UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Proyecto TGA

Laboratorio de Tarjetas Gráficas y Aceleradores

Departmento de Arquitectura

Ordenación por fusión paralela: *MergePath* Implementación en CUDA y OpenCL

Autores:
Alex Herrero
Walter Troiani
tga1009

Profesores
Dani & Agustín

14 de enero de 2024

Agradecimientos

Gracias a Agustín por enseñarme que los tipos de NVIDIA están metiendo mucha pasta y a Dani por las referencias de PAR y decirle hasta a mi abuela que se apuntara a BitsXLaMarató (desgraciadamente ella solo sabe programar la lavadora y no en C). Nos lo hemos pasado muy bien en esta asignatura y hemos aprendido mucho sin haber dado palo al agua hasta navidades, muchas gracias!

Índice

1. Motivación del trabajo

En las últimas clases de teoría con Agustín, él nos comentó, como bien sabíamos, que CUDA solo tenía soporte para las tarjetas gráficas de Nvidia y que si queríamos seguir con el paradigma de paralelismo en tarjetas gráficas para todo tipo de fabricando había una alternativa de código abierto y heterogénea: OpenCL.

Las últimas clases del curso Agustín nos estuvo explicando (un poco por encima) el funcionamiento de OpenCL y nos dijo algo similar a lo siguiente: *Realmente los tipos de NVIDIA tienen muy bien optimizado CUDA para sus tarjetas gráficas ya que ellos mismos saben la arquitectura que hay por dentro y me gustaría poneros algún ejemplo de tiempos para CUDA y OpenCL, pero desgraciadamente no tengo. Si alguien quiere hacer el proyecto de CUDA vs OpenCL y pasarme los tiempos, yo los pondré encantado.*

Así que con afán de protagonismo y ganas de cambiar el mundo (y que nuestra alma quede grabada para el resto de la eternidad en las transparencias de TGA, junto al gráfico de escala logarítmica de nvidia vs ATI) decidimos tomar la iniciativa y correr un programa en CUDA, OpenCL y comparar diferentes parámetros para ver el rendimiento.

2. El algoritmo y su implementación

2.1. Algoritmo

Dani me hizo recordar un algoritmo que hicimos en PAR el cuatrimestre pasado en una práctica: El *MultiSort*. La idea del *MultiSort* es la siguiente:

1. Partes el vector recursivamente en trozos iguales
2. Cuando el vector tenga un tamaño *pequeño* ejecutas un algoritmo de ordenación básico (por ej *Insertion Sort*)
3. Juntas las partes ordenadas con un algoritmo de fusión

Se podría pensar como un *MergeSort* pero la idea es que cuando llegas a un número determinado de elementos aplicas un algoritmo de ordenación sencillo. Así que la idea ya estaba clara, ahora solo hacía falta programar. Pensando en cómo hacer el algoritmo para CUDA (y así luego pasarlo a OpenCL) nos vino a la mente el cómo hacíamos las reducciones: Las hacíamos en forma de árbol, pero *bottom-up* (es decir: primero comenzábamos con los hijos, luego con los padres de los hijos... hasta llegar a la raíz) y esta idea rondaba nuestra cabeza para poder hacer la parte de fusión.

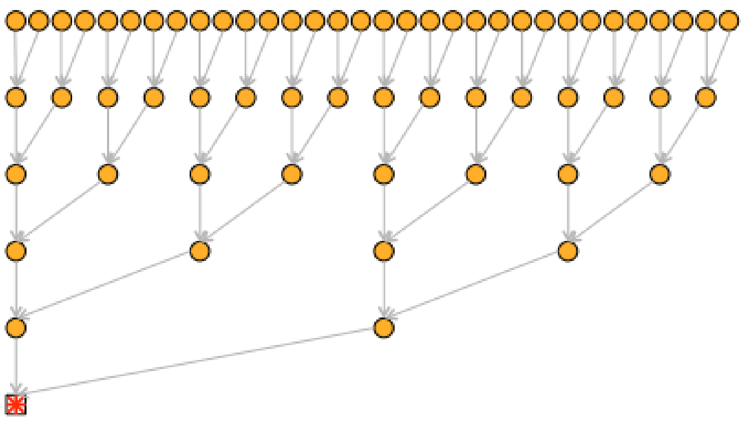

Figura 1: Esquema de cómo hacer una reducción en CUDA, sacado de la práctica 3

Así que con esta idea de construir el árbol ya teníamos en mente cómo hacer el MultiSort:

1. El primer nivel del árbol se encargaría de hacer un *Insertion Sort* a secciones de *size_ i* (una variable que definiremos nosotros) del vector

2. Los demás niveles del árbol serán para hacer la parte de *merge*

Para el resto del trabajo asumiremos que cada nivel de este árbol tiene un 2^m nodos (para un cierto $m \geq 0$) ya que así nos facilitará mucho el código del *merge*. Aunque se podría hacer con números que no sean potencia de dos nos liaría mucho el código y dado que este no es el propósito principal de este trabajo hemos decidido suponer que n y $size_i$ son potencias de dos.

