilizará un vector de 450MB de números en coma flotante de precisión simple y la operación será una raíz cuadrada.

Para las pruebas de **accumulate** se utilizará un vector de 450MB de enteros y la operación será una suma.

Todos los resultados se han obtenido mediante la realización de la media de 10 ejecuciones de las funciones.

6.2. Resultados de las pruebas

Describiremos ahora las columnas de las tablas de resultados:

- **Factor**: factor de uso de GPU.
- **Cálc. GPU** tiempo utilizado por cada GPU en realizar el cálculo que le corresponde.
- **Transfer. GPU**: tiempo utilizado por cada GPU en realizar la reserva y transferencias de memoria.
- **Total GPU**: tiempo total utilizado por cada GPU.
- **Total CPU**: tiempo total utilizado por cada CPU.
- **TOTAL**: tiempo total utilizado por nuestra implementación para realizar el algoritmo (incluye el tiempo de todas las CPUs y GPUs).

En el Cuadro 6.1 están los resultados de aplicar la función **transform** con diferentes factores de uso de GPU. Cada hilo trabaja con un elemento del vector de entrada, por lo tanto, el número de hilos totales es igual al número de elementos: 117964800. El número de hilos por bloque es 512 y el número de bloques es 230400.

Se puede observar en primer lugar que el tiempo de transferencia supera con mucho el tiempo de cálculo como indicábamos en la Sección 3.3.3. Por otra parte, se puede ver que el tiempo utilizado por la CPU es similar al que emplea la GPU contando las transferencias de memoria, con una ventaja hacia la GPU según aumenta el tamaño del vector.

Dadas las pocas diferencias entre la CPU y la GPU no tiene sentido que el mayor rendimiento aparezca cuando se usa solo CPUs o solo GPUs porque implica que hay menos núcleos de procesamiento en ejecución. Como conclusión, el mayor rendimiento se obtiene con factores medios y existe poca diferencia entre ellos.

En el Cuadro 6.2 están los resultados de la aplicación de la función **accumulate**.

Se reafirma la diferencia de tiempos entre las transferencias a la GPU y sus cálculos y se observa que el tiempo de cálculo en la CPU es prácticamente despreciable, ya que en este caso solo se encarga de realizar la operación entre 2 elementos, el número de GPUs del sistema.

Finalmente, en la Figura 6.1 se puede observar el aumento de velocidad de nuestras implementaciones con respecto a las de la STL.

Cuadro 6.1: Resultados de **transform** (tiempos en ms)

Factor	Cálc. GPU	Transfer. GPU	Total GPU	Total CPU	TOTAL
0	–	–	–	843, 8	844, 3
0,2	2, 54	192, 67	215, 84	749, 33	762, 92
0,4	5, 11	362, 18	405, 06	636, 26	638, 84
0,6	10,39	549, 51	565, 17	475, 15	640, 7
0,8	11, 77	565, 23	580, 5	238, 19	658, 05
1	13, 18	647, 3	659, 14	–	738, 96
					TOTAL
Impl. STL					3758,03

Cuadro 6.2: Resultados de **accumulate** (tiempos en ms)

Cálc. por GPU	Transf. por GPU	Total por GPU	Total por CPU	TOTAL
47, 28	321, 37	435, 15	0,002	488, 74
				TOTAL
Impl. STL				1496,96

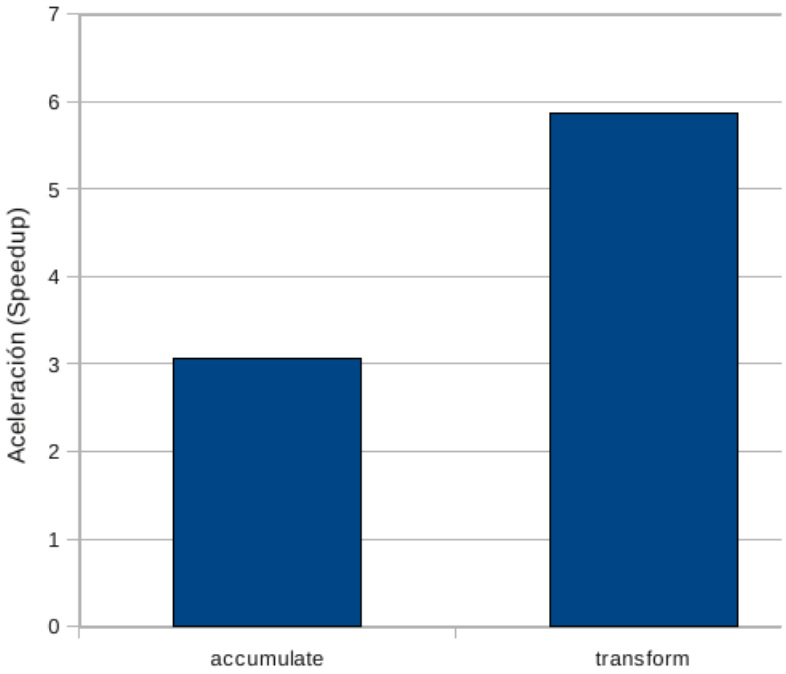

Figura 6.1: Aceleración de nuestra implementación con respecto a la de la STL

Capítulo 7

Conclusiones

En este proyecto hemos implementado los algoritmos **transform** y **accumulate** de la librería STL utilizando las capacidades paralelas de los sistemas multiCPU y multiGPU obteniendo mejoras en el rendimiento con respecto a la implementación de la librería estándar de C++.

Hemos aprendido y utilizado el lenguaje de programación C++, uso y detalles de implementación de la librería STL, uso de la librería POSIX Threads a través de la abstracción Boost.Thread para manejar múltiples hilos y, finalmente, programación en CUDA para GPUs NVIDIA y aprovechamiento de los diferentes sistemas de memoria y del paralelismo de la GPU.

Como trabajo futuro en este ámbito, se podrían implementar más algoritmos paralelizables de la sección *algorithm* y *numeric* de la STL con técnicas parecidas a las aquí presentadas y se podría mejorar el aprovechamiento de los recursos de la CPU, especialmente para el algoritmo **accumulate**.

Bibliografía

[1] Boost.thread.
`http://www.boost.org/doc/html/thread.html`.

[2] C++.
`http://en.wikipedia.org/wiki/C++#Templates`.

[3] Classic risc pipeline.
`http://en.wikipedia.org/wiki/Classic_RISC_pipeline`.

[4] Flynn's taxonomy.
`http://en.wikipedia.org/wiki/Flynn's_taxonomy`.

[5] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Professional, 1997.

[6] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Pres, 2007.

[7] Message passing interface forum.
`http://www.mpi-forum.org`.

[8] NVIDIA. *CUDA Best Practices Guide*, 2009.

[9] NVIDIA. *CUDA Programming Guide*, 2009.

[10] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, 2007.

[11] Bjarne Stoustrup. *The C++ Programming Language*. 0-201-70073-5. Reading, Massachusetts : Addison-Wesley, 2000.

[12] G.S. Almasi y A. Gottlieb. *Highly parallel computing*. ISSN:0018-8670. Benjamin-Cummings publishers, 1989.