PROYECTO FIN DE CARRERA

Arquitectura de Computadores y Sistemas Operativos

Memoria

Desarrollo CUDA en Java y Python

José Alejandro Pérez Sánchez

Consultor: Francesc Guim Bernat

Índice

Introducción

Durante años, el aumento del rendimiento de los procesadores se ha basado en el incremento de las frecuencias de reloj de los mismos, así como en la adición de una mayor cantidad de transistores, mientras se reducía el tamaño de los mismos.

Durante este tiempo, fue de aplicación la llamada ley de Moore, que auguraba la duplicación del número de transistores en un circuito integrado. Sin embargo, debido a limitaciones físicas, como el calor generado o el tamaño del átomo, esto ha dejado de ser así.

Los fabricantes de procesadores han optado, por tanto, por incidir en otras estrategias, además del número de transistores, para incrementar el rendimiento de sus procesadores. Bien soportar un mayor número de hilos, *threads*, de ejecución en cada procesador, o fabricar procesadores con un número distinto de núcleos, *cores*, en su interior. A menudo se trata de una combinación de ambas estrategias.

Por tanto, al disponer de distintos hilos de ejecución, el diseño de programas que puedan aprovechar éstos para ejecutar tareas en paralelo y así exprimir el rendimiento de los procesadores ha sido un importante campo de investigación.

Al mismo tiempo, otra aproximación a la implementación de procesadores paralelos se ha dado por parte de fabricantes de procesadores especializados en la fabricación de procesadores graficos, GPU, *Graphic Processing Units*.

Estas GPU, al contrario que los procesadores mencionados inicialmente, no son procesadores de propósito general, con sus propios hilos de ejecución y datos, sino que deberían verse como procesadores simétricos, donde el hilo de ejecución es el mismo, en todos sus núcleos, y lo que varía es el conjunto de datos (posiciones de memoria), sobre el que se realiza esa ejecución.

Por tanto, las GPU son adecuadas para soportar algoritmos con un alto grado de paralelismo, sobre estructuras de datos no excesivamente complejas, como vectores o matrices de números, y con alta intensidad aritmética.

Además, las GPU presentan ciertas ventajas sobre los procesadores multipropósito con múltiples hilos y núcleos de ejecución, que en general poseen un mayor número de núcleos, (por ejemplo, un procesador Intel i7-3770T, de los de más alta gama con 4 núcleos y 8 hilos por núcleo, cuesta alrededor de 280$, mientras que una GPU de gama media-baja, como NVIDIA GT 630, [Nvd10], que será la usada durante el desarrollo del Proyecto Fin de Carrera, tiene 192 núcleos por menos de 60€).

En el caso concreto de las GPU producidas por la compañía NVIDIA, esta compañía ha desarrollado una plataforma, junto con su modelo de programación para GPUs, llamada CUDA, [Nvd13b], que incluye APIs así como herramientas de programación para esta plataforma en lenguajes como C, C++ o Fortran.

CUDA son las iniciales de *Compute Unified Devide Architecture*. Se trata de una arquitectura que permite el uso de las GPU, comúnmente llamadas *tarjetas gráficas*, para cálculo

matemático, sea para el manejo de gráficos (uso típico de una GPU), o bien para cálculo científico, financiero, etc.. En suma, nos ofrece un *mini super computador simétrico*. Donde los núcleos o *cores* harían el papel de procesadores que ejecutan el mismo conjunto de instrucciones sobre distintos grupos de datos, distribuidos en *blocks* y *threads*.

Así pues, el proyecto se centrará en el uso de GPUs con soporte CUDA como procesadores simétricos, para aprovecharlas en algoritmos de cálculo intensivo, como pueden ser operaciones con vectores o matrices, algoritmos de ordenación y, finalmente, ver cómo se pueden implementar ciertos algoritmos financieros sobre esas arquitecturas.

Sin embargo, al disponer ya de muchos recursos publicados en internet acerca del desarrollo de algoritmos CUDA, el proyecto tendrá un enfoque algo distinto. Y será el analizar el uso de otros lenguajes de programación en el desarrollo de programas que aprovechen la arquitectura CUDA.

CUDA presenta, además, la ventaja de que se puede emplear para GPUs de gama media-baja, pero también, la misma API (según versiones), en procesadores de gama alta, como es la familia de procesadores TESLA de NVIDIA [Nvd13c], como el Tesla K20, con 2496 núcleos, y con una capacidad de proceso reservada, hasta no hace mucho, a máquinas de varios millones de dólares de coste, frente a los pocos miles que cuesta uno de estas *supercomputing GPUs*.

Pero para acercar esta capacidad de cómputo al software más habitual, no de sistemas, el relacionado con desarrollo de software de negocios, financiero o científico, se mostrará cómo se pueden exprimir estas GPU utilizando lenguajes de programación no habituales en interactuar con GPUs, pero sí en el desarrollo de software de negocio, como pueden ser Java y Python.

Motivación del proyecto

La arquitectura GPU CUDA, de NVIDIA, es hoy día una de las más populares en su ámbito, con cientos de miles de GPUs vendidas que la soportan. Además, como se ha comentado, dispone de librerías y herramientas de programación proporcionadas por la compañía para exprimir su potencial, no sólo en el cometido de procesador gráfico, sino para aprovechar su arquitectura de procesador simétrico con memoria compartida para cualquier tipo de cálculo.

Estas librerías proporcionadas por NVIDIA están disponibles, por defecto, para los lenguajes de programación más habituales en desarrollo de sistemas y videojuegos, C y C++, con una gran interacción con OpenGL y Direct3D [Kdh12].

Sin embargo, en la actualidad, una gran cantidad de desarrollo de software se lleva a cabo en otros lenguajes. Fuera del ámbito de los sistemas operativos o desarrollo de sistemas, lenguajes como Java o C#, y sus plataformas, la JVM y .NET dominan sobre los lenguajes C y C++, y vemos como las nuevas grandes aplicaciones corporativas se implementan en Java donde se aísla al desarrollador de detalles técnicos como el manejo de memoria o los problemas de programación multiplataforma.

Entre los desarrollos que hoy día se implementan en lenguajes como Java o C#, no obstante, hay muchas que se pueden beneficiar, para un incremento en su rendimiento, de una arquitectura escalar como CUDA, en el cómputo masivo, tanto en ámbitos financieros como científicos, pero aprovechando las características y librerías de un lenguaje como Java.

En este contexto se centra la **motivación del proyecto**. Se trata de acercar el cómputo paralelo masivo al desarrollo de software de negocios, incluyendo en este ámbito el software empresarial, financiero y, por supuesto, también científico.

Cabe reseñar que el proyecto en sí no trata sobre la programación básica o estándar en CUDA con el CUDA toolkit de NVIDIA en lenguajes C o C++, sino en cómo se puede desarrollar en esta arquitectura con otros lenguajes como alternativa a éstos, con mayores facilidades como gestión automática de memoria, *garbage collector*, o con lenguajes dinámicos, los ya mencionados Java y Python.

Descripción del proyecto

Para mostrar el uso de CUDA en el desarrollo de software *no sistemas*, y empleando lenguajes de programación más habituales en el software de negocios, se implementarán algunos algoritmos habituales en software financiero y cálculos matemáticos comunes en lenguajes como Java, pero aprovechando la potencia que nos proporciona la arquitectura CUDA.

Inicialmente haré un repaso a los conceptos de desarrollo CUDA para ello utilizaré la bibliografía recomendada así como los materiales disponibles en el Kit de desarrollo CUDA de NVidia, [Nvd13b] y los cursos online en Coursera [Cour12] y en Audacity [Auda12] sobre programación paralela en CUDA en C y C++. Sobre todo para entender bien la arquitectura y los conceptos básicos, y poder comparar mejor con los desarrollos en Java y Python. Esta preparación previa se ha realizado en parte fuera de la planificación del proyecto y no se incluye en ésta.

Se realizará un análisis y descripción del modo de uso de las librerías CUDA en Java, la la implementación de los algoritmos, y se hará un análisis de *speedup, accesos a memoria, hits cache, etc..*, obtenidos al paralelizar el algoritmo.

Esta misma aproximación se repetirá utilizando el lenguaje de programación Python. Éste es un lenguaje también muy extendido, particularmente en ciertos ámbitos científicos, estadísticos, etc.., disponiendo de interesantes librerías numéricas como numpy, centradas en la aritmética de matrices, lo que es, en principio, muy tentador para aplicar una arquitectura como CUDA.

Presenta, además, una particularidad frente a Java, y es que se trata de un lenguaje dinámico, frente a Java, que es un lenguaje orientado a objeto con tipado estático. Se analizará si esto supone diferencias significativas en la forma de desarrollo y en las métricas obtenidas en los mismos algoritmos.

Los algoritmos seleccionados, como se comentó, serán de ordenación, reducción de vectores, multiplicación de matrices, y en el campo financiero, cálculo de precios de opciones, binomial y Black-Scholes, una implementación del método Monte Carlo y un ejemplo de generador de secuencias pseudo-aleatorias.

Las librerías utilizadas para interactuar con CUDA serán JCuda, [Jcu13a], y RootBeer [Roo13b] para Java y PyCuda para Python, [Pyc13a] por ser proyectos con varios años de desarrollo y que se encuentran en un estado suficientemente maduro para ser utilizados en desarrollos reales, no *beta*. Además presentan la ventaja inestimable de ser software libre, con acceso a código fuente y sin restricciones respecto a la licencia.

Sin embargo podemos encontrar otras librerías similares, en este caso comerciales, para interactuar con éstos y otros lenguajes, como Kappa, [Psil13] o, especialmente, para FORTRAN, [Pgic13].

Actividades

La estructura del proyecto será:

- **Análisis previo**

  Análisis de las ideas iniciales sobre el ámbito de desarrollo del proyecto.

- **Determinación de objetivos y planificación**

Determinación del alcance del proyecto, análisis de viabilidad y planificación de actividades. Incluye la elaboración de la PEC1.

- **Análisis de algoritmos**

En esta fase se examinarán detalladamente los algoritmos a implementar en Java y Python, haciendo un análisis exhaustivo de su implementación en C o C++ para arquitectura CUDA. Se examinarán las mejores estrategias de implementación, distintos mapeos de memoria, etc..

Los algoritmos a analizar serán: ordenación quicksort, reducción de vectores, multiplicación de matrices (con optimización para matrices dispersas), cálculo de precio de opciones binomial y Black-Scholes, método Monte-Carlo y generación de números cuasi-aleatoria.

- **JCuda**

En esta fase se realizará la puesta a punto del entorno de desarrollo para utilizar JCuda y se implementarán los algoritmos analizados en lenguaje Java utilizando CUDA. Se analizarán las particularidades de programación requeridas y se hará un análisis del rendimiento del código obtenido, haciendo hincapié en la mejora de rendimiento obtenida al utilizar la arquitectura CUDA frente al uso únicamente del procesador normal del ordenador.

El final de esta fase incluirá el desarrollo de la PEC2, pues al final de esta fase se podrán obtener las primeras conclusiones sobre el aprovechamiento de la arquitectura CUDA utilizando un lenguaje ampliamente utilizado como Java, además de que se incluirán todos los análisis iniciales que serán utilizados en la fase siguiente.

- **PyCuda**

Tras el análisis de JCuda, se realizará una implementación de los mismos algoritmos en Python, utilizando para integrarlo con CUDA la librería PyCuda. Se pondrá a punto un entorno de desarrollo al efecto, y se analizarán las diferencias específicas que puede suponer el uso de un lenguaje de tipado dinámico como Python frente a otros fuertemente tipados como Java, o como lo son C y C++ en las librerías originales CUDA.

Se repetirán los análisis de métricas efectuados en el caso de JCuda para analizar si hay diferencias significativas según las librerías y lenguajes utilizados.

- **Elaboración Memoria Final**

Por último, a partir de las PEC1 y PEC2, y agrupando la experiencia global del proyecto, se redactará la memoria final, incluyendo la experiencia en la programación y prueba de los algoritmos, en la integración de las librerías JCuda y PyCuda con la GPU.

Se analizarán más en detalle los resultados de las métricas y los beneficios que pueden suponer el contar con la potencia de las GPUs actuales en el desarrollo de software financiero y científico.

Objetivos del Proyecto

Análisis de herramientas de desarrollo CUDA en lenguaje Java y en lenguaje Python.

Desarrollo de algoritmos CUDA en las soluciones encontradas.

Evaluación de cada una de las herramientas:

- Instalación
- Proceso de desarrollo de algoritmos
- Rendimiento obtenidos
- Conclusiones

Determinación de pasos futuros de investigación o desarrollo en arquitectura CUDA.

Resultado del proyecto

Como resultado de todas las actividades del proyecto se obtendrán:

- Memoria del proyecto
  - Descripción, motivación y planificación
  - Manual de puesta a punto de entorno de desarrollo para JCuda y PyCuda
  - Descripción de los Algoritmos a implementar en Java y Python. Particularidades de la implementación, métricas obtenidas.
  - Conclusiones
- Presentación

  Documento resumen del desarrollo del PFC. Describirá las fases, su alcance y ejemplos de lo obtenido. Así como algunas notas de problemas presentados y posibles próximas ampliaciones.
- Algoritmos implementados

  Código fuente de los algoritmos implementados en Java y Python, así como su documentación relativa, Javadoc y Pydoc

Plan de trabajo

La planificación va acorde a las actividades anteriormente reseñadas. La más complejas se han detallado en subtareas.

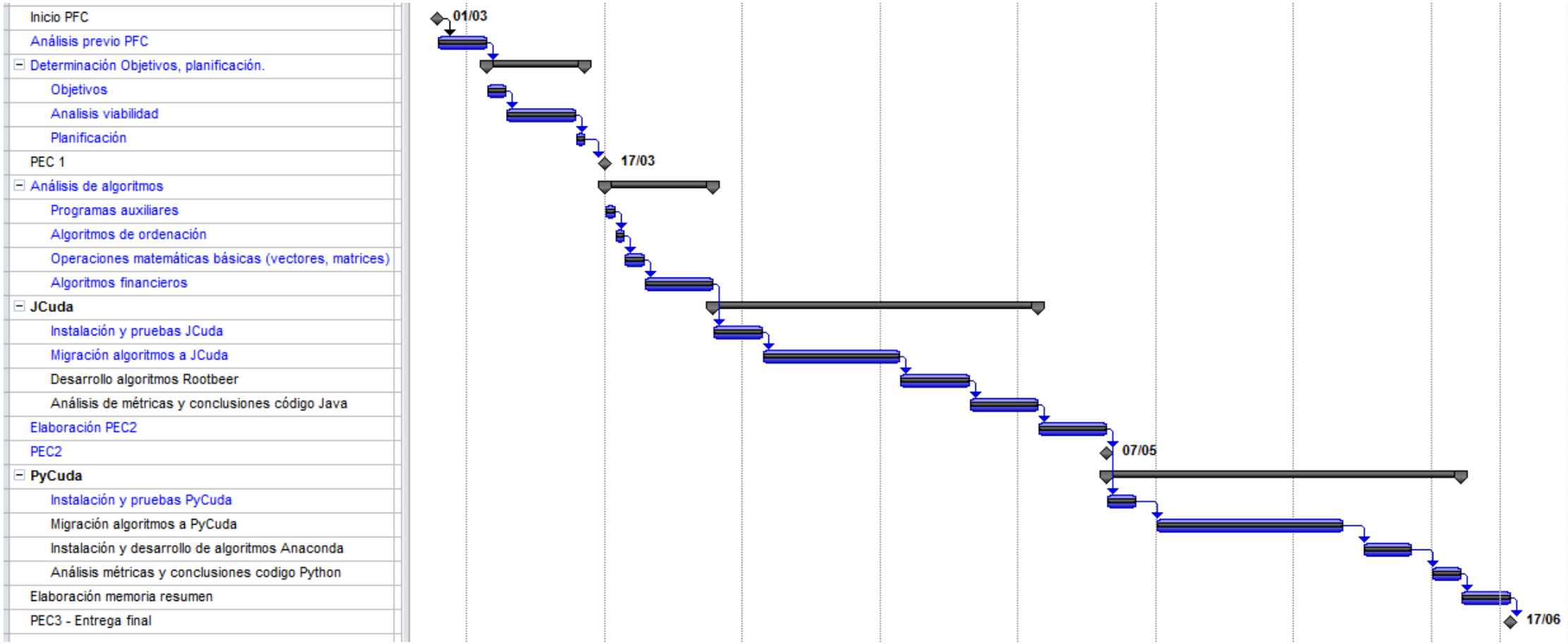
Se estima la dedicación, durante 5 días de la semana, de 2 horas diarias. Debido a compromisos personales, parte de estas horas en algún caso se desarrollarán durante el fin de semana.

El número de días estimado es de 74, 148 horas, dentryo del plazo de desarrollo del PFC.

Antes de la fecha límite de entrega de las PEC2 y PEC3 se dejará un margen de una semana para últimos ajustes antes de entrega.

Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
Inicio PFC	0 días	vie 01/03/13	vie 01/03/13	
Análisis previo PFC	3 días	vie 01/03/13	mar 05/03/13	3
Determinación Objetivos, planificación.	**8 días?**	**mié 06/03/13**	**vie 15/03/13**	**4**
Objetivos	2 días	mié 06/03/13	jue 07/03/13	
Analisis viabilidad	5 días	vie 08/03/13	jue 14/03/13	6
Planificación	1 día?	vie 15/03/13	vie 15/03/13	7
PEC 1	0 días	dom 17/03/13	dom 17/03/13	8
Análisis de algoritmos	**9 días**	**lun 18/03/13**	**jue 28/03/13**	
Programas auxiliares	1 día	lun 18/03/13	lun 18/03/13	
Algoritmos de ordenación	1 día	mar 19/03/13	mar 19/03/13	11
Operaciones matemáticas básicas (vectores, matr	2 días	mié 20/03/13	jue 21/03/13	12
Algoritmos financieros	5 días	vie 22/03/13	jue 28/03/13	13
JCuda	**23 días**	**vie 29/03/13**	**mar 30/04/13**	
Instalación y pruebas JCuda	3 días	vie 29/03/13	mar 02/04/13	14
Migración algoritmos a JCuda	15 días	mié 03/04/13	mar 23/04/13	16
Análisis de métricas y conclusiones código Java	5 días	mié 24/04/13	mar 30/04/13	17
Elaboración PEC2	5 días	mié 01/05/13	mar 07/05/13	18
PEC2	0 días	mar 07/05/13	mar 07/05/13	19
PyCuda	**21 días**	**mié 08/05/13**	**mié 05/06/13**	
Instalación y pruebas PyCuda	3 días	mié 08/05/13	vie 10/05/13	19
Migración algoritmos a PyCuda	15 días	lun 13/05/13	vie 31/05/13	22
Análisis métricas y conclusiones codigo Python	3 días	lun 03/06/13	mié 05/06/13	23
Elaboración memoria resumen	5 días	jue 06/06/13	mié 12/06/13	24
PEC3 - Entrega final	0 días	mié 12/06/13	mié 12/06/13	25

Diagrama de Gantt



Desde la estimación inicial se ha producido cierto cambio en la planificación debido a varias circunstancias.

El estudio se amplió a dos herramientas más no previstas inicialmente, Rootbeer y Anaconda Accelerate, que consideré interesantes para el objetivo del proyecto y aportaba una visión más amplia de las posibilidades de programación CUDA en estos lenguajes.

También se amplió más de lo previsto el proceso de instalación de PyCuda hasta disponer de un entorno funcional, debido a la necesidad de instalar versiones anteriores a las ya instaladas de Visual Studio y de NVIDIA toolkit. Ésta fue, a su vez, una de las motivaciones para indagar en Anaconda Accelerate. Esto provocó casi una semana de retraso sobre el plan previsto.

Al añadir más herramientas al estudio se ha limitado el tiempo de desarrollo de algoritmos inicialmente previsto, reseñando sólo algunos significativos para poder valorar la variedad de posibilidades que estas herramientas ofrecen para el aprovechamiento de las GPU en distintos ámbitos de computación, numérico, financiero, etc...

Valoración económica

Al tratarse de un proyecto de investigación y *prueba de concepto*, su valoración económica no es crítica para estimar su rentabilidad directa, pues ésta puede depender de los resultados obtenidos o de la posible aplicación de éstos en futuros proyectos que generen, ellos sí, valor directamente.

Aún así se puede establecer una valoración aproximada del coste de la realización en base al número de horas invertidas, el coste del software y el hardware y del acceso a los recursos software y documentales necesarios.

En el caso del hardware, se utilizará un ordenador PC clónico algo antiguo, con alrededor de unos 3 años, con 8gb de RAM y con un procesador Intel Core 2 Quad Q9400. Como tal se puede considerar ya amortizado.

Como inversión directa se ha realizado la adquisición de la tarjeta NVIDIA GT 630 ya comentada, en concreto el modelo, MSI GeForce GT630 Nvidia Graphics Card (4GB, PCI-E 2.0 x16), adquirida en Amazon por 60 libras, al cambio aproximadamente 70€.

El software a utilizar será en su mayor parte gratuito, como las librerías CUDA de NVIDA, e incluso Open Source, como lo son JCuda y PyCuda.

Es posible que puntualmente pueda necesitar, en el análisis de los algoritmos en C o C++, utilizar MS Visual Studio, que es una herramienta de pago. Pero en mi caso utilizaré la licencia obtenida como estudiante de la UOC. Además considero que se puede realizar utilizando la versión gratuita de la misma herramienta, VS Express, por lo que no añado coste por este concepto.

Respecto a otro software a utilizar, sistema operativo, ofimática, etc.., el ordenador que se utilizará ya dispone de Windows 7 y MS Office 2007 así como otras muchos programas o bien amortizados o bien gratuitos y Open Source.

Otro tipo de recursos necesarios, como son la electricidad o el acceso a internet serán los habitualmente incluidos en otras actividades, por lo que estimo no estimables como gastos adicionales al proyecto.

El cómputo de electricidad a 7 céntimos por hora aproximadamente sumará un coste aproximado de 10,36€.

Cabría incluir el precio de las horas de trabajo dedicadas al mismo. Al respecto indicar que están estimadas, según la planificación, unas 148 horas.

Como actualización a la estimación inicial, se debería añadir el precio de la licencia del plugin Anaconda Accelerate, 125 dólares USA. Pero al utilizar solamente la versión de prueba de 30 días no se ha realizado ningún desembolso adicional al respecto.

Entorno de Desarrollo CUDA

Para alcanzar los objetivos del proyecto se utilizarán diversos entornos. Un mismo entorno hardware, y distintos entornos de programación software.

- El hardware ya descrito en la introducción, basado en Intel QuadCore con 8Gb de RAM y tarjeta gráfica GeForce GT 630, con 96 núcleos CUDA y 4GB de RAM. El sistema operativo será Windows 64bits.

- El kit oficial de desarrollo NVIDIA CUDA Nsight, para programación en C y C++ basado en Visual Studio 2010, para sistemas operativos Windows. Existe otro NSight basado en Eclipse para sistemas operativos Linux y Mac.

- Eclipse + librerías JCuda y RootBeer para desarrollo Java.

- JetBrains PyCharm (Community Edition) para desarrollo en Python + librerías PyCuda. . Además de otras librerías útiles para implementar ejemplos concretos.

- Distribución de Python Anaconda más el *plugin* Anaconda Accelerate para desarrollo CUDA en Python.

Entorno Hardware

Las características del hardware CUDA se pueden leer a través de la API CUDA proporcionada. Lógicamente esto mismo se puede realizar desde llamadas con JCuda y PyCuda.

La llamaba básica C para esto sería:

```
cudaSetDevice(dev);
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(&deviceProp, dev);
```

Y el equivalente en Java:

```
// Leer capacidad de cómputo
int majorArray[] = { 0 };
int minorArray[] = { 0 };
cuDeviceComputeCapability(
    majorArray, minorArray, device);
int major = majorArray[0];
int minor = minorArray[0];

// Leer atributos del dispositivo
int array[] = { 0 };
List<Integer> attributes = getAttributes();
for (Integer attribute : attributes) {
 String description = getAttributeDescription(attribute);
    cuDeviceGetAttribute(array, attribute, device);
```

```
            int value = array[0];
        }
```

La lista de atributos obtenidos por el código java con el método *cuDeviceGetAttribute* en el array es equivalente al array que se paga en la función C, cudaGetDeviceProperties.

Los programas, siempre que sea posible, no utilizarán valores exactos para las llamadas paralelas, estableciendo un número exacto de bloques y threads, sino que éstos se calcularán en función del tamaño de los argumentos de entrada, (tamaño de vectores o matrices, etc…), para maximizar el nivel de paralelización, pero no contando con los 96 núcleos de la tarjeta disponible, sino consultando, con esta misma API, el número de núcleos disponibles pudiendo ser así el código reutilizable en otros dispositivos CUDA sin necesidad de recodificación.

La descripción de estos atributos para el hardware utilizado es:

Device 0: "GeForce GT 630"	
CUDA Driver Version / Runtime Version	5.0 / 5.0
CUDA Capability Major/Minor version number:	2.1
Total amount of global memory:	4096 MBytes (4294967296 bytes)
(2) Multiprocessors x (48) CUDA Cores/MP:	96 CUDA Cores
GPU Clock rate:	1620 MHz (1.62 GHz)
Memory Clock rate:	500 Mhz
Memory Bus Width:	128-bit
L2 Cache Size:	131072 bytes
Max Texture Dimension Size (x,y,z)	1D=(65536), 2D=(65536,65535), 3D=(2048,2048,2048)
Max Layered Texture Size (dim) x layers	1D=(16384) x 2048, 2D=(16384,16384) x 2048
Total amount of constant memory:	65536 bytes
Total amount of shared memory per block:	49152 bytes
Total number of registers available per block:	32768
Warp size:	32
Maximum number of threads per multiprocessor:	1536
Maximum number of threads per block:	1024
Maximum sizes of each dimension of a block:	1024 x 1024 x 64
Maximum sizes of each dimension of a grid:	65535 x 65535 x 65535

Maximum memory pitch:	2147483647 bytes
Texture alignment:	512 bytes
Concurrent copy and kernel execution:	Yes with 1 copy engine(s)
Run time limit on kernels:	Yes
Integrated GPU sharing Host Memory:	No
Support host page-locked memory mapping:	Yes
Alignment requirement for Surfaces:	Yes
Device has ECC support:	Disabled
CUDA Device Driver Mode (TCC or WDDM):	WDDM (Windows Display Driver Model)
Device supports Unified Addressing (UVA):	Yes
Device PCI Bus ID / PCI location ID:	2 / 0

Compute Mode:

CUDA Driver = CUDART,

CUDA Driver Version = 5.0

CUDA Runtime Version = 5.0,

NumDevs = 1,

Device0 = GeForce GT 630

NVIDIA CUDA Toolkit v5.0

El desarrollo para la arquitectura CUDA se basa en el kit de desarrollo oficial de NVIDIA para CUDA. Éste soporta los lenguajes C y C++ y los IDE de desarrollo Visual Studio 2008 y 2010 en sistemas operativos Windows, y Eclipse para Linux y Mac OS X.

Incluye el *driver* de conexión con la tarjeta CUDA y un conjunto de librerías, documentación, ejemplos y herramientas como un precompilador para las directivas CUDA en el código C y C++ y herramientas de depuración para los nucleos CUDA. El *driver* de conexión CUDA es la base de la comunicación entre la CPU *host* y la GPU, tanto para datos como para código.

Los plugin que integran el desarrollo con las herramientas Visual Studio y Eclipse se llaman, respectivamente, NVIDIA® Nsight™ Development Platform, Visual Studio Edition y NVIDIA® Nsight™ Development Platform, Eclipse Edition.

Aunque vayamos a desarrollar en otros lenguajes, como Java o Python, desafortunadamente todavía se requiere el kit de desarrollo oficial NVIDIA, pues para llegar a la máxima optimización programando directamente los núcleos CUDA todavía precisa de código CUDA-C, el cual, a su vez, debe ser compilado con esta herramienta para poder funcionar. Sin embargo, como se mostrará, se trata de una parte muy acotada del código a utilizar en casos muy concretos.

A su vez, esa parte del código se puede incluir en el proyecto Java o Python, bien como un *interfaz nativo*, o bien pudiendo incrustar el código C dentro de un objeto String dentro del código que puede compilarse en momento de *ejecución* del código Java a ejecutar.

Esto último presenta la desventaja de que se precisa, allá donde se ejecute, que el entorno de compilación NVIDIA CUDA esté configurado para ser utilizado pero también ofrece la ventaja de que será totalmente portable y que podrá compilarse de forma optimizada para la plataforma donde se ejecute.

Código de Núcleo CUDA

El código que se ejecuta dentro del núcleo CUDA debe ser código C o C++ compilado para la GPU usando el kit oficial NVIDIA. Esto se debe a que no hay una máquina virtual Java que funcione directamente en los núcleos de la GPU.

Este código debe compilarse con el compilador NVCC del kit oficial, que es el que genera el código ensamblador/máquina que admiten las GPU.

El compilador puede generar dos tipos de fichero:

- CUBI: Es código CUDA ya compilado en binario. Puede ser el código compilado para uno o varios dispositivos específicos distintos.
- PTX: Se trata de código CUD A compilado pero en (pseudo) ensamblador. Es finalmente convertido en binario ejecutable una vez cargado en el dispositivo. Permite código más portable con optimización en momento de ejecución.

Como se puede comprobar, los ficheros en formato CUBIN, aunque pueden incluir varias versiones de ejecutable, están limitados a éstas, especialmente a la capacidad de computación, *Computer Capability*, de la GPU para la o las que fue compilado.

Será por tanto, en general, preferible utilizar el formato PTX para generar un código lo más portable posible, lo cuál puede ser importante a la hora de utilizar lenguajes como Java y las librerías JCuda.

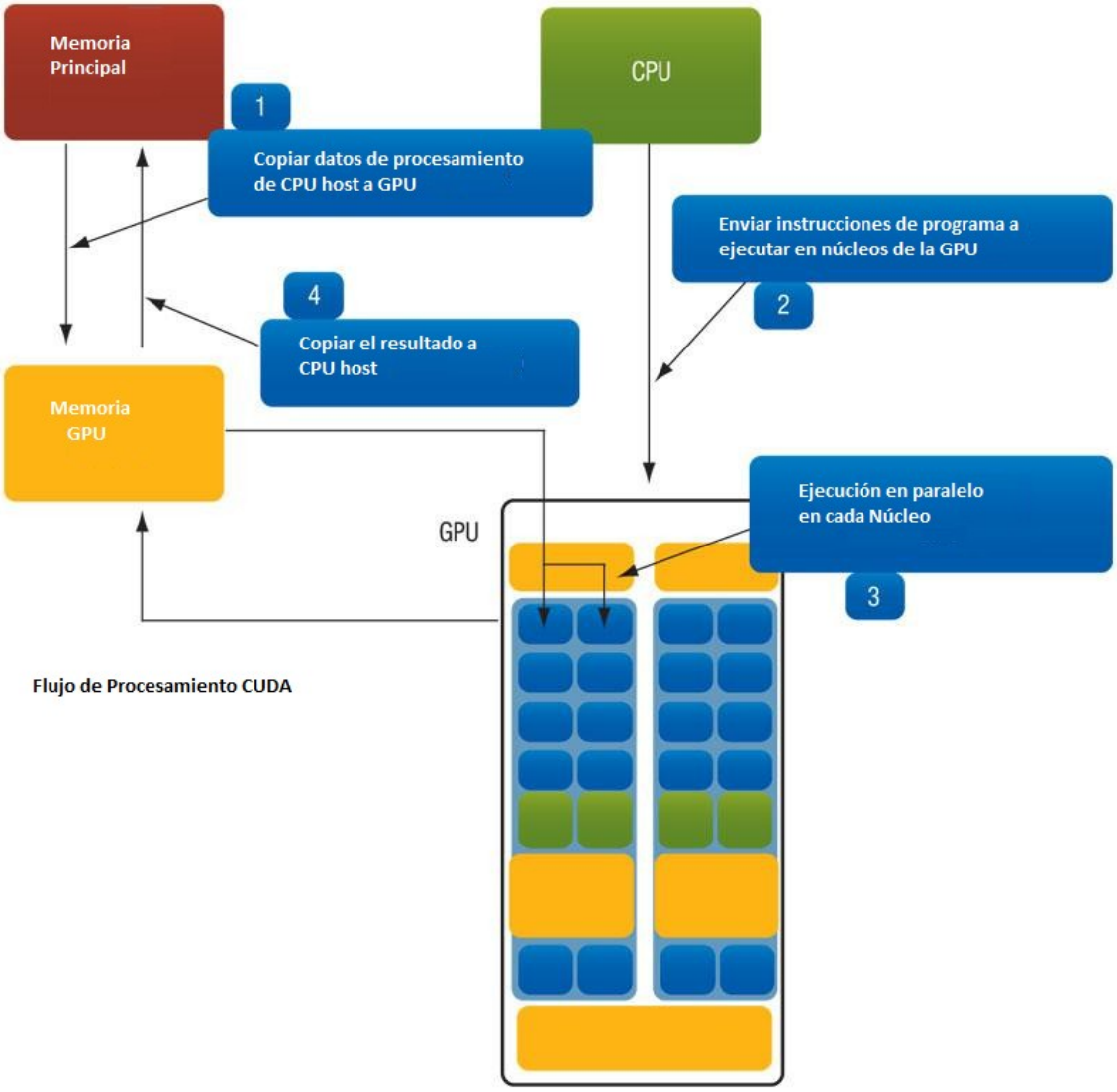
Para generar un ejecutable en formato CUBIN a partir del compilador nvcc de NVIDIA se especificará la o las capacidades de computación para las que se quiere generar código objeto y tamaño del bus de la GPU de destino, así, para una GPU como la utilizada en el proyecto, con 128 bit y capacidad de computación 2.1, una instrucción de compilación típica sería:

```
nvcc -cubin -m128 -arch sm_21 JCudaVectorAddKernel.cu -o
JCudaVectorAddKernel.cubin
```

Para generar un *ejecutable* PTX a partir del mismo código fuente la instrucción al compilador sería:

```
nvcc -ptx VectorAddKernel.cu -o VectorAddKernel.ptx
```

La ejecución de un programa CUDA, independientemente del lenguaje de implementación, vendrá dado por el siguiente diagrama de flujo:



Flujo de datos e instrucciones de un programa CUDA

La óptima comunicación entre la CPU host y la GPU, así como el adecuado balanceo del procesamiento paralelo determinará en gran medida la ganancia de rendimiento obtenida por el uso de la GPU para un algoritmo concreto.

Lógicamente no todos los algoritmos son adecuados para ello y en algunos puede verse penalizado el rendimiento debido al sobre coste del esfuerzo de paralelización y de comunicación entre la CPU y la GPU.

Desarrollo CUDA en lenguaje Java

Hace algunos años, al poco de surgir la arquitectura CUDA, la única forma de interactuar con ésta mediante lenguaje Java se limitaba al uso de JNI, (*Java Native Interface*), con el que establecer *bindings* con el código CUDA-C, [Bot11].