2.2. Su versión secuencial

Esta versión realmente es lo que uno esperaría de hacer una modificación al *MergeSort* donde `insertion` y `merge` son las típicas funciones para hacer *Insertion Sort* y *Merge Sort* respectivamente.

```
    void multisort(int *v, int* aux, int i, int j) {

        if (j-i+1 == size_i) insertion(v, i, j);

        else {
                int mid = i + (j-i)/2;
                multisort(v, aux, i, mid);
                multisort(v, aux, mid+1, j);
                merge(v, aux, i, mid, j);
        }
}
```

El parámetro de $size_i$ puede producir resultados en el rendimiento ya que este determina el caso base de nuestra recursividad, donde hacemos *Insertion Sort*. Así que tenemos que tener presente que, modificar el valor de $size_i$, puede afectar a nuestro rendimiento. También tenemos que tener en cuenta el *input* de los vectores, ya que los momentos de *Insertion Sort* pueden resultar en más o menos eficientes. Para ello hemos supuesto tres posibles casos:

- Vector totalmente ordenado (modo ascendiente)
- Vector ordenado al revés (modo descendiente)
- Input aleatorio (para generar este input usaremos la semilla 21364 en honor a la CPU de Alpha 21364. Agustín dijo en clase que jugaba a este número a la lotería, así que lo usaremos como semilla para ver si nos puede dar un poco de suerte en la nota de este trabajo).

3. Experimentación

3.1. Preámbulo

Para todos los tiempos que vamos a presentar hemos ejecutado en el *Boada-10* cada `job.sh` 5 veces y con un *script* hecho en *Python* (llamado `mean.py`, si se quiere ejecutar previamente entra al código a leer el primer comentario que hay) hemos hecho la media de los 5 tiempos. Para más detalles consultar el archivo de la práctica (`experimentos/README.md`). Los resultados también los dejamos dentro de los la carpeta `experimentos/` para que puedan ser consultados (aunque dejaremos solo la media de los cinco tiempos, ya que se nos olvidó guardar también los otros cinco resultados).

Los experimentos que hemos elegido hacer son:

- Encontrar el valor de $size_i$ que mejores tiempos dé en cada caso. Para ello fijamos $n = 2^{25}$ elementos y 1024 *threads* por bloque.
- Con el valor de $size_i$ encontrado en el anterior apartado vamos moviendo el tamaño de los bloques de *threads* para encontrar el óptimo. De nuevo seguimos usando $n = 2^{25}$ elementos.
- Finalmente, con los dos parámetros encontrados anteriormente, vamos modificando el valor de n en el rango de 2^{15} hasta 2^{27}.
- Cada experimento se hace para cada posible *input* del vector.

Para los *kernels* de *merge* hemos decidido lanzar múltiples instancias del kernel en vez de ejecutarlo una sola vez. El porqué de esto es simplemente porque nos tenemos que asegurar que todos los *threads* acaben el mismo nivel del árbol antes de ejecutar el siguiente, si no podríamos tener problema de que un *thread* machaque el resultado de otro y el vector no se ordene correctamente. Como los *kernels* se lanzan de forma asíncrona y se ejecutan de forma secuencial (es decir, no se ejecutan dos kernels en paralelo) simplemente enviamos todos los niveles del árbol en orden y así nos aseguramos de ejecutar todos y que todos los threads acaben el mismo nivel.

Y dicho esto, como dirían los dermatólogos, vamos al grano.

3.2. Versión secuencial

3.2.1. Determinando el valor de $size_i$ para cada caso

Los resultados obtenidos en **Figure 2** son de lo más esperados: Por la naturaleza del *Insertion Sort* cuanto más ordenado esté el input menos operaciones requiere hacer. Así que parece normal que en el *input* ordenado lo mejor sea hacer el *Insertion Sort* lo más grande posible, en el *input* al revés lo mejor sea parar pronto el *Sorting* y en el *input* aleatorio pararte más o menos por el medio de los valores elegidos.

$size_i$	Elapsed Time (ms)	Mode	Elapsed Time (ms)	Mode	Elapsed Time (ms)	Mode
2	766.542	Ordenado	4728.912	Random	904.838	Al revés
4	661.111	Ordenado	4590.567	Random	852.774	Al revés
8	593.976	Ordenado	4427.412	Random	831.837	Al revés
16	565.239	Ordenado	4277.904	Random	888.678	Al revés
32	542.969	Ordenado	4162.833	Random	1057.361	Al revés
64	510.091	Ordenado	4142.445	Random	1444.874	Al revés
128	511.859	Ordenado	4351.063	Random	2296.787	Al revés
256	472.952	Ordenado	4914.304	Random	3821.133	Al revés
512	454.506	Ordenado	6227.814	Random	6796.477	Al revés
1024	432.095	Ordenado	9026.392	Random	12734.847	Al revés
2048	419.006	Ordenado	14814.404	Random	24687.505	Al revés
4096	403.629	Ordenado	26563.575	Random	48488.934	Al revés

Figura 2: Tabla con los tiempos secuenciales cambiando la variable $size_i$, $n = 2^{25}$

No parece haber salido ninguna sorpresa de aquí.

3.2.2. Comportamiento según aumenta el tamaño

Una vez hemos encontrado el valor de $size_i$ en el anterior apartado, ahora veremos hasta dónde puede llegar nuestro algoritmo, para así poder tener una idea sobre qué mejoras hacemos respecto a ejecutar esto en una tarjeta gráfica. Los resultados se pueden observar en **Figura 3**.

n	Elapsed Time (ms)	Mode	Elapsed Time (ms)	Mode	Elapsed Time (ms)	Mode
2^{15}	0.239200	Ordenado	1.967000	Random	0.565800	Al revés
2^{16}	0.370200	Ordenado	12.231799	Random	1.278600	Al revés
2^{17}	1.258200	Ordenado	20.300200	Random	7.766200	Al revés
2^{18}	3.356200	Ordenado	25.133200	Random	12.441600	Al revés
2^{19}	6.726600	Ordenado	48.264200	Random	23.368200	Al revés
2^{20}	10.809000	Ordenado	101.012199	Random	27.068800	Al revés
2^{21}	20.390000	Ordenado	215.542404	Random	44.628401	Al revés
2^{22}	35.863200	Ordenado	448.207605	Random	86.203801	Al revés
2^{23}	76.040199	Ordenado	948.131799	Random	182.165399	Al revés
2^{24}	183.657999	Ordenado	1985.882007	Random	402.039203	Al revés
2^{25}	399.904804	Ordenado	4161.796875	Random	842.171790	Al revés
2^{26}	884.004993	Ordenado	8720.373047	Random	1771.928784	Al revés
2^{27}	1950.042212	Ordenado	18299.381641	Random	3780.739014	Al revés