Al poco, surgieron ciertas iniciativas para facilitar la computación CUDA en Java sin tener que recurrir, al menos explícitamente, a JNI, y procurando encapsular la mayor parte de la funcionalidad con librerías Java puras. La más veterana de estas iniciativas, y además open source, es JCuda. Otras librerías más recientes como *JCUDA* comercial, o *rootbeer* aportan aproximaciones algo diferentes, como la generación de código *ptx* directamente desde Java.

Al presentar las mayores diferencias entre ambas, JCuda y rootbeer han sido las analizadas en este proyecto.

JCuda

JCuda permite el desarrollo de aplicaciones Java que utilicen la potencia de la arquitectura CUDA.

JCuda es básicamente un conjunto de librerías que enlazan una API Java con las librerías NVIDIA CUDA, así como otro conjunto de clases auxiliares.

Para utilizarlas, por tanto, es preciso disponer, además de una GPU CUDA, del Kit oficial NVIDIA CUDA.

Como decisión de diseño se ha optado porque estas librerías imiten lo más fielmente posible a la API original en C/C++, optando, por tanto del uso de métodos estáticos, imitando lo más posible la signatura y semántica, salvo, obviamente, en el manejo de punteros.

Si bien en un primer momento esto pueda resultar poco adecuado, no impide poder realizar, sobre esta capa, un diseño más orientado a objeto, utilizando los patrones adecuados.

Presenta la limitación ya comentada respecto a la codificación de la ejecución dentro de los núcleos CUDA, que todavía, en la versión utilizada en el proyecto, JCuda 0.5.0a, debe realizarse en C o C++ CUDA, bien utilizando código ya compilado con NVIDIA CUDA Kit para la plataforma donde se desarrolle y compile el proyecto, o bien utilizando código fuente en C/C++ añadido al proyecto Java o incluso embebido dentro del código Java y que se compilará en el momento de ejecutar ese código Java concreto.

Para esto último se precisará configurado, incluso en momento de ejecución, el NVIDIA CUDA Kit y el compilador c/c++ nativo de la plataforma (GNU C Compiler, MS VS C Compiler, etc…).

Conjunto de librerías JCuda

Las librerías JCuda proporcionan enlaces con la API Driver de CUDA.

Se distribuyen en 6 ficheros jar:

- JCuda: Permite interactuar con el dispositivo CUDA. Es la capa real de integración con la capa nativa. Permite la gestión de memoria, la compilación y carga de programas en los núcleos CUDA y la comunicación entre la GPU y el host.

El resto de librerías son específicas para distintos ámbitos. Éstas encapsulan las librerías provistas por NVIDIA en su CUDA *tookit*, las llamadas "*GPU Accelerated Libraries*", [Nvd13e], cuyo nombre es idéntico al de la equivalente JCuda sin la J inicial, éstas son:

- JCublas: Permite el uso de la librería nativa CUBLAS, de NVIDIA, que es una implementación optimizada para CUDA de funciones de optimización de álgebra lineal. Se organiza en 3 niveles, siendo éstos:
    1. Operaciones entre vectores
    2. Operaciones matriz - vector
    3. Operaciones entre matrices

- JCuff: Enlaza con la librería nativa CUDA FFT, que provee de una implementación optimizada para CUDA de las transformadas de Fourier.

- JCudpp: Integra con la librería nativa para implementar operaciones con datos masivamente paralelos, incluyendo ordenación, escaneo paralelo, operaciones matemáticas con matrices dispersas, etc...

- JCusparse: Implementa funciones matemáticas de la librería BLAS integrando conversión de matrices dispersas a través de la librería CUDA CUSPARSE.

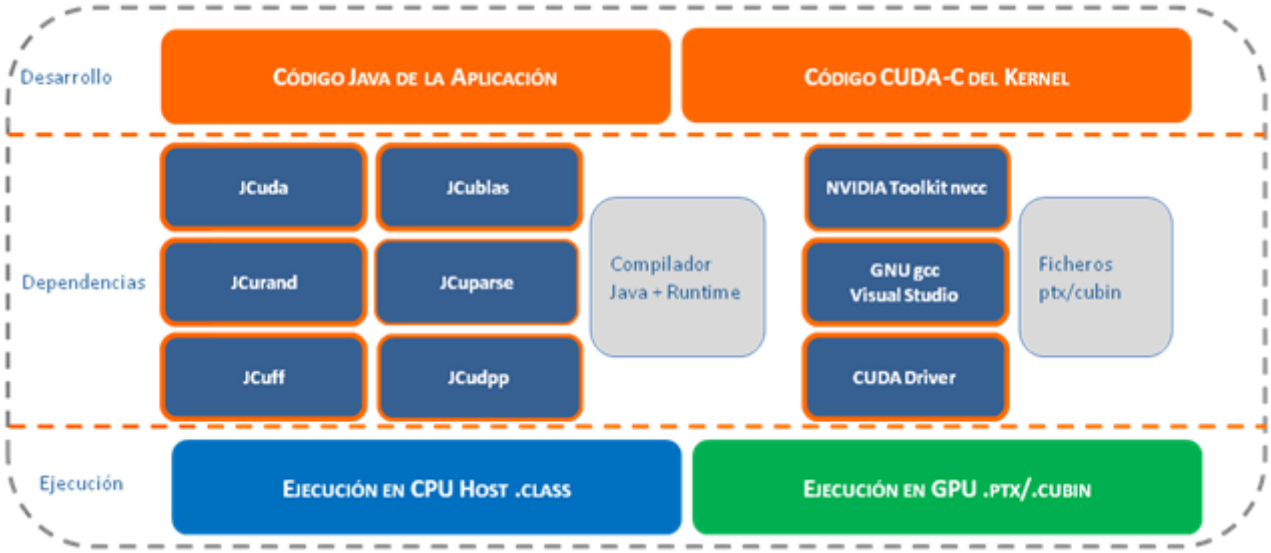
- JCurand: Implementa generación de números aleatorios utilizando el generador CUDA CURAND.

Muchos de los ejemplos implementados, en general la generación de números aleatorios, operaciones con matrices, etc…, podrían implementarse con llamadas a estas librerías, pero en algunos casos se han implementado totalmente sin utilizarlas para mostrarlo como ejemplo de codificación de dicho algoritmo (por ejemplo, la multiplicación de matrices), sin usar directamente JCublas.

En otras librerías para soportar CUDA, como PyCuda o Anaconda, también se han implementado *wrappers* para estas librerías.

Componentes y dependencias de un programa JCuda

Los componentes y dependencias de un proyecto JCuda descritos en los dos anteriores apartados podrían resumirse en el siguiente diagrama:



Librerías y herramientas para Compilación y Ejecución JCuda

Un programa JCuda constará de una parte, la más importante, desarrollada en Java y de una o varias funciones a ejecutar en el *kernel* de la GPU implementadas en CUDA-C. Opcionalmente puede hacer uso de las librerías auxiliares que encapsulan la funcionalidad de las CUDA Accelerated Libraries, CURAND, CUBLAS, etc…

El código Java tendrá las librerías utilizadas, además de JCuda siempre, como dependencias y deberá ser compilado con un *Java Development Kit* certificado. Para ejecutarse en la CPU deberá disponer también de un *Java Runtime Environment* compatible con el compilador.

El código CUDA-C será pre-compilado con la utilidad nvcc del NVIDIA CUDA Toolkit, y será enlazado usando bien gcc, como *linker*, en entornos Linux o Mac, o bien, en entornos Windows utilizando en *linker* de Visual Studio.

Finalmente cada porción del código será ejecutado bien en la CPU o en la GPU a través del CUDA *driver*, que también sincronizarála comunicación entre CPU host y la GPU.

Uso de código compilado nativo para ejecución en GPU. CUDA C.

El código C nativo utilizado para las funciones que se ejecutan en la GPU puede ser compilado aparte del código Java, utilizando el SDK CUDA y generando los ejecutables en el formato que se precise, *ptx* o *cubin*. Esto presenta la ventaja de que la ejecución del código JCUDA - GPU no precisaría el Kit de desarrollo de CUDA instalado en la máquina donde se fuese a ejecutar, sino tan sólo el JRE y el driver NVIDIA correspondiente.

Otra posibilidad sería incluir esa fase de compilación en el propio proyecto Java y que éste sea compilado en el momento de la ejecución del código Java. Sin duda esta opción es de las más adecuadas para fases de desarrollo y test, y es la opción seguida en los algoritmos JCUDA implementados en el proyecto.

Para utilizar código C embebido en el código Java para ser compilado (y ejecutado) en tiempo de ejecución del código Java se precisa que el entorno de compilación de CUDA y el compilador de C estén habilitados en tiempo de ejecución.

Los pasos básicos que se realizan en cada programa JCuda que use programación específica del núcleo son:

- Preparación del fichero de código fuente C/C++:

```
ptxFileName = preparePtxFile("reduction.cu");
```

Compilación y carga del código compilado en memoria del núcleo:

```
// Load the module from the PTX file
      module = new CUmodule();
      cuModuleLoad(module, ptxFileName);

      // Obtain a function pointer to the "reduce" function.
      function = new CUfunction();
      cuModuleGetFunction(function, module, "reduce");
```

```
static native int cuModuleGetFunctionNative
(CUfunction hfunc, CUmodule hmod, String name);
```

Al ejecutar el código java que *prepara* (en realidad compila) este código se vería la llamada en el sistema operativo del tipo:

```
nvcc -ptx -o reduction.cu -o reduction.ptx
```

Obviamente también se puede preparar para generar binarios en formato CUBIN. Para esto hay que indicar, además, el tipo de destino y y capacidad de computación del mismo, por ejemplo para la GPU que utilizaremos, una GPU de 128 bits con capacidad de cómputo 2.1:

```
nvcc -cubin –m128 -arch sm_21 reduction.cu -o reduction.cubin
```

La función cuModuleLoad admite ambos formatos de ejecutable.

- Por último, y una vez inicializados los parámetros de llamada al núcleo (argumentos, bloques, threads, etc..), se realiza la llamada a la ejecución *nativa* en el núcleo:

```
// Call the kernel function.
cuLaunchKernel(function,
       blocks,  1, 1,          // Grid dimension
       threads, 1, 1,          // Block dimension
       sharedMemSize, null,    // Shared memory size and stream
       kernelParameters, null // Kernel- and extra parameters
       );
```

Código C del núcleo embebido en código fuente Java

Una tercera posibilidad que ofrece JCUDA en incluir el código C necesario como un objeto String dentro del código Java, y éste es compilado de forma similar a la anterior.

Los pasos básicos son idénticos, salvo que se utilizará la clase KernelLauncher que encapsula la compilación del código fuente, a partir de un objeto String Java, y la llamada a la ejecución del mismo:

- Incluir el código fuente dentro de un objeto String de Java:

```
String sourceCode =
"extern \"C\"" + "\n" +
"__global__ void add(float *result, float *a, float *b)" + "\n" +
      "{" + "\n" +
      "    int i = threadIdx.x;" + "\n" +
      "    result[i] = a[i] + b[i];" + "\n" +
      "}";
```

- Ejecutar la compilación de dicho código, obteniendo un objeto de la clase KernelLauncher:

```
KernelLauncher kernelLauncher =
            KernelLauncher.compile(sourceCode, "add");
```

El método compile internamente genera un fichero en formato cu, con el nombre que se pasa como parámetro en la llamada *compile*, en este caso sería *add.cu*, y que se almacenará en un directorio temporal y será compilado en formato PTX. Ese código compilado se almacena en memoria y se almacena la dirección en el objeto kernelLauncher.

- Ejecutar el código del núcleo obtenido:

```
. . .
kernelLauncher.setBlockSize(size, 1, 1);
kernelLauncher.call(dResult, dA, dB);
. . .
```

Antes de la llamada se le pasarán los parámetros habituales para las llamadas a código de núcleo: tamaño del bloque, número de threads y la configuración de la memoria (global, shared, stream, etc..).

Gestión de Memoria con JCUDA

La gestión de memoria es un aspecto importante en el rendimiento de las aplicaciones paralelas en CUDA en general, y con JCuda se dispone de una flexibilidad cercana a C o C++. Los tipos de memoria que manejan los programas CUDA son:

Memory	Location	Cached	Access	Scope	Lifetime
Register	On chip	N	R/W	1 thread	Thread
Local	RAM	N	R/W	1 thread	Thread
Shared	On chip	N	R/W	Threads in a block	Block
Global	RAM	N	R/W	All thread + host	Host allocation
Constant	RAM	Y	R	All thread + host	Host allocation
Texture	RAM	Y	R	All thread + host	Host allocation

[Cour12]

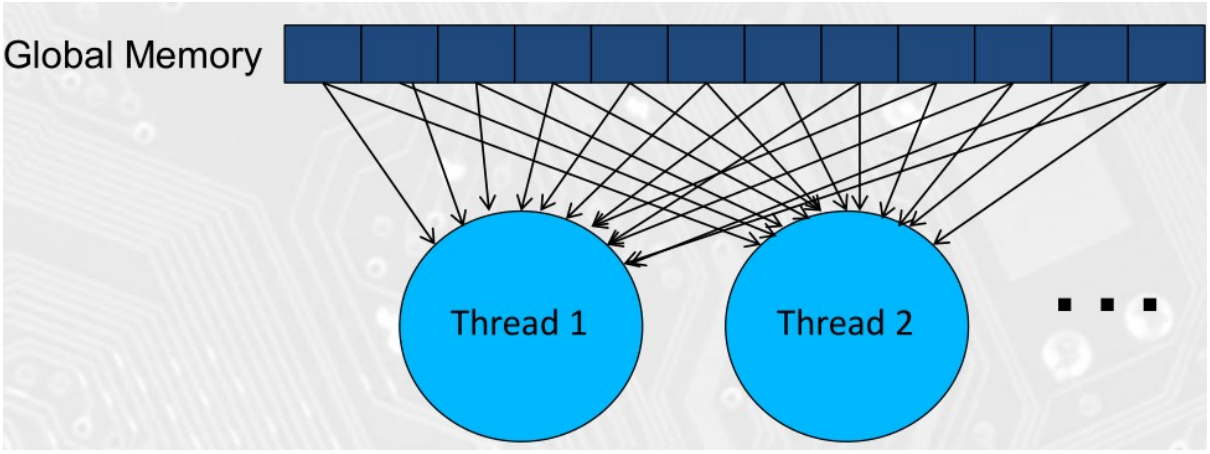
Lógicamente dentro del código del núcleo se puede indicar cualquiera de los ámbitos de memoria dentro de la GPU, *Register, Local* o *Shared*, mientras que los modos de memoria definidos en el Host se harán con la clase JCUDA Pointer, *Global, Constant* y *Texture*, por ejemplo:

```
Pointer punteroGPU = new Pointer();

// Reserva de la memoria para el puntero
JCuda.cudaMalloc(punteroGPU, 4 * Sizeof.FLOAT);

// Asignación de un puntero a un array definido
float array = new float[8];
Pointer punteroHost = Pointer.to(array);

// Indicar un desplazamiento en la posición del puntero
Pointer punteroHostDespl = punteroHost.withByteOffset(2 * Sizeof.FLOAT);

// Copiar 4 elementos desde la mitad del array Java a la GPU

// En la memoria Global
JCuda.cudaMemcpy(punteroGPU, punteroHostDespl, 4 *
Sizeof.FLOAT, cudaMemcpyKind.cudaMemcpyHostToDevice);
```

La reserva o liberación memoria en el kernel debe hacerse al modo de C, con malloc o free. En el Host la memoria reservada con las funciones JCuda.malloc deben liberarse igualmente con JCuda.cudaFree. Con JCuda se debe reservar la memoria y liberarla para los datos que se envían y reciben de la GPU.

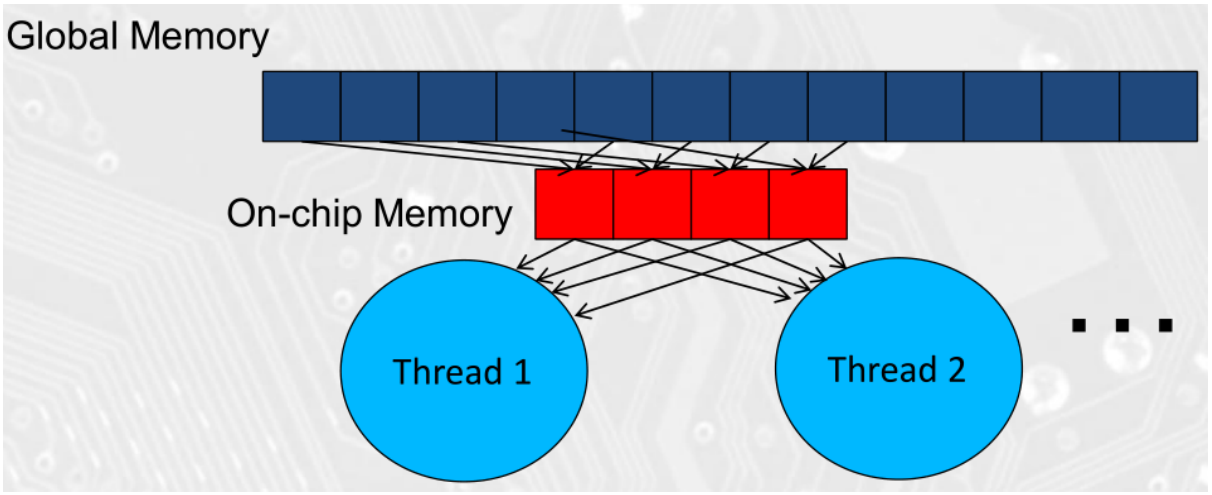
Patrones de acceso a memoria

En la programación para CUDA, hay un aspecto importante que afecta mucho al rendimiento que es la comunicación de datos entre el *host* y la GPU, así como la latencia de las distintas memorias respecto a los *cores.* Para ello hay distintos patrones de acceso a memoria desde los *threads* de ejecución, siendo los más comunes los de acceso global y los de *tiles,* teselas:



Global Memory Access Pattern [Cour12]

Si bien es un patrón que funciona, presenta un grave problema de congestión (y latencia) en la transmisión de datos, y hay otros patrones que se utilizan en la programación CUDA que optimizan esta comunicación, como es el patrón de acceso en *tesela*, (tiles):



Shared Memory Tiling Pattern [Cour12]

El funcionamiento de este patrón es el siguiente:

- Se identifica un *tile* (o tesela) de la memoria global a la que acceden múltiples *threads.*

- Se carga esa posición de memoria en el chip (*shared memory)*

- Se permiten a múltiples chips acceder a sus datos

- Se carga la siguiente tesela

Ésta es una optimización que consigue una gran ganancia de rendimiento (reduciendo latencia de datos) en casos donde los *threads* acceden a los datos en el mismo momento (o muy cercano). Lo cual suele ser así dado que en la mayor parte de las computaciones paralelas CUDA todos los *threads* se ejecutan en un tiempo similar.

Si por características del programa estos tiempos de acceso difiriesen éste modelo de acceso supondría, por contra, un cuello de botella. El uso del segundo patrón en JCuda sería análogo a como se haría en su equivalente C o C++, adaptando el número de *blocks* y *threads* en la llamada, y adaptando la memoria *shared* del código C del núcleo al tamaño de la tesela:

```
// Llamada a una función con modelo memoria tiles de 16x16

int blockW = TILE_WIDTH; //Tamaño de la tile

int blockH = TILE_WIDTH;

int gridW = width / blockW; //Número tiles

int gridH = height / blockH;


    JCudaDriver.cuLaunchKernel(

          function,

          gridW, gridH, 1, //tiles de TILE_WIDTHx TILE_WIDTH

          blockW, blockH, 1, //número de tiles

          0, null, // Memoria Shared

          kernelParameters, null // Otros parámetros

    );

    JCudaDriver.cuCtxSynchronize();
```

En el código C del kernel, las variables que se utilizarán estarán en relación al tamaño de esta tesela:

```
 __shared__ float ds_A[TILE_WIDTH][TILE_WIDTH];

 __shared__ float ds_B[TILE_WIDTH][TILE_WIDTH];
```

Paso de parámetros de host a funciones kernel GPU

Arrays multidimensionales

Los argumentos por defecto que soporta JCuda son arrays de punteros. Para poder *simular* argumentos que sean arrays multidimensionales se crean arrays de punteros, con tanto nivel de anidación como dimensiones se precisen.

```
/* Ejemplo de argumento array de 2 dimensiones con un número de filas
    'numThreads' y un número de columnas con valores de 0 a size-1. */

      float hostInput[][] = new float[numThreads][size];

      for (int i = 0; i < numThreads; i++)

      {

       for (int j=0; j<size; j++) {

             hostInput [i][j] = (float)j;

         }

      }


/* Reservar memoria para los arrays en la GPU, uno por cada fila. Los objetos
Pointer de estos arrays se almacenan en la memoria del host. */

      CUdeviceptr hostDevicePointers[] = new CUdeviceptr[numThreads];

      for(int i = 0; i < numThreads; i++) {

          hostDevicePointers[i] = new CUdeviceptr();

          cuMemAlloc(hostDevicePointers[i], size * Sizeof.FLOAT);

      }


/* Se transmite el contenido de las filas desde el host a los arrays en el
dispositivo que se acaban de copiar */

      for(int i = 0; i < numThreads; i++)  {

     cuMemcpyHtoD(hostDevicePointers[i], Pointer.to(hostInput[i]),
                 size * Sizeof.FLOAT);

      }
```

```
/*  Reservar memoria en el dispositivo para el array de punteros y copiar el
      array de punteros desde el host a la GPU. */

        CUdeviceptr deviceInput = new CUdeviceptr();

        cuMemAlloc(deviceInput, numThreads * Sizeof.POINTER);

        cuMemcpyHtoD(deviceInput, Pointer.to(hostDevicePointers),

            numThreads * Sizeof.POINTER);
```

Como se puede comprobar no es, en principio, tan sencillo como transmitir simplemente un argumento de tipo array multidimensional, pero es factible y, por lo tanto, se podría codificar este comportamiento de forma más genérica empleando, por ejemplo, anotaciones Java.

Otra opción más sencilla, si bien implica ciertos cambios adicionales en los algoritmos y en la signatura de las operaciones es de simular los arrays bidimensionales en un Array unidimensional, añadiendo como parámetro adicional el número de columnas de la matriz. Así, la fila 0 de la matriz iría desde la posición 0 hasta la posición **numColumn – 1**, y a partir de la posición **numColumn** del array comenzaría la segunda fila, hasta la posición del elemento **2*numColumn – 1**.

Ésta opción es la que se ha tomado en los algoritmos de operaciones con matrices implementados en el proyecto.

Argumentos de tipo estructurado (tipo Struct de C)

Existe una limitación inherente a las primeras versiones de JNI que JCuda ha heredado. Se debe a la imposibilidad de *mapear* tipos de datos *struct* de C en Java. Esto se debe a que esta posibilidad fue incluida en JNI (Java Native Access ahora, ***JNA API 3.2.7***), con la clase Structure , y JCuda partió de una versión anterior a este soporte.

Un ejemplo de esta limitación es, por ejemplo, un código que funciona bien en C, que pasa el código de una accion, el día y el precio, con un array de structs del tipo:

```
typedef struct {

  char code[3];

  char date[8]

  float adjustedClose;

      } share;
```

Para pasar un array de estos objetos al kernel de la GPU, para cálculos, basta con hacer:

```
share *aShare;

share gpuShare;

aShare =(share*)malloc(sizeof(aShare)*size);

// Se inicializa la memoria para la GPU así, siendo "size" el número de objetos share en el array que se pasa a la GPU:

cudaMalloc((void**)&gpuShare,sizeof(gpuShare)*size);

// Y se copian los datos

cudaMemcpy(gpuShare,aShare,sizeofgpuShare)*size,cudaMemcpyHostToDevice);
```

Esto no es posible, en principio, en Java. La solución propuesta, simulando otros ejemplos y por respuestas en foros es pasar 3 arrays, una para el código, otro para la fecha y otro para el precio, manteniendolos relacionados por el número de índice. Es decir:

```
byte shareCode[] = new byte[size*3];

byte shareDate[] = new byte[size*8];

float shareAdjPrice[] = new float[size];

// Hay que preparar entonces 3 punteros para pasar los datos al kernel, 1 para el código, otro para la fecha, y otro para el precio, siendo también size el número de elementos:

CUdeviceptr gpuShareCode = new CUdeviceptr();

cuMemAlloc(gpuShareCode , Sizeof.BYTE*3*size);
```

```
cuMemcpyHtoD(gpuShareCode, Pointer.to(shareCode), Sizeof.BYTE*3*size);

CUdeviceptr gpuShareDate = new CUdeviceptr();

cuMemAlloc(gpuShareDate , Sizeof.BYTE*8*size);

cuMemcpyHtoD(gpuShareCode, Pointer.to(shareDate), Sizeof.BYTE*3*size);

CUdeviceptr gpuShareAdjPrice = new CUdeviceptr();

cuMemAlloc(gpuShareAdjPrice, Sizeof.FLOAT*size);

cuMemcpyHtoD(gpuShareAdjPrice, Pointer.to(shareAdjPrice ),
Sizeof.FLOAT*size);
```

Se prepara un objeto de tipo Pointer, que incluye todos estos objetos que se copiarán a la memoria de la GPU,

```
  Pointer kernelParameters = Pointer.to(

          Pointer.to(gpuShareCode),

          Pointer.to(gpuShareDate ),

          Pointer.to(gpuShareAdjPrice)

         );
```

Y ya se llama a la ejecución paralela en Kernel:

```
cuLaunchKernel(calculo,blocks, 1, 1,threads, 1, 1,0, null,kernelParameters,
null);
```

Aparentemente el objeto "Pointer" que gestiona en JCuda el trasiego de la información es donde está la limitación, pues sólo soporta arrays de tipos básicos, no objetos más completos.. Como mucho un array de otros punteros.., pero más complicado de manejar.

Una posibilidad rápida de obtener un mapeo *directo* entre un objeto Java y un *struct* de C puede ser utilizando la librería *JOCL-structs-0.0.1a-alpha.jar* de JOCL, [Jocl13], que es un proyecto *gemelo* de JCuda, para *bindings* de Java con OpenCL.

Aún así, el mapeo de *structs* es complejo, sobre todo el alineamiento en memoria de los campos desde una clase Java a un tipo en C, debiendo ajustar muy bien los tamaños de los objetos en el *host* y en la GPU, por lo que está, por ahora, desaconsejado.

Entorno de desarrollo JCuda

Para la programación de los algoritmos del proyecto, se ha partido de la instalación previa de NVIDIA CUDA toolkit, versión 5.0, con la última versión del *driver* NVIDIA con soporte CUDA.

En concreto para JCuda se ha utilizado la versión de Java 1.7.0_21 de 64 bits, sobre Windows 7 64bits., las librerías JCuda 0.5.0a y Eclipse Juno como IDE.

Las librerías JCuda son librerías Java pero incluyen las librerías dinámicas nativas, *dll*, que enlazan el código Java con el código C del *driver* CUDA que comunica con la GPU. Se han añadido al entorno de desarrollo Eclipse como librerías de usuario, con su correspondiente *dll*, y sus ficheros de código fuente y *javadoc*.

La instalación no ha supuesto dificultad, siendo similar al uso de cualquier otra librería Java con interfaz nativo, como por ejemplo JOGL.

El desarrollo y ejecuión de un programa JCuda vendrá dada por:

1 **Desarrollo del código Java**. Que incluirá la preparación de los argumentos de entrada, conversión de datos, y configuración de la memoria para la ejecución CUDA.

2 **Desarrollo CUDA C** que se ejecutará en el núcleo de la GPU. La parte del código que se ejecutará en paralelo.

3 **Compilación código Java**. Precisará el JDK instalado.

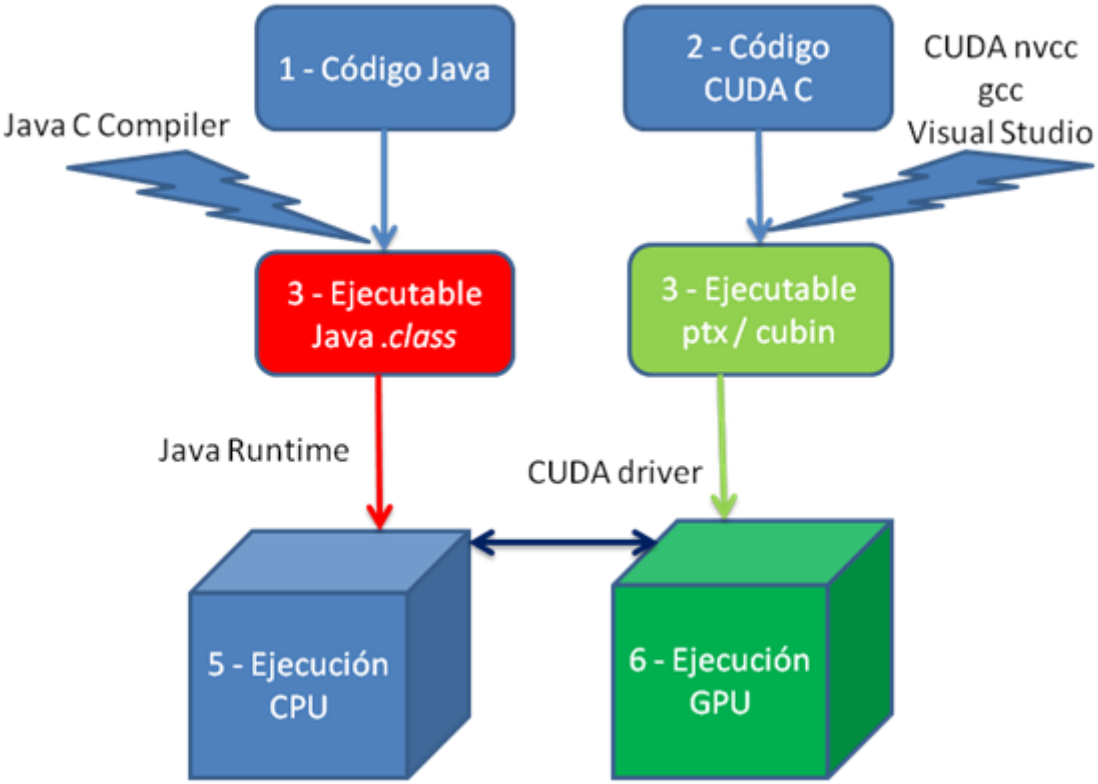
4 **Compilación código CUDA C** en formato ejecutable en la GPU, sea *ptx* o *cubin*. Precisará CUDA toolkit y entorno de compilación nativo correspondiente según el sistema operativo, gcc o Visual Studio.

Esta compilación del código Java y CUDA C no tiene que realizarse siempre, obviamente, sino que para distribución de los ejecutables se podrían entregar los códigos binarios Java, *class*, y los compilados CUDA, *ptx* o *cubin*. Pero la máquina donde se vaya a ejecutar sí debería tener instalados tanto el Java Runtime Environment como el CUDA driver correspondientes.

5 **Ejecución en la CPU**. En esta fase el código Java se ejecuta, inicializando los argumentos de entrada y la configuración de la memoria en la GPU según programación. Si bien la memoria en Java se gestiona automáticamente, esto no es así en lo relativo a la GPU, donde hay que realizar la reserva y liberación explícitamente. En esta fase es donde se envía el código ejecutable a la GPU así como los datos de entrada.

6 **Ejecución en la GPU**. El código compilado ptx o cubin se ejecuta en la GPU. La comunicación con la GPU para recibir tanto el código como los datos, y devolver los resultados, se realizarán utilizando el CUDA driver.

Flujo de Ejecución programa JCuda



Compilación y ejecución aplicación JCuda

Algoritmos Implementados en JCuda

La implementación de los algoritmos ha partido de los ejemplos y consejos ofrecidos de implementación para las operaciones matemáticas principalmente del libro de Kirk, David B., and W. Hwu Wen-mei, [Kdh12] y para el resto de ejemplos principalmente del mismo NVIDIA CUDA toolkit.

Todos estos ejemplos están desarrollados bien en lenguaje C o C++, con lo que han sido portados y adaptados a lenguaje Java, bien con JCuda o Rootbeer.

En el caso de los algoritmos de operaciones con vectores y matrices, como se comentó anteriormente, las "*GPU Accelerated Libraries*", [Nvd13e] de NVIDIA facilitan en gran medida las operaciones, pues ya implementan los cálculos más habituales de serie, y normalmente serían las utilizadas en estos casos para un óptimo rendimiento. Sin embargo para el proyecto se han implementado estas operaciones directamente, sin el uso de JCublas, salvo para un test en la multiplicación de matrices, para comprender mejor la forma de codificación con JCuda y sus peculiaridades.

Un punto clave en el rendimiento de una aplicación CUDA, que se explica detalladamente en el algoritmo de multiplicación de matrices, es el manejo de la memoria compartida, *shared memory*, como caché de datos entre la CPU *host*, y la GPU, para minimizar la comunicación entre ambas, que suele ser el cuello de botella del rendimiento en este tipo de programas.

Cada ejemplo tendrá una parte del código implementado en CUDA-C, que será la parte que se ejecutará en cada núcleo de la GPU, mientras que la parametrización y configuración de la memoria (reserva, mapeo, liberación en la GPU, etc..), y la obtención de los resultados será en lenguaje Java y JCuda.

En la memoria me extenderé algo más en los primeros algoritmos, al ser más sencillos y base para los demás, describiendo para los más complejos tan sólo las particularidades específicas de su implementación en JCuda.

Operaciones con Vectores, suma, resta e inversión.

Suma, resta e inversión

Para las distintas operaciones con Vectores y Matrices se ha implementado una clase común de utilidad, *Utility*, que implementa la funcionalidad común relativa a la inicialización del *CUDA driver*, la generación de vectores y matrices aleatorias como argumentos de entrada, el cálculo del número de bloques y threads óptimo según el tamaño de la entrada, etc..

Las operaciones suma, resta, inversión y reducción de vectores se han implementado en un único fichero de código fuente CUDA C, *VectorOperationKernel.cu,* ya que, salvo la reducción son bastante sencillas y ninguno precisa de llamadas a otras funciones.

Código CUDA C de ejecución de la suma de vectores

```
extern "C"
__global__ void add(int n, float *a, float *b, float *sum)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i<n)
    {
        sum[i] = a[i] + b[i];
    }

}
```

Se trata de un algoritmo sencillo donde solamente se ha optimizado el número de bloques y threads utilizados según el tamaño de los argumentos de entrada para optimizar el uso de la GPU.

```
public int getNumBlocks(int inputSize)

    {

        int blocks = 0;

        int threads = getNumThreads(inputSize);

        blocks = (inputSize + (threads * 2 - 1)) / (threads * 2);

        blocks = Math.min(maxNumBlocksX, blocks);

        return blocks;

    }
```

```
public int getNumThreads(int inputSize) {

       int threads = 0;

       threads = (inputSize < maxNumThreads*2) ? nextPow2((inputSize + 1)/ 2)
                   : maxNumThreads;

       return threads;

    }
```

Como una pequeña muestra de la diferencia de rendimiento obtenida por la implementación de la suma de vectores en JCuda frente a la versión canónica en Java, para dos vectores con contenido aleatorio de 1.000.000 de elementos float:

Tiempo reserva memoria y copia argumento 1 a GPU: 2049388 ms

Tiempo reserva memoria y copia argumento 2 a GPU: 2124962 ms

Tiempo reserva memoria argumento salida en GPU: 991338 ms

Tiempo ejecución y sincronización de threads en GPU: 2180944 ms

Tiempo copia de datos de GPU a Host: 1555352 ms

Tiempo computo sólo Java: 6450929

Realmente es de esperar que para un número pequeño de elementos y cuando la operación a computar es muy sencilla la ejecución lineal sea más eficiente. Básicamente el tiempo de gestión de memoria y de comunicación de datos entre la CPU y la GPU penaliza la ejecución CUDA, aunque la ejecución de la operación suma de vectores en sí misma sea 3 veces más rápida aún para una simple suma.