Figura 3: Tabla con los tiempos secuenciales cambiando la variable n

3.2.3. Comentarios a destacar

Parece ser que el valor de *Insertion Sort* con el que comenzar va a ser algo que nos va a afectar en el rendimiento de los siguientes *kernels*. La naturaleza del algoritmo de *merge* clásico es hacer siempre los mismos pasos sin importar cómo esté el *input* del vector. El siguiente *kernel* es simplemente la implementación de este exacto algoritmo pero en paralelo, así que uno se puede esperar resultados similares para los valores de $size_i$.

3.3. Primera Versión del *Merge*

Esta es la versión que cualquier persona podría esperar: El clásico código de *merge* pero esta vez se hacen en paralelo (podemos pensar en esta ejecución del *Merge* como **Figura 1**). Para ello cada nivel del árbol lo identificaremos con el nivel i donde i es el multiplicador del tamaño de $size_i$ que tiene que tomar cada nodo (para tener una idea en el nivel i tenemos que hacer una fusión de dos vectores de tamaño $n = \frac{i}{2} \cdot size_i$). El primer nivel del *merge* es $i = 2$, luego $i = 4$, luego $i = 8$... El código del *Kernel* se puede consultar (`cudaFirstMerge.cu` o `cudaFirstMergeBlock.cu`). Pero la idea es que cada *thread* comienza en la posición $id \cdot i \cdot size\_i$ y acaba en la posición $beg + size\_i \cdot i - 1$ (el punto medio es simplemente $beg + size\_i \cdot (i/2) - 1$. Los vectores a fusionar son $v[beg...mid], v[mid + 1...end]$)

3.3.1. Determinando el valor de $size_i$

Como el anterior experimento ejecutamos, para cada modo, con $n = 2^{25}$, diferentes valores de $size_i$ desde $size_i = 2^1$ hasta $size_i = 2^{12}$. Presentamos los resultados en la siguiente **Tabla 4**. Los resultados del valor son similares a la versión secuencial, salvo que en el *input* aleatorio vemos que lo mejor es pararlo pronto.

$size_i$	T. Kernels(ms)	Modo	T. Kernels(ms)	Modo	T. Kernels(ms)	Modo
2	5348.041894	Ordenado	8013.150488	Random	5409.243847	Al revés
4	5334.277637	Ordenado	8012.198047	Random	5406.132715	Al revés
8	5335.623438	Ordenado	8022.601660	Random	5409.669531	Al revés
16	5350.372851	Ordenado	8021.432129	Random	5411.079590	Al revés
32	5350.800683	Ordenado	8041.643848	Random	5465.024609	Al revés
64	5345.064258	Ordenado	8084.479199	Random	5585.198535	Al revés
128	5339.679981	Ordenado	8165.512500	Random	5786.523828	Al revés
256	5325.435547	Ordenado	8367.383789	Random	6603.608984	Al revés
512	5314.324707	Ordenado	8785.497461	Random	8116.268555	Al revés
1024	5296.339453	Ordenado	8436.172656	Random	8591.048047	Al revés
2048	5286.476758	Ordenado	9425.470117	Random	8546.990235	Al revés
4096	5281.515723	Ordenado	13815.802930	Random	18131.647656	Al revés

Figura 4: Tabla con los tiempos del primer merge cambiando la variable de $size_i$, $n = 2^{25}$ y 1024 *threads* por bloque

3.3.2. Comportamiento según el tamaño del bloque

Un curioso comportamiento que podemos observar puede ser a la hora de modificar el tamaño de un bloque de *threads*. Por ello, fijando los valores de $size_i$ encontrados en el apartado anterior, fijando $n = 2^{25}$, vamos a ir cambiando el tamaño de los bloques. Los resultados se pueden ver en **Figura 5**.

Tamaño	T. Kernels(ms)	Modo	T. Kernels	Modo	T. Kernels(ms)	Modo
1024	5289.06	Ordenado	8006.76	Random	5395.30	Al revés
512	5207.41	Ordenado	7951.11	Random	5329.47	Al revés
256	5143.54	Ordenado	7867.68	Random	5263.55	Al revés
128	5090.58	Ordenado	7859.28	Random	5209.65	Al revés
64	5065.96	Ordenado	7850.74	Random	5184.41	Al revés
32	5055.35	Ordenado	7837.95	Random	5151.71	Al revés
16	5000.57	Ordenado	7655.95	Random	5049.94	Al revés
8	4933.29	Ordenado	7386.72	Random	4982.59	Al revés
4	4877.91	Ordenado	6865.65	Random	4924.20	Al revés
2	4819.03	Ordenado	6145.51	Random	4865.92	Al revés
1	4757.38	Ordenado	5284.25	Random	4818.79	Al revés

Figura 5: Cambio del tamaño de bloques con $n = 2^{25}$ y $size_i$ encontrado en el apartado anterior

3.3.3. Comportamiento según aumenta el tamaño

Como en la parte secuencial, una vez hemos encontrado los valores de $size_i$ y del tamaño de bloque vamos modificando el tamaño de n. Los resultados se presentan en **Figure 6**.