Realizando pruebas con mayores conjuntos de datos, de hasta 500 millones de elementos (previo ajuste a la memoria *heap* de Java), los resultados han sido similares, con ligera ventaja para la ejecución lineal, debido, de nuevo, a que la operación de cálculo suma es muy sencilla y se penaliza mucho la comunicación y sincronización CPU - GPU.

Para conseguir mejores tiempos en la ejecución CUDA para este tipo de operaciones tan sencillas se podría utilizar la librería JCublas, que es el *wrapper* de la librería CUDA CUBLAS que optimiza los cálculos con vectores y matrices, donde se consigue ventaja para conjuntos de datos de más de 10 millones de elementos.

Reducción de vectores

Esta operación se basa en computar todos los elementos de un vector para obtener un único resultado. Puede ser su suma, cálculo de medias, medianas, máximos, mínimos, etc.. Se ha implementado como operación de reducción la simple suma de los elementos de un vector, y se ha optimizado, en este caso, el uso de la memoria compartida para minimizar la comunicación entre CPU y GPU.

Se trata de reducir varios elementos por *thread*. El número se determinará por el número de bloques activos (según el tamaño de la entrada). Si se dispone de más bloques se asignarán menos threads por bloque para optimizar la carga de cada núcleo de la GPU. Cada thread pondrá el resultado de su suma en la memoria compartida, y después se irá reduciendo los resultados parciales de la memoria compartida.

Así, se puede considerar que se lograría una complejidad de log(n), además siendo ejecutada en paralelo. Los rendimientos comparados para distintos números de elementos frente a la versión Java del mismo algoritmo han sido:

Reducción de 100000 elementos

JCuda: 1.268ms, resultado: 50115.382813 (copia: 0.599ms, computación: 0.669ms)

Java : 2.619ms, resultado: 50115.378906

Reducción de 200000 elementos

JCuda: 1.078ms, resultado: 100113.539063 (copia: 0.633ms, computación: 0.445ms)

Java : 0.875ms, resultado: 100113.539063

Reducción de 400000 elementos

JCuda: 1.094ms, resultado: 199994.640625 (copia: 0.560ms, computación: 0.534ms)

Java : 1.663ms, resultado: 199994.640625

Reducción de 800000 elementos

JCuda: 1.554ms, resultado: 400348.468750 (copia: 0.997ms, computación: 0.557ms)

Java : 3.322ms, resultado: 400348.468750

Reducción de 1600000 elementos

JCuda: 2.480ms, resultado: 800442.500000 (copia: 1.827ms, computación: 0.653ms)

Java : 6.668ms, resultado: 800442.500000

Reducción de 3200000 elementos

JCuda: 4.163ms, resultado: 1600066.250000 (copia: 3.232ms, computación: 0.931ms)

Java : 13.357ms, resultado: 1600066.125000

Reducción de 6400000 elementos

JCuda: 7.415ms, resultado: 3199079.500000 (copia: 5.888ms, computación: 1.527ms)

Java : 26.657ms, resultado: 3199079.500000

Reducción de 12800000 elementos

JCuda: 13.827ms, resultado: 6399753.000000 (copia: 11.091ms, computación: 2.736ms)

Java : 53.402ms, resultado: 6399753.000000

Reducción de 25600000 elementos

JCuda: 26.879ms, resultado: 12800108.000000 (copia: 21.699ms, computación: 5.180ms)

Java : 106.463ms, resultado: 12800108.000000

Lógicamente en algunos decimales hay alguna mínima diferencia debido al redondeo en coma flotante.

Operaciones con Matrices

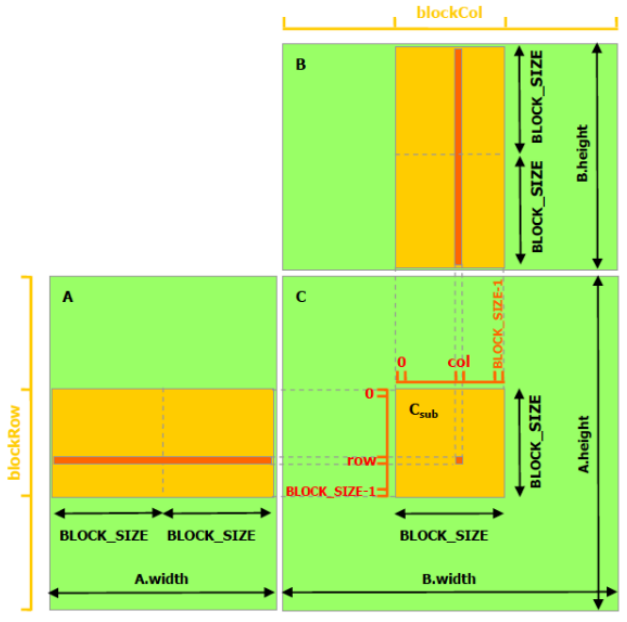
Las "*GPU Accelerated Libraries*", [Nvd13e] de NVIDIA facilitan en gran medida las operaciones con matrices, se han implementado dos versiones de la multiplicación de matrices, una utilizando JCublas y otra implementando directamente el algoritmo. Para la inversión de matrices se ha implementado directamente el algoritmo.

Multiplicación de Matrices

El algoritmo de multiplicación de matrices es el estándar definido por la regla matemática, siendo A y B las matrices a multiplicar y la matriz C el resultado, el elemento C[i,j] de la matriz producto se obtiene multiplicando cada elemento de la fila i de la matriz A por cada elemento de la columna j de la matriz B.

Para un mayor rendimiento hay que utilizar adecuadamente la *shared memory* como caché, con el patrón *tiling* comentado, sino, para cada thread que calcule una posición en la matriz de destino habría que enviar los datos de la fila-columna correspondiente cada vez. Y, como se ha comentado repetidamente, la comunicación CPU - GPU es el mayor cuello de botella de la computación CUDA. Lógicamente lo ideal sería poder enviar totalmente las matrices de entrada a la GPU, a la *shared memory*, pero desgraciadamente esta memoria suele ser escasa y no suficiente para grandes tamaños de argumento de entrada, que es, precisamente, donde destaca la computación paralela CUDA.

Para ello, y por simplificación del algoritmo se pueden equiparar las dos matrices argumento de entrada convirtiendo ambas matrices cuadradas del mismo tamaño. Así pues, para el cálculo se descompondrán las dos matrices en dos de tamaño *block_size * block_size*, o tamaño del block de que dispongamos.



Mapeo de memoria compartida fila - columna [Kdh12]

En el caso de la prueba el tamaño máximo es 1024. Así, como se muestra en el gráfico, la fila y las columnas correspondientes, al tener el mismo índice, se pasarán al mismo grupo de *threads* que deben computarlo, ya que ambas matrices tendrán la misma longitud.Básicamente se enviaría a memora la parte más a la izquierda de la primera matriz y la parte superior de la segunda.

Entonces se podrá calcular en la GPU el producto de los primeros 1024 bloques (*block_size)*, y sumarlos leyendo los productos de la memoria compartida. Dado que estas submatrices ya están en esta memoria cada *thread* de cada bloque que calcula el producto de las submatrices de 1024*1024 elementos puede calcular también la porción de su suma correspondiente directamente desde esa memoria compartida.

Una vez finalizado este cómputo para las primeras submatrices se cargarían en memoria el siguiente bloque de 1024*1024 elementos. Una vez todas las submatrices se han multiplicado ya tendríamos el resultado en el argumento de salida.

Hay un **hecho clave** que diferencia la multiplicación de matrices que se puede implementar en JCuda y la que se implementaría directamente en C/C++ usando CUDA. La diferencia es que **en Java no se pueden utilizar Arrays bidimensionales para representar las matrices**, por la limitación comentada en la implementación JNI en la que se basa JCuda, por lo que hay que representarlas como Arrays unidimensionales indicando el número de columnas para saber cuándo finaliza una fila y comienza la siguiente.

Como código de kernel se utilizó el mismo proporcionado por NVIDIA en su toolkit, *matrixMul_kernel.cu* implementando en JCuda la parametrización de los *tiling.*

Para la implementación sin CUBLAS se ha utilizado la clase KernelLauncher para definir los parámetros:

```
KernelLauncher kernelLauncher =
      KernelLauncher.create("matrixMul_kernel.cu", "multipM", false);

.... reserva de memoria y copia de argumentos a la GPU

kernelLauncher.setBlockSize(BLOCK_SIZE, BLOCK_SIZE, 1);
kernelLauncher.setGridSize(WC / BLOCK_SIZE, HC / BLOCK_SIZE);
kernelLauncher.setSharedMemSize(2*BLOCK_SIZE*BLOCK _SIZE*Sizeof.FLOAT);
kernelLauncher.call(Output, InputA, InputB, columnsA, columnsB);

... obtención de resultados y liberación de recursos en GPU.
```

La misma multiplicación se puede implementar de forma más sencilla llamando a **JCublas**, que ya optimiza tanto gestión de memoria como ejecución de la multiplicación, los comandos básicos serían:

```
status=cublasSetMatrix(HB,WB,sizeof(float),B,HB,BB,HB);

    /*KERNEL*/
        cublasSgemm('n','n',HA,WB,WA,1,AA,HA,BB,HB,0,CC,HC);....
cublasGetMatrix(HC,WC,sizeof(float),CC,HC,C,HC);
```

Inversión de Matrices

Para implementar esta operación en JCuda he utilizado directamente la librería JCublas.

Básicamente nos proporciona la funcionalidad básica de aritmética de matrices, así, la llamada a esta API sería:

```
public static void invertMatrix(int size, float matrix[])

{

Pointer pA = new Pointer();

cublasAlloc(size * size, Sizeof.FLOAT, dA);

cublasSetMatrix(size, size, Sizeof.FLOAT, Pointer.to(matrix), size, pA, size);

invertMatrix(size, pA);

cublasGetMatrix(size, size, Sizeof.FLOAT, pA, size, Pointer.to(matrix), size);

cublasFree(pA); }
```

Siendo la llamada invertMatrix la llamada conjunta al cálculo da factorización de la matriz, cálculo de los pivotes, y después la inversa de la matriz factorizada.

```
private static void invertMatrix(int n, Pointer dA)

          {

              // Factorización.

              int[] pivots = cudaSgetrfSquare(n, dA);

              // Inversión de la matriz factorizada

              cudaSgetri(n, dA, pivots);

          }
```

Reseñar que, para hacer uso de JCublas en un programa hay que inicializar el CUDA *driver* en modo de soporte CUBLAS, para ello JCuda proporciona las instrucciones: **cublasInit()** y **cublasShutdown()** que encapsulan las equivalentes del NVIDIA toolkit.

Generación de Números pseudo aleatorios JCurand

La generación de números aleatorios es bastante común para cierto tipo de algoritmos científicos y financieros. Muchas veces se precisa generar gran cantidad de datos aleatorios que después deben servir como argumento de entrada a computaciones paralelas CUDA. El realizarlo de forma tradicional, generando estos números en la CPU y después transfiriéndolos a la GPU consume gran ancho de banda, penalizando el rendimiento.

Por ello NVIDIA provee de la librería CURAND que permite la generación de estos números directamente en la GPU, además a gran velocidad pues pueden generarse a la vez por todos los núcleos de la GPU, almacenándolos ordenadamente en la memoria compartida. Incrementándose así la velocidad de generación y ahorrando ancho de banda en los cálculos.

Al igual que las otras librerías CUDA de NVIDIA, JCuda proporciona un *wrapper* a CURAND, JCurand, y su uso es sencillo. Así, en el ejemplo implementado, íntegramente, sería:

```
// Habilitar excepciones y omitir chequeo de errores
        JCuda.setExceptionsEnabled(true);
        JCurand.setExceptionsEnabled(true);
        int n = 1000000;
        curandGenerator generator = new curandGenerator();
        // Almacenar los n números float en host.
        float hostData[] = new float[n];
        // Reservar memoria para los n números en la GPU
        Pointer deviceData = new Pointer();
        cudaMalloc(deviceData, n * Sizeof.FLOAT);
        // Inicializar el generador de números pseudo-aleatorios
        curandCreateGenerator(generator, CURAND_RNG_PSEUDO_DEFAULT);
        // Establecer semilla
        curandSetPseudoRandomGeneratorSeed(generator, 1234);
        // Generar los n números en la GPU
        curandGenerateUniform(generator, deviceData, n);
        // Copiarlos a la CPU host
        cudaMemcpy(Pointer.to(hostData), deviceData,
            n * Sizeof.FLOAT, cudaMemcpyDeviceToHost);
```

```
// Mostrar el resultado
System.out.println(Arrays.toString(hostData));
// Liberar memoria de la GPU
curandDestroyGenerator(generator);
cudaFree(deviceData);
```

Los números generados en el array **deviceData** podrían utilizarse directamente para otras computaciones en la GPU sin necesitad de comunicación adicional entre la CPU host y la GPU.

Ejemplo de código C embebido en Java con KernelLauncher

Como se describió anteriormente, el código CUDA C del kernel no tiene porqué implementarse siempre en ficheros adicionales ***.cu*** a compilar, sino que JCuda permite insertar ese código dentro del código Java. La ventaja es de poder mantener todo el código dentro de código fuente Java, además la clase **KernelLauncher** permite indicar todos los parámetros de ejecución sus atributos, mediante los métodos de inicialización y sus *setters*.

El algoritmo inicial de suma de vectores, implementado directamente con KernelLauncher:

```
        JCudaDriver.setExceptionsEnabled(true);
// Código CUDA - C a ejecutar en el núcleo
        String sourceCode =
            "extern \"C\"" + "\n"
            "__global__ void add(float *result, float *a, float *b)" + "\n" +
            "{" + "\n" +
            "    int i = threadIdx.x;" + "\n" +
            "    result[i] = a[i] + b[i];" + "\n" +
            "}";
        // Inicializar el núcleo
        System.out.println("Compilando código del nucleo...");
        KernelLauncher kernelLauncher =
            KernelLauncher.compile(sourceCode, "add");
        // Generar datos de entrada
        System.out.println("Generando vectores de entrada...");
        int size = 100000;
        float result[] = new float[size];
        float a[] = new float[size];
        float b[] = new float[size];
        for (int i=0; i<size; i++)
        {
            a[i] = i;
            b[i] = i;
```

```
}

// Reservar memoria y copiar argumentos de entrada
System.out.println("Inicializando memoria de GPU...");
CUdeviceptr dResult = new CUdeviceptr();
cuMemAlloc(dResult, size * Sizeof.FLOAT);
CUdeviceptr dA = new CUdeviceptr();
cuMemAlloc(dA, size * Sizeof.FLOAT);
cuMemcpyHtoD(dA, Pointer.to(a), size * Sizeof.FLOAT);
CUdeviceptr dB = new CUdeviceptr();
cuMemAlloc(dB, size * Sizeof.FLOAT);
cuMemcpyHtoD(dB, Pointer.to(b), size * Sizeof.FLOAT);
// Ejecutar código de núcleo
System.out.println("Ejecutando en GPU...");
kernelLauncher.setBlockSize(size, 1, 1);
kernelLauncher.call(dResult, dA, dB);
// Copiar resultado a CPU
System.out.println("Recuperando resultados...");
cuMemcpyDtoH(Pointer.to(result), dResult, size * Sizeof.FLOAT);
System.out.println("Result: "+Arrays.toString(result));
// Liberar memoria de GPU
cuMemFree(dA);
cuMemFree(dB);
cuMemFree(dResult);
```

Realmente el manejo de KernelLauncher para parametrizar la ejecución del núcleo es bastante útil, pero no considero que incrustar el código CUDA-C sea adecuado, pues no todos los algoritmos del núcleo serán tan sencillos e impide compilarlos separadamente al código Java.

Cálculo binomial de precios de opciones

Los algoritmos de cálculo de precios de opciones, binomial y Black-Scholes se han tomado de los ejemplos proporcionados por NVIDIA en el CUDA Toolkit, con la gran documentación realizada por Victor Podlozhnyuk , "*Binomial Option Pricing Model*", [Nvd13b], complementada con bibliografía adicional sobre manejo de opciones, *"Fundamentals of Futures and Options Markets"* [Hull04]. Al igual que en el ejemplo proporcionado por NVIDIA, se calcularán precios para opciones de tipo Europeo, (que puede ejercitarse sólo al vencimiento).

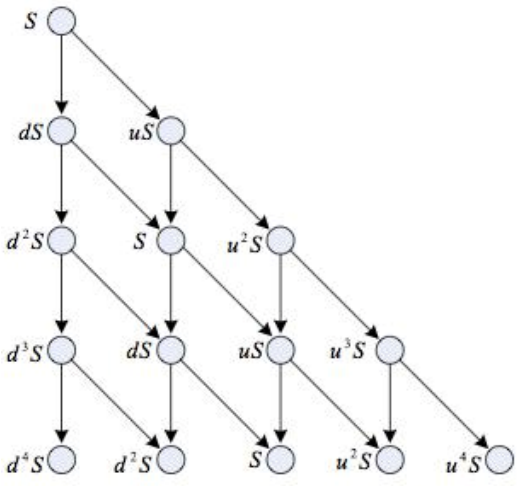
Sin embargo ha habido una importante adaptación debido a la limitación ya comentada de uso de objetos *structs* de C en JCuda por la limitación de JNI.

Esta problemática ya describió en detalle anteriormente en el apartado "*Argumentos de tipo estructurado*".

Por tanto, donde en el algoritmo original se usaban punteros a objetos TOption, que contiene los datos de precio del subyacente, precio de la opción, strike o vencimiento y la tasa de retorno *sin riesgo*, en la implementación JCuda ha habido que adaptarlo en 3 arrays diferentes, utilizando el índice como sincronizador entre los datos.

Básicamente el modelo binomial es una solución iterativa que simula la evolución del precio de la opción hasta su vencimiento para periodos equivalentes según la variación del precio del subyacente. Se le asigna a estas variaciones de precio unas probabilidades arbitrarias partiendo el día de hoy, como nodo inicial.

Así, cada nodo siguiente partirá de los calculados para el día anterior, generándose un árbol de posibilidades cada vez más grande hasta el vencimiento.



Árbol de dependencias algoritmo cálculo binomial precios opciones.

El aprovechamiento de la GPU en esta computación se da al ir asignando los sucesivos nuevos nodos hoja del árbol para cada día posterior a diferentes *threads* en la GPU para su computación en paralela, utilizando la memoria compartida para la comunicación entre *threads*.

La implementación realizada ha seguido directamente la existente en el NVIDIA toolkit, siendo la condición de terminación la fecha del vencimiento de la opción.

```
//Cálculo para memoria compartida
Int tid = threadIdx.x
for(int k = c_start - 1; k >= c_end;){
// Inicio del cálculo de precio de la Call
__syncthreads();
if(tid <= k)
callB[tid] = puByDf * callA[tid + 1] + pdByDf * callA[tid];
k--;
// Fin del cálculo
__syncthreads();
if(tid <= k)
callA[tid] = puByDf * callB[tid + 1] + pdByDf * callB[tid];
k--;
}
```

Cálculo de precio de opciones por Black-Scholes

Se ha partido también de los ejemplos proporcionados por NVIDIA y del trabajo de Podlozhnyuk y Hull. Así, el valor de cada día al vencimiento de una opción Call será según la fórmula:

$$V_{call} = S \cdot CND(d_1) - X \cdot e^{-rT} \cdot CND(d_2)$$

Siendo V el precio de la opción al vencimiento (tipo europeo), S el precio actual de la opción, CND(d) las funciones de distribución normal que simulan el movimiento del precio del subyacente:

$$d_1 = \frac{\log(\frac{S}{X}) + (r + \frac{v^2}{2})T}{v\sqrt{T}} \quad \text{y} \quad d_2 = \frac{\log(\frac{S}{X}) + (r - \frac{v^2}{2})T}{v\sqrt{T}}$$

X el precio del Strike, T el tiempo que falta hasta la expiración, r la tasa de interés sin riesgo, y v la volatilidad del subyacente.

La computación del algoritmo ya venía proporcionada por el NVIDIA toolkit, ya que la parte de codificación en CUDA-C es similar, salvo el uso de varios Arrays en vez de un objeto C *struct* TOption. Una vez mas, el verdadero reto de la elaboración de programas en CUDA es el mapeo entre las estructuras de datos y los *threads* y bloques para conseguir, a la vez, una mínima carga de comunicación entre la CPU y la GPU, y una máxima *ocupación* de los núcleos CUDA de la GPU.

Así, la asignación básica más sencilla sería asignar a cada *thread* un único índice de cada uno de los arrays argumentos de entrada, y sería posible si el conjunto de datos de entrada fuese menor que el número de threads disponibles en la GPU, en el caso de prueba serían 1024 *threads* * 65536 bloques = 67.107.840 elementos, que es una cantidad importante, pero el algoritmo debe ser más general, capaz de repartir n elementos entre cada thread para mayores cantidades de elementos a computar. Así habrá una función BlackScholesSingleOption que calculará el valor de una opción concreta y que será llamada por otra función BlackScholes que reparte la entrada, el número todal de opciones a computar, entre los threads disponibles en la GPU, sincronizados en memoria compartida:

```
int tid = blockDim.x * blockIdx.x + threadIdx.x; int numThreads = blockDim.x
* gridDim.x;

    for(int index = tid; index < NumOptions; index += numThreads)

      BlackScholesSingleComputation(d_CallResult[index], d_PutResult[index],
      d_StockPrice[index], d_OptionStrike[index],d_OptionYears[index],
      riskfree, volatility );
```

La función `BlackScholesSingleComputation` realiza el cálculo de las fórmulas Black-Scholes para cada opción individual.

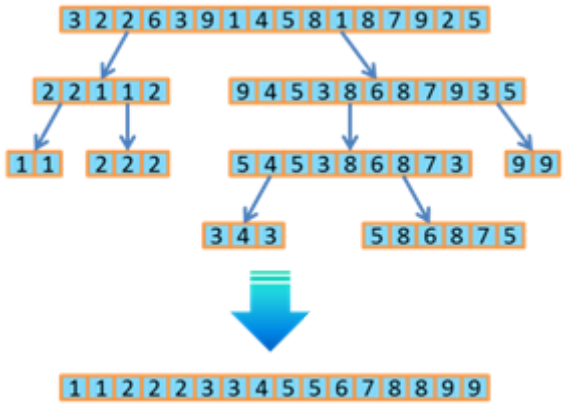
QuickSort

Al igual que para las operaciones aritméticas con matrices, también ha disponibles potentes librerías CUDA que podemos utilizar para la ordenación, quizás de las más interesantes las GPU Quicksort Library, de Distributed Computing and Systems, [DiCo08]. Pero para el proyecto se ha optado por implementar el algoritmo.

El mayor reto ha sido el implementar un óptimo uso de la memoria compartida para minimizar accesos y maximizar la ocupación de la GPU. Según la documentación consultada se deben aplicar distintas optimizaciones según la distribución de los datos a ordenar, según sean una distribución normal, gaussiana, Bucket, ya ordenada o distribución cero. Pero el algoritmo básico no cambia, especificando los pivotes de quicksort según el bloque y el *thread* de ejecución:

```
// Código de kernel

int idx =  threadIdx.x + blockIdx.x * blockDim.x;
int start[MAX_LEVELS];
int end[MAX_LEVELS];

start[idx] = idx;
end[idx] = N - 1;
```

Conforme se van ordenando tramos mayores del conjunto de datos habrá menos threads en ejecución, hasta al final encontrar todo el conjunto de datos de salida ordenados en memoria compartida.



Cada *thread* deposita en memoria compartida su tramo de vector ordenado.

Conclusiones trabajando con JCuda

Desarrollar con JCuda, en el apartado relativo al desarrollo para GPU es muy parecido al desarrollo que se haría en C o C++ con el NVIDIA tookit.

El tener que desarrollar el código del núcleo en CUDA-C implica que el o los desarrolladores deben tener conocimientos del mismo, si en una empresa o equipo de desarrollo de varias personas se podrían tener roles específicos.

Aún así, el disponer de la posibilidad de interactuar con las GPU desde Java de forma tan transparente es muy interesante. Cabe destacar, desde mi punto de vista, sobre todo, el soporte a las *Accelerated Libraries*, que es lo que realmente hace destacar a JCuda frente a la otra alternativa probada. Esto me parece clave para la adopción de JCuda en desarrollos en proyectos reales de producción.

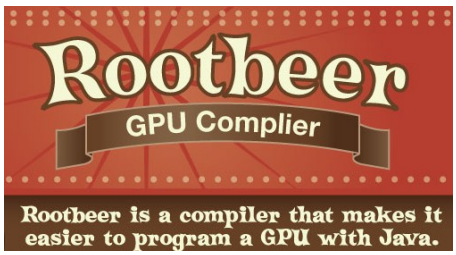
De forma práctica, creo preferible separar el código Java y CUDA C en diferentes proyectos, que puedan se probados por separado, y preparar el código Java para utilizar ya los ficheros *ptx* o *cubin* ya compilados para evitar el soporte a la precompilación en cada proyecto.

Es interesante aprovechar el soporte Java para multithreading para, una vez enviada una tarea de cálculo intensivo a la GPU, aprovechar otro thread Java para continuar con la ejecución de otras tareas, hasta obtener los resultados de la GPU.

Como puntos débiles, los habituales en toda computación CUDA, el manejo crítico de la memoria y de la ocupación de los núcleos y, sobre todo, las limitaciones en el uso de arrays multidimensionales u objetos complejos, *structs*.

Rootbeer

Código de núcleo en Java



Esta librería es otro acercamiento al desarrollo de software optimizado para la arquitectura CUDA de GPUs en Java. Aunque todavía carece del gran conjunto de librerías y bindings que proporciona JCUDA, sí aporta algo de lo que JCUDA carece, y es la posibilidad de programar el código del núcleo, el que se ejecuta en cada uno de los *cores* de la GPU, en lenguaje Java, sin tener que recurrir a C o C++ como en el caso de JCUDA (y en PyCuda para la integración con Python).

Si bien no está aún muy extendido, desde mediados de 2012, el proyecto *Root Beer* permite poder programar código a ejecutar dentro de los núcleos CUDA de la GPU en lenguaje Java.

Con éstas librerías (y las herramientas auxiliares), se puede desarrollar código para los núcleos en Java, utilizando objetos Java normalmente, incluyendo métodos estáticos y de instancia, excepciones, etc..., y automáticamente realizará las conversiones necesarias para generar el código PTX de núcleo necesario, así como para serializar y enviar (realizar la allocation correcta) al a memoria de la GPU los objetos para su ejecución en paralelo.

La generación de código CUDA C desde Java, y su compilación, puede realizarse independientemente de la complación del programa Java, tanto en tiempo de compilación o en tiempo de ejecución. Para ello se aconseja, en proyectos reales, separar el código en dos proyectos, la parte que se ejecutará en la CPU y en otro la parte paralelizada. La cual se puede probar por separado, generando el código CUDA-C y generando los ejecutables *ptx.*

Salvo algunas limitaciones como el uso de Reflection e invocación dinámica de métodos, así como la liberación de memoria automática (Garbage collector), creo que se trata de una iniciativa extremadamente interesante para poder llegar a una programación CUDA en lenguaje Java al 100%.

Hay que tener en cuenta que esta librería no paraleliza automáticamente el código, sino que sólo lo transforma y realiza de forma transparente la serialización de los objetos Java y su comunicación a las GPU, sin embargo hay que especificar mediante programación qué parte del código y sobre qué conjunto de datos se va a ejecutar en cada núcleo de la o las GPUs.

Root Beer sigue precisando aún del entorno de desarrollo CUDA de NVIDIA para generar, aunque sea dinámicamente, los ejecutables PTX, a partir del código programado en Java para el núcleo, pero consigue aún así un alto grado de optimización del código generado

Un sencillo ejemplo de código de núcleo en Java, para ser después compilado/traducido por Root Beer sería:

```
public class DoubleArrayMult implements Kernel {

  private int[] m_source;
```

```
  private int m_index;


  public DoubleArrayMult(int[] source, int index){
    m_source = source;
    m_index = index;
  }
public void gpuMethod(){
    m_source[m_index] *= 2;
  }
}
```

Para invocar a este código ejemplo, desde un proyecto Java utilizando Root Beer, el código fuente sería:

```
public void multArray(int[] array){
    List<Kernel> jobs = new ArrayList<Kernel>();
    for(int i = 0; i < array.length; ++i){
      jobs.add(new DoubleArrayMult (array, i));
    }
    Rootbeer rootbeer = new Rootbeer();
    rootbeer.runAll(jobs);
  }
```

Los patrones de acceso a memoria descritos para JCuda son totalmente aplicables en Rootbeer. Para el patrón en tesela, *tiling*, tan sólo es preciso definir el tamaño de la tesela y realizar de forma similar la asignación de memoria. Se definirán el número de *threads,* llamados *jobs* en terminología rootbeer, y el tamaño del array de argumento de entrada, ambos valores determinarán el tamaño de la tesela (*tile*).

Para los arrays de varias dimensiones en rootbeer, si bien están soportados, por motivos de rendimientos es preferible aconseja evitarlos, optando por separarlos en distintos argumentos de tipo array unidimensional.

Para los tipos de datos complejos, *strutcs*, rootbeer opta por tipos referencia.

La única carencia reseñable en la actualidad es la de colecciones típicas Java, List, Set y Maps, debiendo siempre convertirse a arrays. Se espera que estén soportadas en el futuro.

Resumiendo las características de Rootber:

Soporte Java para generación de CUDA C

- Tipos referencia compuestos
- Métodos de instancia y estáticos
- Arrays uni o multidimensionales de cualquier tipo!! Parcialmente pero operativo en algunso casos.
- Soporte a la palabra clave synchornize
- Soporte a System.out.println, printf en C

Características CUDA:

- Soporte de las variables de código de núcleo:
  - threadIdx.x
  - blockIdx.x
  - blockDim.x
  - __syncthreads
- Soporte a getión de memoria compartida

En el futuro se prevée añadir soporte a la conversión de colecciones Java

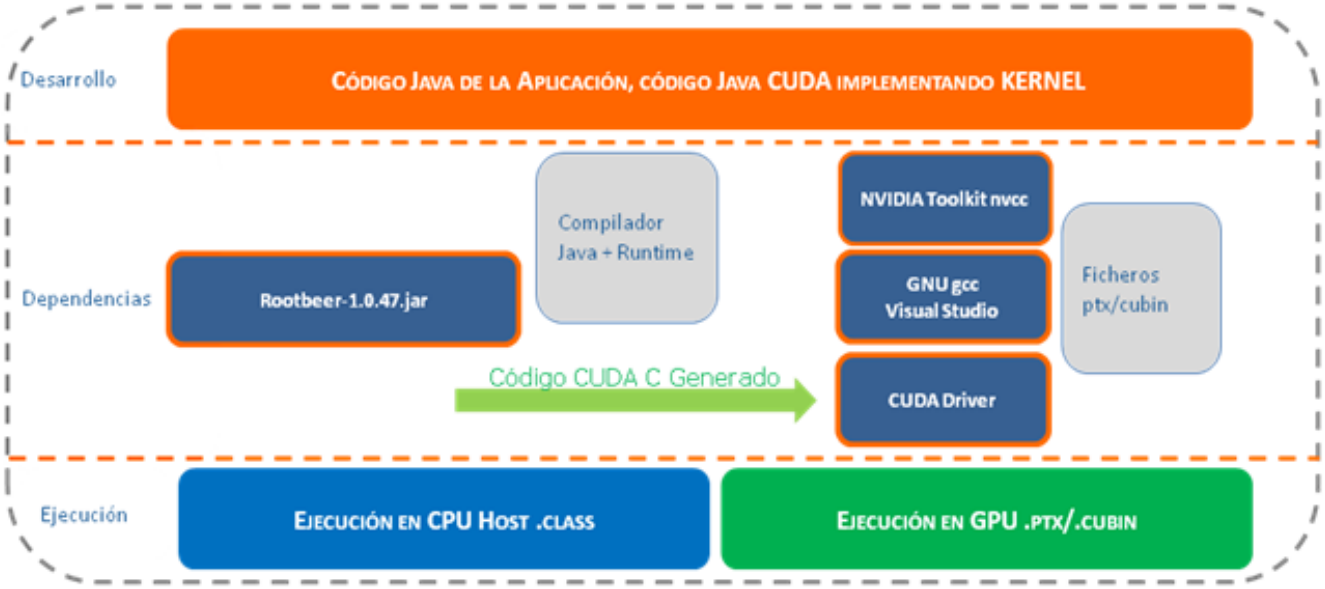
- LinkedList/ArrayList
- HashSet/TreeSet/LinkedHashSet
- HashMap/TreeMap/LinkedHashMap

Plantillas para código de núcleo

Implementa la una serialización cuasi nativa desde la CPU a la GPU para los patrones de envio de parámetros más comunes, Arrays, Matrices, escalares, siendo más rápido que la comunicación codificada programáticamente. Además simplifica el desarrollo de código de núcleo en la gran mayoría de casos.

Componentes y dependencias de un programa RootBeer

Los componentes y dependencias de un proyecto Rootbeer son los siguientes:



Librerías y herramientas para Compilación y Ejecución Rootbeer

Como se puede verificar, el entorno de desarrollo y ejecución es muy similar a JCuda, presentando la ventaja de no tener que implementar nada en CUDA-C, sino solamente en Java con mínimas adaptaciones y uso de la librería Rootbeer. Durante la ejecución, Rootbeer traduce automáticamente el código de las clases que extienden la clase Kernel a pseudocódigo CUDA-C que es compilado a formato ptx y enviado a ejecutarse a la GPU.

Para cálculos masivos el tiempo de generación de código de núcleo y su compilación es insignificante frente a la ganancia que supone la ejecución en paralelo.

Como desventaja, no dispone de *wrappers* para las *Accelerated Libraries*, CUBLAS, CURAND, etc…, con lo que está en gran desventaja para desarrollos basados en cálculos comunes que son facilitados en gran media con su uso.

Algoritmos implementados en RootBeer

El mayor reto en la implementación de algoritmos CUDA con RootBeer es el mismo que el hacerlo en cualquier otro lenguaje, y es el manejo óptimo de la memoria compartida como caché entre la GPU y la CPU para minimizar la comunicación entre ambas. La diferencia más significativa es que el código del núcleo se implementará directamente en Java, siendo el mayor inconveniente la peor integración, casi nula, con las librerías NVIDIA CUDA, Cublas, Curand, etc...