n	T. Kernels(ms)	Modo	T. Kernels(ms)	Modo	T. Kernels(ms)	Modo
2^{15}	3.924051	Ordenado	4.062720	Random	3.639514	Al revés
2^{16}	7.954150	Ordenado	8.090989	Random	7.246054	Al revés
2^{17}	16.000320	Ordenado	16.155495	Random	14.465248	Al revés
2^{18}	32.114464	Ordenado	32.289248	Random	28.908038	Al revés
2^{19}	64.307621	Ordenado	64.579275	Random	57.804858	Al revés
2^{20}	157.466513	Ordenado	163.945514	Random	142.688837	Al revés
2^{21}	295.110620	Ordenado	328.010736	Random	300.993249	Al revés
2^{22}	590.406384	Ordenado	656.290833	Random	602.231640	Al revés
2^{23}	1181.464966	Ordenado	1313.147436	Random	1205.071606	Al revés
2^{24}	2366.720996	Ordenado	2627.727881	Random	2411.243799	Al revés
2^{25}	4738.353906	Ordenado	5257.937696	Random	4825.041406	Al revés
2^{26}	9481.280469	Ordenado	10547.728906	Random	9654.998828	Al revés
2^{27}	19049.212891	Ordenado	21198.063672	Random	19318.919141	Al revés

Figura 6: Variación de n fijando $size_i$ y el tamaño de bloques con los apartados anteriores

3.3.4. Comentarios a destacar

Aquí tenemos algunas cosas a destacar:

- Respecto a variar la variable de $size_i$ los tiempos son mucho más estables en este *kernel* que en su versión secuencial (ver **Figura 2 vs Figura 4**). Esto nos hace ver que, en el caso paralelo, los tiempos de *kernels* son mayormente afectados por el *merge*. No parece que sea crucial mejorar el tiempo de *Insertion*.
- El tiempo del *kernel* es mucho más grande que el tiempo en secuencial (**Figura 6 vs Figura 3**). A primeras parece no tener sentido, pero si uno estudia las tarjetas gráficas se da cuenta de que sí que puede pasar: Este kernel, según vamos profundizando en el árbol, vamos usando menos *threads* y cada *thread* hace un trabajo más grande. Un *thread* de *GPU* es mucho más lento que un *thread* de CPU (en GPU tenemos muchísimos pero *tontos*, en CPU tenemos el caso contrario) así que es probable que en algún punto del árbol hacerlo directamente en CPU sea mucho mejor que hacerlo en GPU (intuyo que en los últimos niveles, donde tener solo $2, 4$ threads de GPU puede ser peor que tener solo 1 thread de CPU).
- Bajo toda sorpresa parece que lo mejor para el tamaño de bloques es tener un solo *thread* por bloque. Tenemos la sospecha de que tenga que ver con que a medidas que vas avanzando en el *merge* vas necesitando menos *threads* para ejecutar, aunque por falta de tiempo hemos decidido no investigar más a fondo.

3.4. Segunda Versión del *Merge*

Esta es la versión mejorada del *Merge*. En el anterior *kernel* he comentado que el problema principal es el tiempo que tarda el *merge* en ejecutarse. La principal razón de esto es que perdemos paralelismo a medida que vamos yendo más profundo en el árbol de *merge* y cada *thread* en un nivel del árbol o bien se descarta o bien hace un trabajo el doble de grande que el anterior.

En esta versión vamos a solucionar estas dos cosas, haciendo que todos los *threads* trabajen todo el rato y que el trabajo de un *thread* no aumente a medida que bajamos en el árbol, sino que siempre haga un trabajo constante. Para ello vamos a basarnos en una técnica llamada *MergePath* (aunque nos basamos en[1] solo nos basaremos en la idea, el código que ofrecen ellos es horrible y he decidido implementar el mío propio).

La idea del *MergePath* es la siguiente: Dados dos vectores A, B que queremos fusionar y n threads queremos que cada *thread* haga $(|A| + |B|)/n$ pasos. Para ello a cada *thread* le asignaremos

	B[0]	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]
A[0]	1	1	1	0	0	0	0	0
A[1]	1	1	1	0	0	0	0	0
A[2]	1	1	1	1	0	0	0	0
A[3]	1	1	1	1	0	0	0	0
A[4]	1	1	1	1	1	0	0	0
A[5]	1	1	1	1	1	0	0	0
A[6]	1	1	1	1	1	0	0	0
A[7]	1	1	1	1	1	0	0	0

Figura 7: Ejemplo *PathMerge*

una coordenada (i, j) donde i es la posición $A[i]$ y j es la posición $B[j]$ donde tiene que comenzar. Cada *thread* hará $(|A| + |B|)/n$ pasos acabando en las posición (k, l). Entonces, en paralelo, el siguiente *thread* tiene que comenzar en la posición (k, l) y aplicar el mismo razonamiento... Parece magia, pero simplemente es intelecto sobre cómo encontrar esa posición.

Para encontrar esa posición se hace una cosa bastante inteligente: Imaginemos una matriz ficticia con $|A|$ filas y $|B|$ columnas. En la posición (i, j) de la matriz pondremos un uno si $A[i] > B[j]$ y un zero en caso contrario. En esta matriz, para seguir el camino de *Merge*, dada una posición nos moveremos a la derecha si en esa posición hay un uno y nos moveremos para abajo en el caso contrario. La gracia de esto es que, si nos movemos a la derecha, estamos aumentando la coordenada de la columna, mientras que si nos movemos abajo estamos aumentando la coordenada de la fila (esta es la gracia de cómo hemos definido lo que es un 1 y lo que es un 0 en esta matriz).