Multiplicación de matrices

Al igual que en JCuda, reseñar que las matrices se representan por Arrays unidimensionales, indicando el número de columnas de la matriz. El código de inicialización de bloques y *threads* sería similar al del algoritmo con patrón *tiling* de JCuda, variando la parte de ejecución en el núcleo al ser ésta en Java.

```
public void run(){
   int num_each = blockSize / numCores;
   int start_row = index * num_each;
   int stop_row = (index + 1) * num_each;
   if(index == numCores - 1){
     stop_row = blockSize;
   }
   int b_columns = blockSize * gridSize;
   int a_columns = blockSize;
   for(int i = start_row; i < stop_row; ++i){
     for(int j = 0; j < b_columns; ++j){
       float sum = 0;
       int dest_index = i*b_columns+j;
       for(int k = 0; k < a_columns; ++k){
         int a_src = i*a_columns+k;
         int b_src;
           b_src = j*a_columns+k;
         float a_value = m_a[a_src];
         float b_value = m_b[b_src];
         sum += a_value * b_value;
       }
       c[dest_index] = sum;
     }
   }
 }
```

El equivalente a __syncthreads() en rootbeer es m_thread.join();

Generación de números aleatorios

Rootbeer ha mapeado directamente la clase Random de Java a la equivalente de CURAND dentro la GPU. Así, cuando se hace referencia a un objeto Random en el código Java, si éste se encuentra dentro de una clase que implementa el interfaz **Kernel**, y por lo tanto va a ser convertida en ejecutable CUDA, la librería automáticamente convertirá el tipo de ese número a tipo CURAND. Así, un código que maneje números aleatorios en la GPU será casi indistinguible a otro que los maneje en código Java normal:

```
public import java.util.List;
import java.util.Random;
import java.util.ArrayList;
import edu.syr.pcpratts.rootbeer.runtime.Kernel;

 class RandomRemap implements Kernel {

  private Random m_random;
  private int m_randomNumber;

  public RandomRemap(Random random){
    m_random = random;
  }

  public void gpuMethod(){
    m_randomNumber = m_random.nextInt();
  }

  public int getRandomNumber(){
    return m_randomNumber;
  }
}
```

Quicksort

Conforme a la codificación CUDA, el algoritmo de ordenación Quicksort que se ejecutará en el núcleo sería casi indistinguible del código Quicksort Java normal, con la salvedad del manejo de bloques y *threads,* y que la recursión es reemplazada por la iteración manteniendo la memoria compartida como pila de ejecución*.*

Así, éste sería una implementación de Java estándar:

```
public void quicksort(int array[], int start, int end){
   int i = start;
   int k = end;

   if (end - start >= 1
     int pivot = array[start
     while (k > i){
      while (array[i] <= pivot && i <= end && k > i){
        i++;
      }
      while (array[k] > pivot && k >= start && k >= i){
        k--;

      }
      if (k > i
        swap(array, i, k);
      }
     }
     swap(array, start, k);
     quicksort(array, start, k - 1);
     quicksort(array, k + 1, end);
   } else {
     return;
   }
 }
```

Y ésta la implementación del código de kernel de quicksort con rootbeer:

```
public void gpuMethod() {

/* Única diferencia, establecimiento de pivotes según los bloques y threads
de ejecución */
   int pivot, L, R;
    start[index] = index;
    end[index] = size - 1;

    while (index >= 0) {  // Iteración para simular la recursión.

      L = start[index];
      R = end[index];
      if (L < R) {
        pivot = values[L];
        while (L < R) {
          while (values[R] >= pivot && L < R)
            R--;
          if(L < R)
            values[L++] = values[R];
          while (values[L] < pivot && L < R)
            L++;
          if (L < R)
            values[R--] = values[L];
        }
        values[L] = pivot;
        start[index + 1] = L + 1;
        end[index + 1] = end[index];
        end[index++] = L;
  if (end[index] - start[index] > end[index - 1] - start[index - 1]) {
          // swap start[idx] and start[idx-1]
          int tmp = start[index];
          start[index] = start[index - 1];
          start[index - 1] = tmp;
          tmp = end[index];
          end[index] = end[index - 1];
          end[index - 1] = tmp;
        }

      } else {
        index--;
      }
    }
  }
```

El rendimiento obtenido, para un vector aleatorio, de 64 elementos ha sido de un speedup aproximadamente x4 en las pruebas realizadas.

Resultado en la GPU: 447
Resultado en la CPU: 1411

Conclusiones trabajando con RootBeer

Como se ha comentado, Rootbeer realiza la generación del código CUDA-C desde java y después lo compila para obtener el ejecutable, *ptx* que se enviará a la GPU. Por motivos prácticos es conveniente automatizar este proceso con un *script*, bien con ant, maven o herramienta similar.

También es, por tanto, conveniente, separar la implementación que se ejecutará en CUDA en un proyecto paralelo, pues acelerará la compilación en caso de proyectos grandes, sin interferiren la parte secuencial de la aplicación. La generación de código CUDA C puede tardar algo de tiempo en el caso de programas complejos.

Hay que vigilar el tamaño de las operaciones que se ejecuten en el núcleo y su tiempo de ejecución, pues a veces pueden provocar *timeout* y fallar en la sincronización de *threads*. Si bien esto es dependiente de la capacidad de la GPU donde se ejecute.

Por tanto hay también que tener ciertos conocimientos de la arquitectura CUDA y de la adecuada gestión de memoria y distribución de *jobs* en los bloques y *threads*.

Otras consideraciones son genéricas al desarrollo CUDA, una gestión adecuada de la memoria compartida como caché es crucial, así como repartir la carga adecuadamente entre los núcleos de la GPU para evitar tener muchos inactivos mientras otros realizan muchas computaciones.

Aún así, personalmente, me ha resultado muy interesante el manejo de RootBeer, que ha convertido Java en una especie de DSL para generar código CUDA-C, y que se nota que va mejorando en sus sucesivas versiones.

Como nota negativa, la carencia de soporte de las librerías CUBLAS, CURAND, etc.., si bien nada impide poder utilizar, en un mismo proyecto, Rootbeer junto con JCuda para casos especiales.

Evaluación del desarrollo CUDA en Java

JCUDA

La ventaja más evidente, común también a *rootbeer*, es que se puede integrar directamente los cálculos con CUDA en un proyecto realizado en Java, con una mínima integración. Además JCuda encapsula como librerías auxiliares encapsulación para las librerías de NVIDIA llamadas **GPU Accelerated Libraries**, [Nvd13e], que incorporan mucha funcionalidad de la API CUDA encapsulada, como el soporte a operaciones de matrices con JCublas, JCufft para transformadas de Fourier, o JCusparse para matrices disperas.

En el caso de JCusparse, por ejemplo, ya da preparado/optimizado, una gestión de memoria muy estudiada para ese tipo de matrices, conforme a las mejores prácticas recomendadas por NVIDIA, y conversión de formatos, etc...

Sin embargo lo más destacable de estas librerías no es sólo que implementen funcionalidad muy utilizada en el cálculo científico, financiero o analítico en general sino, sobre todo, que optimizan la gestión de memoria y de transferencia de datos entre el ordenados *host* y la GPU, ya descrito anteriormente como *Tiling Pattern*, así como la optimizada distribución de los argumentos de entrada en los adecuados *blocks* y *threads* para sí optimizar el aprovechamiento de los *cores* paralelos, optimizando el rendimiento.

En suma, JCuda se puede decir que básicamente ahorra y/o encapsula lo que sería necesario implementar con llamadas JNA (Java Native Access) implementadas manualmente, y también proporciona cierto "mapeo" de tipos básicos java (int, float, etc..), y esos tipos en C++. La distribución de los objetos en memoria debe hacerse siguiendo formas parecidas a con C++, para distribuir la carga entre kernels (blocks y threads).

RootBeer

Esta librería tiene la ventaja de que el código del kernel se hace en java, y ése se compila de Java a una especie de C automáticamente, y de ahí a código ptx o cubin para ejecutar en la GPU.

Sin embargo el tema del manejo de la memoria es distinto. No permite la "flexibilidad" de JCuda, de asignación de memoria. Sino que la clase que ejecuta el código del kernel debe extender una clase llamada Kernel, que ellos llaman "plantilla de Kernel":

Entonces el código es java puro, aunque algo menos flexible que con JCuda, en algunos casos no se puede optimizar tanto la configuración de la memoria para la ejecución, pero es todo puro Java, y se encargan de toda la gestión (alloc, free) de memoria. No hay que reservar "explícitamente" memoria o liberarla y según pruebas de rendimiento el código CUDA, ptx o cubin generado es de buena calidad.

Desarrollo CUDA en lenguaje Python

El principal soporte a programación CUDA en Python, la librería **PyCuda**, surgió por la iniciativa del matemático Andreas Klöckner en 2009. Estas librerías son código gratuito y libre, open source, utilizando la licencia MIT/X, y presentan una funcionalidad en cierta forma parecida a JCuda, aunque quizás algo más integrada tanto con el toolkit nativo CUDA de NVIDIA como con las propias librerías numéricas de Python, como Numpy, dándole una gran potencia.

Como contrapunto a JCuda, la última versión disponible de PyCuda hasta la fecha sólo soporta una versión relativamente antigua del CUDA tookit de NVIDIA, la 3.2, frente a la 5.0 utilizada en las pruebas con JCuda y Rootbeer, y además, en entornos windows, está limitado al compilador MS Visual Studio 8, no soportando versiones más recientes. Del mismo modo, para poder compilar ejecutables CUDA en Windows, precisa de la librería Boost versión 1.38.

En entornos Linux las limitaciones son similares, salvo que se soporta el compilador **gcc** hasta la versión 4.2 y, a partir de la versión 0.94 de PyCuda tampoco es necesaria la librería Boost.

Otra posibilidad disponible para trabajar con GPUs CUDA en lenguaje Python sería el producto **Anaconda Accelerate** de Continuum Analytics, [CoAn13a]. En concreto se refiere al compilador NumbaPro, capaz compilar para distintas GPUs directamente desde sintaxis Python, al estilo *RootBeer* con Java, y directamente mapea en GPUs CUDA lo que serían llamadas matriciales a librerías NumPy, también realiza sincronización de threads y gestión de memoria compartida de forma automática, (*tiling*), tan crítico para obtener buen rendimiento en ejecuciones paralelas en CUDA al optimizar el tráfico entre el ordenador host y la GPU, además de otras características propias de Python, directamente mapeadas a CUDA, como el uso de bucles multi-threaded o el uso de *closures*.

Sin embargo ésta es una herramienta comercial, que, aunque cuenta con el respaldo oficial de NVIDIA, [Nvd13d], durante la elaboración de este Proyecto Fin de Carrera se ha probado sólo parcialmente su funcionamiento en ejemplos sencillos para comprobar su utilidad con gran satisfacción. Por ello se reseñará como uno de los posibles trabajos futuros de ampliación del proyecto, pues su instalación es significativamente más sencilla que PyCuda y su manejo algo diferente.

Finalmente comentar otra posibilidad para aprovechar una GPU CUDA programando en Python, como es el proyecto CUDAMat, [CuMa10], que es una librería que implementa una clase matriz con operaciones optimizadas para aprovechar la arquitectura CUDA. Es un proyecto *open source* alojado en el repositorio público ofrecido por google, pero aparentemente no ha tenido nuevas versiones desde 2010 y tampoco ha sido probado en este trabajo.

PyCUDA

Esta librería funciona, en Python, de forma similar a JCuda en Java, pudiendo incluir el código ejecutable del kernel tanto en fichero adicionales a los de código fuente Python como código CUDA C embebido dentro del código Python como plantillas, para ser compilado, si no lo está ya, y ejecutado en tiempo de ejecución del programa. Por ejemplo, el código para la plantilla para *Reducción* de vectores para mapear un producto escalar sería:

```
from pycuda.reduction import ReductionKernel
dot = ReductionKernel(dtype out=numpy.oat32, neutral="0",
reduce expr="a+b", map expr="x[i]_y[i]",
arguments="const oat _x, const oat _y")
```

También encapsula, de forma similar a JCuda, las librerías adicionales del kit CUDA de NVIDIA, como CUBLAS, CURAND, etc.., pudiendo hacerse llamadas directamente a éstas.

Continuando con el ejemplo de código anterior, utilizando la librería CURAND para la generación de números aleatorios desde Python, para las matrices a multiplicar:

```
from pycuda.curandom import rand as curand
x = curand((1000_1000), dtype=numpy.oat32)
y = curand((1000_1000), dtype=numpy.oat32)
```

Finalmente ejecutamos la operación:

```
x dot y = dot(x, y). get()
x dot y cpu = numpy.dot(x.get(), y.get ())
```

Un punto muy interesante de esta integración Python - CUDA, es la integración con Numpy, la avanzada librería Python de manejo de matrices numéricas, integrando éstas directamente en la estructura *matricial* inherente a los procesadores vectoriales de la GPU.

Para los algoritmos implementados y probados con PyCuda se ha utilizado la versión 2.7.3 de la versión oficial de Python, y la versión llamada 2012.1 de PyCuda. Como IDE se ha utilizado PyCharm de JetBrains.

Algoritmos Implementados en PyCuda

Operaciones con Vectores y matrices

Este tipo de operaciones en PyCuda son extremadamente sencillas. Obviamente se podrían implementar de nuevo siguiendo unos algoritmos similares a los ya utilizados en JCuda, pero una de las grandes fortalezas de PyCuda es su integracion con Numpi, pudiendo paralelizar en CUDA directamente operaciones con todo tipo de matrices.

Así, una multiplicación de matrices sería tan trivial como:

```
import pycuda.gpuarray as gpuarray
import pycuda.driver as cuda
import pycuda.autoinit
import numpy
import time

 # Siendo a y b los array argumento de entrada
.
a_gpu = gpuarray.to_gpu(a)
b_gpu = gpuarray.to_gpu(b)
resultado = gpuarray.dot(a_gpu,b_gpu)
```

Básicamente se puede utilizar la función .dot, equivalente a la misma en numpy pero con implementación CUDA. Hay una gran documentación al respecto de soporte de gpuarray en la web de PyCuda.

Estos algoritmos también han sido parcialmente implementados en PyCuda, pero no complemtamente por motivos de calendario. Sin embargo se puede comprobar su implementación Python en la siguiente herramienta analizada, Anaconda Accelerate.

Cálculo de generación de precios aleatoria con MonteCarlo

QuickSort

Conclusiones trabajando con PyCuda

En un principio me costó más de lo esperado la instalación de PyCuda. Sobre todo por la dependencia con versiones anteriores a las previamente instaladas, utilizadas con JCuda y Rootbeer tanto del CUDA toolkit como de Visual Studio.

Las previamente instaladas eran Cuda toolkit 5.0 y Visual Studio 2010. Sin embargo la última versión disponible de PyCuda, la 5, 2012.1, precisa la versión 3.5 del NVIDIA toolkit y, como máximo, Visual Studio 2008.

Además precisa de las librerías C++ Boost, que ha habido que descargar y precompilar con Visual Studio.

Lograr tener todo listo me costó alrededor de casi 8 horas más de trabajo, entre documentación pruebas y ajustes, lo que en horario asignado al proyecto supuso casi 4 días o, aproximadamente, algo menos de 1 semana.

Aún así he de reconocer que mereció la pena. El soporte a Arrays directamente mapeados a memoria es algo tremendamente interesante y, sobre todo, la implementación CUDA de las muchas funciones de numpi, la librería más popular en el entorno Python para operaciones con matrices.

También reseñar que la integración con el CUDA Toolkit es más avanzado que en el caso de Java, ya que soporta directamente el interfaz C, pudiendo manejar adecuadamente tanto matrices multidimensionales como *struts*.

En suma, sería recomendable una vez superado el proceso de instalación, logrando ventaja ante Java por su mayor integración con C.

Sólo reseñar que, a diferencia de la herramienta analizada a continuación, aún precisa en muchos casos de desarrollo de código de núcleo en CUDA C, cuando no se utilicen librerías numpi o wrappers de las *Accelerated Libraries*. Sin embargo hay una iniciativa en PyCuda para poder implementar paulatinamente este código del núcleo también en Python, sin tener que recurrir a CUDA-C, y es a través de la traducción de ése código Python a CUDA -C automáticamente, mediante técnicas de *metaprogramación*, y ya hay algunos ejemplos de ello en la propia web principal de PyCuda. Pero por su poca popularidad hasta el momento y falta de tiempo no ha sido posible incluirla en este estudio.

Anaconda Accelerate

Este producto, como se indicó, es comercial y se considera como un complemento, *plugin*, de la distribución Python Anaconda, de Continuum Analytics, [CoAn13a]. Esto presenta la ventaja de una gran integración pero también la desventaja de que no se integra con la distribución de referencia del lenguaje, sino una propietaria, aunque de distribución gratuita.

En cualquier caso la distribución de Python anaconda está muy optimizada para computación de gran cantidad de datos, análisis científico y estadístico, con lo que se complementan magníficamente. Además reseñar que el precio *Anaconda Accelerate*, el *plugin* CUDA es, a junio de 2013, de 129$, siendo gratuita para uso académico.

La clave de *Anaconda Accelerate* sería el componente llamado **CUDA JIT**, que es el encargado de traducir las funciones Python a código ptx, el pseudo ensamblador que se ejecuta en las GPU, (además del formato binario cubin). El código ptx se envía a la GPU a través de las librerías CUDA de NVIDIA, con las que interactúa y ahí es ejecutado.