Sobre el cómo asignar posiciones a *threads* es sencillo: Ponemos n diagonales en la matriz para partirla en trozos iguales y a cada thread le asignamos una diagonal. Para tener un ejemplo más claro: Supongamos dos vectores con $|A| = |B| = 8$ y dos *threads*. El *merge* total se obtiene en 16 pasos pero como tenemos dos *threads* queremos repartir la carga en que cada *thread* haga 8 movimientos. Como son dos *threads* partimos la matriz en dos trozos iguales (claro, al partir me refiero en poner una diagonal en esa matriz) y el primer thread puede comenzar en el inicio $(0, 0)$; El segundo *thread*, por cómo hemos definido los 1's y los 0's tendrá que comenzar en el punto en el que la diagonal cruce un 0 con un 1 (si se piensa un poco se visualiza). Encima esto es maravilloso porque, si miramos la matriz, podemos buscar el cruce de los 1's son los 0's con una búsqueda binaria!.

Un ejemplo puede ser **Figura 7** donde visualizamos un ejemplo en el que partimos la matriz en dos y buscamos en la diagonal el punto en el que se cruzan los 1's con los 0's. El primer *thread* comienza en la posición $(0, 0)$ y el segundo en la posición $(4, 4)$.

Si se quiere ver más claro se puede consultar el código `cudaPathMerge.cu` o `cudaPathMergeBlock.cu`

3.4.1. Determinando el valor de $size_i$

Una vez más procedemos con los mismos experimentos

$size_i$	T.Kernels (ms)	Modo	T.Kernels (ms)	Modo	T.Kernels (ms)	Modo
2	1.597485	Ordenado	1.667315	Random	1.542195	Al revés
4	2.285459	Ordenado	2.789568	Random	2.509402	Al revés
8	4.022374	Ordenado	7.151398	Random	6.165069	Al revés
16	10.535098	Ordenado	17.916800	Random	16.932359	Al revés
32	18.151008	Ordenado	50.536627	Random	79.097698	Al revés
64	24.009107	Ordenado	108.759564	Random	212.039307	Al revés
128	29.465984	Ordenado	205.236090	Random	429.026080	Al revés
256	33.408128	Ordenado	429.081586	Random	1253.394531	Al revés
512	36.686419	Ordenado	864.055152	Random	2787.454248	Al revés
1024	33.186476	Ordenado	504.715961	Random	3272.976025	Al revés
2048	44.874778	Ordenado	1507.630151	Random	3225.232520	Al revés
4096	75.730188	Ordenado	5906.261035	Random	12813.044140	Al revés

Figura 8: Cambio de valor $size_i$ para el *PathMerge*, $n = 2^{25}$

3.4.2. Comportamiento según el tamaño del bloque

De nuevo, fijando $n = 2^{25}$ y $size_i = 2$ para todos los casos obtenemos la siguiente tabla.

Tamaño bloque	Modo	T.Kernels(ms)	Modo	T.Kernels(ms)	Modo	T.Kernels(ms)
1024	Ordenado	1.496192	Random	1.869658	Al revés	1.680979
512	Ordenado	1.486202	Random	1.560442	Al revés	1.498758
256	Ordenado	1.496192	Random	1.566810	Al revés	1.505926
128	Ordenado	1.486426	Random	1.557517	Al revés	1.496563
64	Ordenado	1.484435	Random	1.548090	Al revés	1.497088
32	Ordenado	1.357018	Random	1.475750	Al revés	1.404634
16	Ordenado	2.255891	Random	2.487302	Al revés	2.353043
8	Ordenado	4.223475	Random	4.597702	Al revés	4.413133
4	Ordenado	8.164288	Random	8.637510	Al revés	8.548512
2	Ordenado	15.105478	Random	15.418342	Al revés	15.684915
1	Ordenado	28.193594	Random	28.503654	Al revés	28.975264

Figura 9: Comportamiento de la variación del tamaño de bloques fijando $size_i$ y n

3.4.3. Comportamiento según aumenta el tamaño

Con el valor de $size_i$, junto al tamaño de bloque, encontrado para el anterior apartado repetimos lo mismo.

n	Modo	T.Kernels(ms)	Modo	T.Kernels(ms)	Modo	T.Kernels(ms)
2^{15}	Ordenado	0.058291	Random	0.059616	Al revés	0.060326
2^{16}	Ordenado	0.061434	Random	0.063910	Al revés	0.064230
2^{17}	Ordenado	0.066317	Random	0.069734	Al revés	0.071942
2^{18}	Ordenado	0.076365	Random	0.079450	Al revés	0.082202
2^{19}	Ordenado	0.090982	Random	0.096653	Al revés	0.097952
2^{20}	Ordenado	0.115488	Random	0.123770	Al revés	0.125018
2^{21}	Ordenado	0.159168	Random	0.171546	Al revés	0.170349
2^{22}	Ordenado	0.242182	Random	0.263520	Al revés	0.256454
2^{23}	Ordenado	0.405734	Random	0.442400	Al revés	0.426995
2^{24}	Ordenado	0.724448	Random	0.789440	Al revés	0.754669
2^{25}	Ordenado	1.356602	Random	1.477414	Al revés	1.407526
2^{26}	Ordenado	2.616128	Random	2.849363	Al revés	2.700134
2^{27}	Ordenado	5.133984	Random	5.590374	Al revés	5.286400

Figura 10: Comportamiento de *PathMerge* según aumenta el valor de n

3.4.4. Comentarios a destacar

Esto es mucho más interesante:

- Primero vemos una gran mejora respecto al anterior *Merge*, alineando con lo dicho anteriormente de que el problema de los *kernels* residía en hacer el *Merge*. Para hacernos una idea **Figura 11** representa el *SpeedUp* obtenido de los tiempos del *kernel* del primer *Merge* respecto al segundo.
- El tamaño de $size_i$ es el mismo para los tres tipos de *inputs*. Esto en verdad es lo que más sentido tiene puesto a que, cuanto más pequeño es el tamaño de $size_i$ más *threads* necesitaremos para hacer el *merge* al incio y como los *threads* no se pierden en ningún momento, sino que todo el rato hacen trabajo en todos los niveles entonces se obtiene cada vez más paralelismo.
- El comportamiento según aumenta el número de elementos (**Figura 10**) parece muy estable. De hecho, aunque vemos aumentos exponenciales en el tamaño del *input* los tiempos no aumentan de esa forma, lo hacen más *relajadamente*
- Compartiendo con su otra versión del *Merge* se ve que lo óptimo (aunque esta vez las diferencias no son muy abismales) no es llenar el tamaño de los bloques, sino dejarlos en grupos de 32 *threads* por bloque.

3.5. Comparación entre los dos *kernels*

Finalmente tendremos que dar el ganador a mejor *merge* del año, para ello primero midamos el *SpeedUp* de los tiempos de los *kernels* en la primera y segunda versión. Para ello medimos $SpeedUp = Normal/Path$

n	Modo	*SpeedUp*	Modo	*SpeedUp*	Modo	*SpeedUp*
2^{15}	Ordenado	67.318299	Random	68.148148	Al revés	60.330769
2^{16}	Ordenado	129.474721	Random	126.599734	Al revés	112.814168
2^{17}	Ordenado	241.270262	Random	231.673144	Al revés	201.068194
2^{18}	Ordenado	420.539043	Random	406.409666	Al revés	351.670738
2^{19}	Ordenado	706.816964	Random	668.155929	Al revés	590.134535
2^{20}	Ordenado	1363.488094	Random	1324.598158	Al revés	1141.346342
2^{21}	Ordenado	1854.082604	Random	1912.086181	Al revés	1766.921138
2^{22}	Ordenado	2437.862368	Random	2490.478267	Al revés	2348.302776
2^{23}	Ordenado	2911.920041	Random	2968.235615	Al revés	2822.214794
2^{24}	Ordenado	3266.930126	Random	3328.597336	Al revés	3195.101162
2^{25}	Ordenado	3492.810644	Random	3558.879025	Al revés	3428.030037
2^{26}	Ordenado	3624.165357	Random	3701.784892	Al revés	3575.748029
2^{27}	Ordenado	3710.415321	Random	3791.886495	Al revés	3654.456557

Figura 11: *SpeedUp* del tiempo de los *kernels*

Creo que no hace falta decir quién es el ganador de esta comparación...

3.6. Rendimiento final

3.6.1. Anchos de Banda del *PathMerge*

Para ver cómo de bueno es nuestro kernel para ello primeramente hemos calculado el Ancho de Banda a través de cómo habla NVIDIA de este cálculo[2] (los cálculos de los .txt que pone *Ancho de Banda Kernels* pueden ser ignorados, ya que están mal tomados) donde se calcula el ancho de banda como $BW = (R_B + W_B)/(t \cdot 10^9)$ donde R_b son los *bytes* que lee el kernel, W_B los *bytes* que escribe el *kernel* y t es el tiempo que tarda el kernel que está en milisegundos. Presentamos los datos en la siguiente tabla.

n	Modo	*A.Banda Kernel*	Modo	*A.Banda Kernel*	Modo	*A.Banda Kernel*
2^{15}	Ordenado	4.497161 GB/s	Random	4.397209 GB/s	Al revés	4.345456 GB/s
2^{16}	Ordenado	8.534167 GB/s	Random	8.203536 GB/s	Al revés	8.162665 GB/s
2^{17}	Ordenado	15.811572 GB/s	Random	15.036797 GB/s	Al revés	14.575297 GB/s
2^{18}	Ordenado	27.462214 GB/s	Random	26.395872 GB/s	Al revés	25.512177 GB/s
2^{19}	Ordenado	46.100372 GB/s	Random	43.395487 GB/s	Al revés	42.819993 GB/s
2^{20}	Ordenado	72.636187 GB/s	Random	67.775778 GB/s	Al revés	67.099202 GB/s
2^{21}	Ordenado	105.405710 GB/s	Random	97.800100 GB/s	Al revés	98.487317 GB/s
2^{22}	Ordenado	138.550479 GB/s	Random	127.331633 GB/s	Al revés	130.839964 GB/s
2^{23}	Ordenado	165.401135 GB/s	Random	151.692731 GB/s	Al revés	157.165456 GB/s
2^{24}	Ordenado	185.268961 GB/s	Random	170.016376 GB/s	Al revés	177.849796 GB/s
2^{25}	Ordenado	197.873404 GB/s	Random	181.692779 GB/s	Al revés	190.714385 GB/s
2^{26}	Ordenado	205.215843 GB/s	Random	188.417872 GB/s	Al revés	198.831211 GB/s
2^{27}	Ordenado	209.143976 GB/s	Random	192.069766 GB/s	Al revés	203.113995 GB/s

Figura 12: Anchos de Banda para *PathMerge*

El ancho de banda de la RTX 3080 es de 760GB/s así que en el mejor de los casos estamos usando el 27,5 %; Sí que es cierto que a medida que vamos aumentando el tamaño el ancho de banda también aumenta, lo que nos da una buena señal de que nuestro algoritmo es robusto en el tiempo. Aunque hayamos hecho una gran mejora respecto al primer *merge* vemos que aún nos puede quedar camino para poder dominar aún más ancho de banda. Esto puede ser por la naturaleza de tener tantos condicionales en la parte de *merge* haciendo que todos los *threads* no se puedan ejecutar a la vez. No sé si podemos hacer mucho más, pero aquí pararán nuestras mejoras.

3.6.2. Secuencial vs *PathMerge*

Finalmente veamos cuánta mejora hemos hecho respecto al trabajo secuencial. Los tiempos anteriores solo tienen en cuenta el tiempo de los kernels de *Insertion* más el tiempo de acabar los *merges*. Si queremos comparar respecto al tiempo secuencial, deberíamos de tener en cuenta el tiempo de comunicación de datos entre CPU-GPU. Ahora nos toca hacernos una pregunta básica: Cuánto ganamos respecto a ejecutarlo en secuencial? Para ello vamos a tener en cuenta , para el tiempo de CUDA, el tiempo de pasar los datos de CPU a GPU + el tiempo de ejecutar los kernels + el tiempo de pasar los datos de GPU a CPU. Hemos decidido considerar los datos entre CPU-GPU porque el precio a pagar a cambio de mucho paralelismo es pasar por la transferencia de datos. Para calcular el tiempo entre transferencia de datos hemos hecho uso del ancho de banda que hemos calculado para entre *Host-To-Device* y *Device-To-Host*.

n	Modo	*SpeedUp*	Modo	*SpeedUp*	Modo	*SpeedUp*
2^{15}	Ordenado	4.099006	Random	32.959358	Al revés	9.368969
2^{16}	Ordenado	6.016438	Random	191.102436	Al revés	19.876360
2^{17}	Ordenado	18.920681	Random	290.353549	Al revés	107.666137
2^{18}	Ordenado	43.737836	Random	314.891241	Al revés	150.671787
2^{19}	Ordenado	73.469405	Random	496.436611	Al revés	237.169044
2^{20}	Ordenado	92.804587	Random	809.513309	Al revés	214.777202
2^{21}	Ordenado	126.639632	Random	1242.789280	Al revés	259.177008
2^{22}	Ordenado	145.931197	Random	1677.643551	Al revés	331.479971
2^{23}	Ordenado	184.112533	Random	2108.730435	Al revés	419.565174
2^{24}	Ordenado	248.554387	Random	2469.807801	Al revés	522.636593
2^{25}	Ordenado	288.571661	Random	2758.962344	Al revés	586.245806
2^{26}	Ordenado	330.410531	Random	2985.941245	Al revés	642.598468
2^{27}	Ordenado	371.142964	Random	3162.167000	Al revés	700.041548

Figura 13: *SpeedUp* del tiempo de los *kernels*

Finalmente podemos ver que, efectivamente, hemos conseguido *SpeedUp* respecto a la versión secuencial :-).

3.7. OpenCL

Finalmente vamos a repetir el experimento de *PathMerge* para OpenCL.

3.7.1. Determinando el valor de $size_i$

Finalmente, cambiando a nuestro programa en OpenCL y fijando $n = 2^{25}$, obtenemos para los diferentes 3 modos los siguientes resultados:

$size_i$	T. Kernels (ms)	Modo	T. Kernels (ms)	Modo	T. Kernels (ms)	Modo
2	413.357056	Ordenado	411.673344	Random	408.214016	Al revés
4	697.062656	Ordenado	705.459712	Random	703.894272	Al revés
8	883.670528	Ordenado	886.783488	Random	885.721600	Al revés
16	3086.761984	Ordenado	2977.101824	Random	3017.281792	Al revés
32	3936.190464	Ordenado	4099.335168	Random	4102.098688	Al revés
64	4248.546304	Ordenado	4642.184704	Random	4605.424896	Al revés
128	5514.679040	Ordenado	6042.326784	Random	6195.727360	Al revés
256	5712.922368	Ordenado	6282.612992	Random	6945.082368	Al revés
512	3618.371840	Ordenado	4655.715072	Random	6455.350784	Al revés
1024	1534.979840	Ordenado	2117.998080	Random	4842.137856	Al revés
2048	2599.496192	Ordenado	4117.672192	Random	5852.198144	Al revés
4096	4506.530304	Ordenado	10501.937664	Random	17398.194176	Al revés

Figura 14: Tabla con los tiempos del PathMerge en OpenCL para diversos $size_i$

3.7.2. Determinando el valor de $TamBloq$

Ahora comprobaremos cuál es el mejor tamaño de bloque fijando $n = 2^{25}$, $size_i = 2$:

Tam. Bloq.	T. Kernels (ms)	**Modo**	**T. Kernels (ms)**	**Modo**	**T. Kernels (ms)**	**Modo**
2^1	3908.026	Ordenado	3995.444	Random	4116.886	Al revés
2^2	1995.096	Ordenado	2195.446	Random	2204.700	Al revés
2^3	1018.544	Ordenado	1163.696	Random	1160.785	Al revés
2^4	565.954	Ordenado	652.858	Random	638.066	Al revés
2^5	381.790	Ordenado	418.892	Random	422.0736	Al revés
2^6	420.233	Ordenado	432.875	Random	435.434	Al revés
2^7	485.994	Ordenado	495.413	Random	494.809	Al revés
2^8	437.133	Ordenado	448.262	Random	444.021	Al revés
2^9	398.243	Ordenado	410.332	Random	404.191	Al revés
2^{10}	409.318	Ordenado	422.232	Random	417.646	Al revés