Así, un programa Python - CUDA con este compilador sería similar al implementado con PyCuda, con la salvedad del uso de algunas anotaciones para el código del *kernel*, que no necesitaría estar expresado en C / C++, sino en Python, y que el compilador traduciría, en tiempo de ejecución, a formato *ptx*. Por tanto el funcionamiento sería al que tiene *rootbeer* en Java. Por ejemplo, una simple suma de vectores:

```
from numbapro import cuda, jit, float32
import numpy as np
```

Éste sería el código del kernel:

```
@jit(argtypes=[float32[:],float32[:], float32[:]], target='gpu')
def cuda_sum(a, b, c):
        tid = cuda.threadIdx.x
        blkid = cuda.blockIdx.x
        blkdim = cuda.blockDim.x
        i = tid + blkid * blkdim
        c[i] = a[i] + b[i]
```

CUDA JIT es capaz de reconocer las variables CUDA threadIdx, blockIdx, blockDim, gridDim. Éstas están definidas en el módulo *numbapro.cuda*.

Como las funciones que se ejecutan en núcleos CUDA no devuelven ningún valor, sino que lo hacen a través de uno o más argumentos de la función, las funciones Python con la anotación *@jit*, si el target es *gpu* tampoco lo devolveran. Para ejecutar una función de kernel hay que proporcionar los valores de *grid* (threads) y *block*, por defecto son 1. Así, continuando con el código anterior, configuramos estos falores a 16 bloques y 32 threads por bloque:

```
griddim = 16, 1
blockdim = 32, 1, 1
cuda_sum_configured = cuda_sum[griddim, blockdim]
```

Y se hace la llamada, con argumentos generados aleatoriamente con NumPy:

```
a = np.array(np.random.random(512), dtype=np.float32)
b = np.array(np.random.random(512), dtype=np.float32)
c = np.empty_like(a)
cuda_sum_configured(a, b, c)
```

La llamada se puede hacer todo en uno, indicando directamente los parámetros al estilo de las llamadas en CUDA C, <<<griddim, blockdim>>>(…).:

```
cuda_sum[griddim, blockdim](a, b, c)
```

Para la comunicación entre la CPU *host* y la GPU se utilizan las funciones:

```
device_array = cuda.to_device(array)
```

Y la inversa,

```
device_array.to_host()
```

Esta instrucción el contenido de la memoria de la GPU a un buffer de arrays en el host. el objeto Python device_array es de una clase llamada DeviceNDArray, subclase de numpy.ndarray. Básicamente este objeto sería un mapa de la memoria de la GPU, y cuando este objeto es *liberado* entonces la memoria de la GPU es liberada, como cuando en JCuda se ejecuta un JCuda.cudafree().

Para la coordinación de la ejecución entre núcleos de la GPU y evitar que unos núcleos respondan antes que otros produciendo efectos indeseados, la ejecución en la GPU se organiza en *streams*. Así, una llamada normal a la GPU se iniciará configurando un *stream*, enviando los argumentos a la función en el kernel, la ejecución, la obtención de datos y la sincronización, con la instrucción stream.synchronize() para asegurar que todas las ejecuciones han finalizado, sería el equivalente a *__syncthreads() en C/C++*:

```
stream = cuda.stream()
devary = cuda.to_device(an_array, stream=stream)
a_cuda_kernel[griddim, blockdim, stream](devary)
cuda.to_host(devary, stream=stream)
stream.synchronize()
```

Realizando esta llamada usando el contexto, más cercano al estilo Python:

```
stream = cuda.stream()
with stream.auto_synchronize():
    devary = cuda.to_device(an_array, stream=stream)
    a_cuda_kernel[griddim, blockdim, stream](devary)
    cuda.to_host(devary)
```

Uso de memoria compartida

Como en todo desarrollo CUDA, para obtener el máximo rendimiento minimizando la comunicación entre la CPU host y la GPU, hay que utilizar de forma adecuada la memoria compartida de la GPU, *shared memory*, para realizar manualmente *caching* de los datos.

Anaconda proporciona el objeto *cuda.shared.array(shape, dtype)*, para inidicar un objeto del tipo array de numpy en la GPU.

Un ejemplo de uso de memoria compartida con Anaconda se explica, paso a paso, en el algoritmo implementado de multiplicación de Matrices.

Sincronización de Threads

Para sincronizar el resultado de la ejecución de cada *thread* se utiliza la instrucción *cuda.syncthreads()*, equivalente al comando C de la librería NVIDIA *__syncthreads()*.

Algoritmos Implementados en Anaconda Accelerate

Para mostrar ejemplos de implementación de algoritmo en CUDA con Anaconda se ha realizado la multiplicación de matrices y un ejemplo de estimación de precios por el método Montecarlo.

Multiplicación de Matrices

Para simplificar el algoritmo se asumirá que ambas matrices son cuadradas. Se utilizará memoria compartida del mismo modo, con el mismo reparto de argumentos de entrada en *blocks* y *threads* que la implementación en JCuda.

```
bpg = 50
tpb = 32
n = bpg * tpb

@jit(argtypes=[float32[:,:], float32[:,:], float32[:,:]], target='gpu')
def cu_square_matrix_mul(A, B, C):
    sA = cuda.shared.array(shape=(tpb, tpb), dtype=float32)
    sB = cuda.shared.array(shape=(tpb, tpb), dtype=float32)

    tx = cuda.threadIdx.x
    ty = cuda.threadIdx.y
    bx = cuda.blockIdx.x
    by = cuda.blockIdx.y
    bw = cuda.blockDim.x
    bh = cuda.blockDim.y

    x = tx + bx * bw
    y = ty + by * bh

    acc = 0.
    for i in range(bpg):
        if x < n and y < n:
            sA[ty, tx] = A[y, tx + i * tpb]
            sB[ty, tx] = B[ty + i * tpb, x]

        cuda.syncthreads()

        if x < n and y < n:
            for j in range(tpb):
                acc += sA[ty, j] * sB[j, tx]

        cuda.syncthreads()

    if x < n and y < n:
        C[y, x] = acc
```

Cálculo de generación de precios aleatoria con MonteCarlo

La característica de la generación de precios por Montecarlo es que es bastante intensivo en cómputo, pero afortunadamente es bastante paralelizable dado que cada simulación es casi totalmente independiente de la otra, salvo el punto de partida, que es el precio anterior.

Así pues, la peculiaridad de la implementación del método Montecarlo en CUDA trata de ejecutar cada simulación en paralelo, en un thread, utilizando la memoria compartida para compartir los precios para simulaciones posteriores.

Es un algoritmo muy interesante para demostrar el speedup que se puede obtener al utilizar CUDA. Como optimización adicional, se utilizará la librería el wrapper de la librería CURAND para generar los números aleatorios precisos directamente en la GPU, sin necesidad de acceder a la CPU Host.

Así, detalle de la implementación del método en Anaconda Python:

```
from numbapro.cudalib import curand   "User CURAND"
from numbapro import jit, cuda

@jit('void(double[:], double[:], double, double, double, double[:])',
     target='gpu')

"Esto Genera el equivalente en CUDA C a
  i = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
además determina que los argumentos son escalares, y ahorra el paso de
punteros que se haría de ser arrays"
def step(last, paths, dt, c0, c1, normdist):
    i = cuda.grid(1)
    if i >= paths.shape[0]:
        return
    noise = normdist[i]
    paths[i] = last[i] * math.exp(c0 * dt + c1 * noise)


"Algoritmo montecarlo, una ejecución por Thread"
def monte_carlo_pricer(paths, dt, interest, volatility):
    n = paths.shape[0]
    blksz = cuda.get_current_device().MAX_THREADS_PER_BLOCK
    gridsz = int(math.ceil(float(n) / blksz))
    # Instanciar CURAND para generar números aleatorios
    prng = curand.PRNG(curand.PRNG.MRG32K3A)
    # Reserva de memoria en la GPU
    d_normdist = cuda.device_array(n, dtype=np.double)
    c0 = interest - 0.5 * volatility ** 2
    c1 = volatility * math.sqrt(dt)
    # Simulación
    d_last = cuda.to_device(paths[:, 0])
    for j in range(1, paths.shape[1]):
        prng.normal(d_normdist, mean=0, sigma=1)
        d_paths = cuda.to_device(paths[:, j])
        step(d_last, dt, c0, c1, d_normdist, out=d_paths)
        d_paths.copy_to_host(paths[:, j])
        d_last = d_paths
```

Conclusiones trabajando con Anaconda Accelerate

Esta librería sería la equivalente a Rootbeer para Python, pero con algunas diferencias significativas.

La primer es la licencia, comercial aunque como plugin de la distribución gratuita de Python Anaconda. Esto puede apartar a ciertos desarrolladores pero opino que el precio es bastante bajo y que el valor añadido que aporta bien merece la pena.

Como Rootbeer, presenta la ventaja de poder codificar también el código del núcleo en Python, con gestión casi automática de la memoria en algunos casos, pero permitiendo la suficiente flexibilidad para parametrizar la ejecución en los *threads* y bloques deseados.

Además puede optimizar según el tipo de argumentos sean escalares o no (arrays, objetos, etc.., que se gestionan con punteros).

Y, desde mi punto de vista, clave, la integración con las *Accelerated Libraries* de NVIDIA, CUBLAS, CURAND, etc.. también disponible y que, para mí, era la gran carencia actual de Rootbeer.

También destacar la facilidad de instalación, con asistentes, en los distintos sistemas operativos, Linux, Mac, Windows, y el soporte dado tanto por la Comunidad y la Compañía.

En suma, se nota que es un software soportado comercialmente a un nivel más intenso que el resto de las herramientas probadas en el proyecto, y creo que apunta a lo que llegarán en el futuro productos como Rootbeer de continuar su evolución.

Personalmente, de necesitar en este momento desarrollar en Python para la arquitectura CUDA, sería la opción seleccionada.

Conclusiones finales

Como resultado del proyecto, investigando herramientas para desarrollar en tecnología CUDA he de indicar que personalmente ha sido una experiencia muy enriquecedora.

Por una parte he podido conocer más a fondo las interioridades de esta tecnología de NVIDIA, y aprender la forma correcta de implementar algoritmos en ella, que sería independiente del lenguaje o plataforma de implementación seleccionada.

Por otro lado he podido aplicar el lenguaje que conozco mejor, Java, a esta tecnología, comprobando cómo funcionan librerías tan potentes como JCuda, que en ciertas condiciones podría rivalizar con aplicaciones implementadas totalmente en C/C++, o tecnologías tan prometedoras como Rootbeer, que me parece que es el camino a seguir en el futuro.

Otro aspecto estudiado ha sido el estudio de cómo se puede desarrollar en lenguaje Python en esta tecnología. Tenía interés en aprender más sobre este lenguaje y, en especial muchas de las librerías que incluye como numpy y pandas (para series numéricas).

Cabe reseñar que, en un principio, y en la planificación inicial, las únicas herramientas CUDA para Java y Python previstas eran, respectivamente JCuda y PyCuda pero, tras documentarme al respecto, descubrí la existencia de Rootbeer y Anaconda y decidí que sería muy interesante añadirlas al proyecto, al aportar varias características distintivas.

Otro dato interesante es que en el código implementado se aprecia cómo el desarrollo en lenguaje Python, tanto PyCuda como Anaconda, es siempre más breve que el equivalente Java, debido a la famosa *verbosidad* de Java, además de a la potencia de librerías Python como *numpy*, su manejo de *closures*, etc....

En cuanto al rendimiento, éste ha sido parecido en ambas herramientas. Aún tratándose Python de un lenguaje dinámico, al funcionar directamente sobre el sistema operativo, y al utilizar muchas optimizaciones implementadas directamente en C, *cython*, su rendimiento es, no sólo comparable a Java, sino a vece superior.

Respecto al código ejecutado en el núcleo, para ambos es similar, ya que es el mismo tipo de código, CUDA C - *ptx*, sólo variando si debe ser generado, Anaconda o Rootbeer, o CUDA C explícito, JCuda, PyCuda.

Aún así, en caso de ser generado, éste puede generarse en una fase previa y reutilizarse ya precompilado.

Por último reseñar que la computación paralela me parece el futuro del rendimiento, y para ello se precisan tanto buenos diseños de algoritmos como arquitecturas accesibles, y CUDA presenta la ventaja de acercar esta potencia al consumidor y desarrollador medio, frente a los prohibitivos computadores escalares de no hace tantos años.

Comparativa de herramientas CUDA analizadas

Se indica una pequeña tabla comparativa de las herramientas analizadas:

	JCuda	**RootBeer**	**PyCuda**	**Anaconda Acc.**
Lenguage	Java	Java	Python 2.* - 3.*	Python 2.* - 3.*
Lenguaje Código Núcleo Cuda	CUDA - C	Java	CUDA -C, Python con metaprogramming	Python
Soporte *Accelerated Libraries*	Sí	No	Sí	Sí
NVIDIA toolkit soportado	5.0	5.0	3.5	5.0
Licencia	MIT / X11	MIT	MIT / X11	Comercial, EULA
Coste	Gratis	Gratis	Gratis	129 $

Futuras ampliaciones

Como continuación a este proyecto se podrían seguir varias vías.

Por una parte profundizar en una o varias de estas tecnologías, bien desarrollando un programa de cálculo científico completo en Java, con JCuda, o por ejemplo un programa de minado de Bitcoins. O bien profundizar en Rootbeer y sus características de traducción de lenguaje Java a CUDA -C.

En el caso de Python, se podría incidir en cómo las librerías de cálculo numérico se están optimizando para CUDA con PyCuda, o las reseñadas nuevas características de metaprogramación que aporta PyCuda.

O, si se opta por la herramienta comercial Anaconda Accelerate, aún se puede colaborar con su activa comunidad de usuarios, o implementar diverso software en Python y CUDA.

Otra iniciativa de corte similar sería estudiar cómo desarrollar en arquitectura CUDA en otros lenguajes, especialmente en lenguajes diferentes a los ya estudiados, como pueden ser los funcionales. Al respecto destaca, para **Haskell**, el paquete **meta-par-accelerate**, que permite ejecutar y enrutar cálculos en mónadas en núcleos CUDA, [Amspm12].

También comentar que, en la propia UOC cabría la oportunidad de combinar un proyecto CUDA con otro de Informática Gráfica, trabajando en la integración entre JCuda y JOGL, por ejemplo para la renderización de gráficas en 3d u otros cálculos.

Bibliografía

[Kdh12] Kirk, David B., and W. Hwu Wen-mei. Programming massively parallel processors: a hands-on approach. Morgan Kaufmann, 2nd Edition, 2012.

[Nvd13a]: "NVIDIA GeForce GT 630" [Internet], NVIDIA GeForce Card, 2013, http://www.nvidia.es/object/geforce-gt-630-oem-es.html#pdpContent=2

[Nvd13b]: *"NVIDIA® CUDA® Toolkit",* [Internet], 2013 https://developer.nvidia.com/cuda-toolkit

[Nvd13b]: "NVIDIA CUDA faq" [Internet], CUDA Libraries, 2013, https://developer.nvidia.com/cuda-faq

[Nvd13c]: "TESLA Personal Supercomputing GPU" [Internet],, 2013, http://www.nvidia.com/object/personal-supercomputing.html

[Nvd13d] "*Python incorpora soporte de NVIDIA CUDA*" NVIDIA Sala de Prensa. http://www.nvidia.es/object/cuda-gpu-computing-python-support-18mar-2013-es.html

[Nvd13e] "*GPU Accelerated Libraries*" [Internet] NVIDIA. https://developer.nvidia.com/gpu-accelerated-libraries

[Bot11]: "Numerical computations in Java with CUDA" [Internet], 2011, http://www.world-education-center.org/index.php/P-ITCS/article/view/768

[Jcu13a]: "*JCUDA JAVA bindings for CUDA*" [Internet], 2013, http://www.jcuda.org/

[DiCo08]: "*GPU Quicksort Library*" [Internet], 2008, Distributed Computing & Systems http://www.cse.chalmers.se/research/group/dcs/gpuqsortdcs.html

[Pyc13a]: "*PyCUDA*" [Internet], 2013, GPU Programming with Python http://mathema.tician.de/software/pycuda

[Roo13a]: *"RootBeer"* [Internet], 2013, Root Beer Java CPU Compiler http://rbcompiler.com/

[Roo13b]: *"RootBeer GitHub"* [Internet], 2013, Root Beer Git Repository https://github.com/pcpratts/rootbeer1

[Psil13] *"Psi Lambda LLC | Parallel computing made practical",* [Internet] 2013 http://psilambda.com/products/kappa/

[Pgic13] *"PGI CUDA Fortran Compiler"* [Internet], 2013 http://www.pgroup.com/resources/cudafortran.htm

[Cour12] *"Heterogeneous Parallel Programming",* [Internet] Wen-mei W. Hwu, 2012 https://www.coursera.org/course/hetero

[Auda12] "*Introduction to Parallel Programming*", [Internet] Owens, Luebke, Cheng-Han, Roberts, 2012 https://www.udacity.com/course/cs344

[Jocl13]: "*jocl.org - Utilities*" [Internet], 2013, jocl.org - Utilities http://www.jocl.org/utilities/utilities.html

[Hull04]: John C. Hull, "*Fundamentals of Futures and Option Markets*" 5th Edition, 2004, Prentice Hall.

[CuMa10]: "*A Python matrix class that uses CUD.*" [Internet], CUDAMat http://code.google.com/p/cudamat/

[CoAn13a]: "*Anaconda*" [Internet], 2013, Continuum Analytics https://store.continuum.io/cshop/anaconda/

[CoAn13b]: "*Anaconda Accelerata*" [Internet], 2013, Continuum Analytics https://store.continuum.io/cshop/accelerate/

[Amspm12]: " *A Meta-Scheduler for the Par-Monad*" [Internet, pdf], 2012, Indiana Univ. http://www.cs.indiana.edu/~rrnewton/papers/meta-par_submission.pdf