Figura 15: Tabla con los tiempos del PathMerge en OpenCL para diversos tamaños de bloque

3.7.3. Comportamiento según él aumenta el tamaño

Una vez ya hemos encontrado los valores óptimos de ambas variables: $size_i = 2^1, Tam.Bloq. = 2^9$, ejecutamos para diferentes tamaños de vectores para ver cuál es la tendencia:

N	**T. Kernels (ms)**	**Modo**	**T. Kernels (ms)**	**Modo**	**T. Kernels (ms)**	**Modo**
2^{15}	0.546816	Ordenado	0.582912	Random	0.564480	Al revés
2^{16}	0.638720	Ordenado	0.688384	Random	0.665088	Al revés
2^{17}	0.976640	Ordenado	1.041920	Random	1.040128	Al revés
2^{18}	1.870848	Ordenado	2.210560	Random	2.230528	Al revés
2^{19}	4.589056	Ordenado	5.011456	Random	4.926464	Al revés
2^{20}	9.714944	Ordenado	10.040832	Random	9.977344	Al revés
2^{21}	20.488960	Ordenado	20.999680	Random	20.993536	Al revés
2^{22}	44.335616	Ordenado	45.234432	Random	45.138176	Al revés
2^{23}	96.932352	Ordenado	87.490304	Random	86.907392	Al revés
2^{24}	182.345216	Ordenado	188.546560	Random	186.270464	Al revés
2^{25}	397.732096	Ordenado	410.244096	Random	403.810304	Al revés
2^{26}	834.599936	Ordenado	872.870912	Random	845.612032	Al revés
2^{27}	1753.564160	Ordenado	1804.399872	Random	1777.910784	Al revés

Figura 16: Tabla con los tiempos del PathMerge en OpenCL para diversos tamaños de N

3.7.4. Comentarios a destacar

- Programar en OpenCL es exageradamente tedioso. Sí que es verdad que da mucha más flexibilidad que CUDA, pero la curva de aprendizaje también es más alta y hay menos recursos en línea (prácticamente solo se ha usado la documentación de Grupo Khronos)
- Una vez ya se tiene el esqueleto del programa, sí que es cierto que hacer ediciones es relativamente sencillo (teniendo en cuenta la complejidad de la API de OpenCL) y ofrece grandes ventajas respecto a CUDA como poder ejecutar de manera heterogénea, con CPU's, GPU's, FPGA's, Aceleradores... El manual de códigos de error fue muy útil durante el desarrollo de este experimento[3] y la documentación de Khronos [4].
- El tamaño óptimo de $size_i$ sigue siendo 2, igual que en CUDA. Esto respalda lo que comentábamos previamente, que es cuando más paralelismo se consigue.
- Por algún extraño motivo, el tiempo de ejecución escala de una forma muy dispar al tiempo en CUDA. Nuestro equipo se ha volcado en intentar solucionar este detalle o posible bug de implementación, pero por falta de conocimiento y tiempo (hemos dedicado más tiempo del que nos gustaría admitir) no hemos podido arreglar el posible problema. Por eso mismo decidimos finalmente no hacer una comparativa de CUDA vs OpenCL como teníamos planeado inicialmente ya que no sabemos si los tiempos obtenidos son correctos.

4. Conclusiones y Futuro trabajo

- Primero de todo hemos de decir que creemos que los valores sacados en OpenCL es posible que estén mal. Realmente no consideramos que debería de haber tanta diferencia entre CUDA y OpenCL. O quizá puede ser que sí por la naturaleza de estos *kernels* de lanzar múltiples instancias cambiando parámetros de posición en el *merge*. Sea como fuere, una futura revisión del código de OpenCL estaría bien. También hemos visto que programar en OpenCL, comparado con CUDA, es muy tedioso.

- Si se quisiera realizar una comparativa entre CUDA y OpenCL con este proyecto, se podría lograr comparando los tiempos que tarda *Insertion Sort* en ejecutarse. Hemos decidido no comparar estos tiempos aparte, ya que no consideramos que sea un kernel lo suficientemente interesante como para tener una comparación en sí.

- Programar en tarjetas gráficas no es siempre la panacea, como hemos podido ver en el primer tipo de *merge*. Algunos algoritmos hay que darles una vuelta antes de paralelizarlos.

- Una idea tan curiosa como la de *PathMerge* ha dado unos rendimientos absolutamente espectaculares, demostrando que la ingeniería de algoritmos es igual de importante que el computador donde se ejecuta.

- La naturaleza del *merge* es usar muchos condicionales haciendo que muchos *threads* no se puedan ejecutar a la vez, sería muy curioso ver el comportamiento de otros algoritmos que usen el mínimo número de condicionales posibles.

- Una optimización para el primer *merge* podría ser que, en cierto punto, pasamos a acabar el *merge* en CPU en vez de hacerlo todo en GPU. En este proyecto no lo hacemos por falta de tiempo, ya que hemos gastado más de $25h$ en la implementación de *PathMerge*.

- Por favor ponednos un 10 o minaremos bitcoins en el Boada (Hasta que tga1009 desaparezca de los confines del Boada).

Referencias

[1] O. Green, R. Mccoll, and D. Bader, "Gpu merge path: a gpu merging algorithm," *Proceedings of the International Conference on Supercomputing*, 06 2012.

[2] NVIDIA, "How to implement performance metrics in cuda c/c++," 2024.

[3] bmount, "Opencl error codes." `https://gist.github.com/bmount/4a7144ce801e5569a0b6`, 2013. GitHub Gist.

[4] T. K. Group, "Khronos group - opencl," 2024.