Heuristic-based Automatic Face Detection

Oscar Mañas Sanchez
Víctor Antón Domínguez

Problema a resolver

El problema que tratamos de resolver es el de detectar caras humanas en una imagen digital. La detección de caras es útil en numerosos campos, y es el primer paso de aplicaciones muy importantes como: reconocimiento facial, interfaz hombre-computador, y vigilancia. El rendimiento de estas aplicaciones está limitado por su capacidad de detectar caras de manera rápida y precisa. Esta tarea es muy costosa ya que cada cara es diferente y pueden variar muchos factores, como la expresión, la postura, las condiciones de iluminación o incluso la capacidad del dispositivo usado para obtener la imágen. Ahora bien, si conseguimos implementar este primer paso de manera eficiente, esto permitirá que otros algoritmos que necesiten este primer paso de detección facial se ejecuten más rápidamente, de modo que podrán dedicar más tiempo a partes más importantes del algoritmo.

Para implementar el algoritmo hemos seguido un artículo científico del año 2003, en el que se explica un método de bajo coste para detectar caras en una imagen. Este método se basa en varios heurísticos para detectar caras en imágenes en escala de grises con fondos complejos. El método se divide en dos etapas; la primera etapa busca posibles regiones de la imagen que podrían ser caras y la segunda etapa determina si una posible cara es realmente una cara usando dos discriminadores. De estos dos discriminadores, nosotros solo hemos implementado el segundo, ya que el paper no especifica detalles de implementación y con la información de la que disponíamos no pudimos llegar a hacer que el primero funcionase correctamente. Usando dos pasos, uno simple y otro complejo, se busca optimizar el uso de los recursos, y hacer que se apliquen sólo allí donde es necesario.

Cabe destacar que el método que presentamos funciona con imágenes en escala de grises y detecta caras en posición frontal y con unas condiciones uniformes de iluminación (sin sombras duras). Ahora bien, la entrada del algoritmo puede ser perfectamente una imagen en color, ya que también hemos implementado la conversión a escala de grises.

Como hemos avanzado, el método usa dos etapas. La primera etapa busca posibles caras en la imágen usando un heurístico sencillo: *En una cara, la intensidad promedio de los ojos es menor que la intensidad de la parte de la nariz que queda entre los ojos.* La segunda etapa toma como entrada la salida de la primera etapa, es decir, toma las regiones con posibles caras y determina si una posible cara es realmente una cara. El discriminador que implementamos se basa en la detección de aristas de la posible cara.

El algoritmo que hemos implementado aplica el método anterior a una región de la imagen (de ahora en adelante window). La idea básica es ir aplicando los diferentes filtros y heurísticos del método explicado anteriormente a la window, de modo que si consigue pasar todas las fases se considerará que ahí hay una cara. De lo contrario, la window se descarta.

Aplicado a una imagen entera, el procedimiento consiste en dividir la imagen en muchas windows (cuantas más, mayor precisión), y aplicar el método a cada una de las windows. También lo podríamos imaginar como una “sliding window”; es decir, tenemos una

window de un determinado tamaño y la vamos desplazando por toda la imagen determinando si lo que hay debajo es una cara. Y esto lo repetimos para cada tamaño de window. Finalmente, dibujamos en color verde todas aquellas windows que han pasado la criba, es decir, que supuestamente contienen una cara.

A continuación, profundizaremos en el funcionamiento de cada etapa del método.

La primera etapa

Los ojos son una parte muy importante de una cara, muestran cada emoción, pero no cambian drásticamente, como sí pasa con la boca. Usamos este principio para centrar toda nuestra atención en los ojos porque tiene una baja variabilidad. Sería difícil hacer una búsqueda de los ojos en la imagen de una cara en alta resolución; eso es más bien una tarea de reconocimiento de objetos que necesita muchos recursos. Por este motivo, es conveniente trabajar con las imágenes de caras en baja resolución, ya que así solo se preserva la información más relevante.

Se propone el siguiente heurístico:

En una cara con iluminación uniforme, la intensidad promedio de los ojos es menor que la intensidad de la parte de la nariz situada entre los ojos.

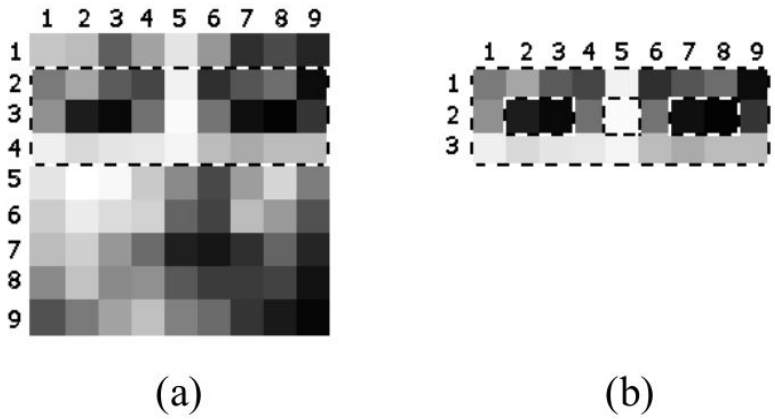
Para aplicar este heurístico, hacemos un reescalado de la window a un tamaño de 9x9 píxeles y a continuación aplicamos un método llamado *histogram equalization* para aumentar el contraste de la window, incrementando así las diferencias de intensidad entre píxeles.

En un tamaño de 9x9 es difícil determinar muchas de las características de la cara, pero este tamaño es suficiente para ubicar los ojos. En este tamaño, es fácil ver que los ojos son de aproximadamente 2x1 píxeles y son más oscuros que la parte de la nariz que queda entre los ojos, correspondiente al píxel central.

Así, la región de los ojos en una window de 9x9 corresponde a las filas 2, 3 y 4. Entonces, la región de los ojos tiene un tamaño de 9x3 (véase la imagen de más abajo). La ecuación para calcular el valor del heurístico de la window, *ENdif*, se muestra a continuación, donde *I* es la intensidad del píxel:

$$ENdif = I_{5,2} - \left(I_{2,2} + I_{3,2} + I_{7,2} + I_{8,2}\right)/4$$

En nuestro caso hemos añadido también la región correspondiente a la boca, ya que comprobamos experimentalmente que esto daba mejores resultados.



(a) Región de la cara de tamaño 9x9 píxeles; y la región de los ojos (encuadrada con una línea discontinua) corresponde a las filas 2, 3 y 4.
(b) Región de los ojos tomada de (a).

Finalmente, si el resultado del heurístico está por debajo de un cierto umbral (determinado experimentalmente), se considera que en aquella región podría haber una cara. Entonces esta window pasa a la segunda etapa.

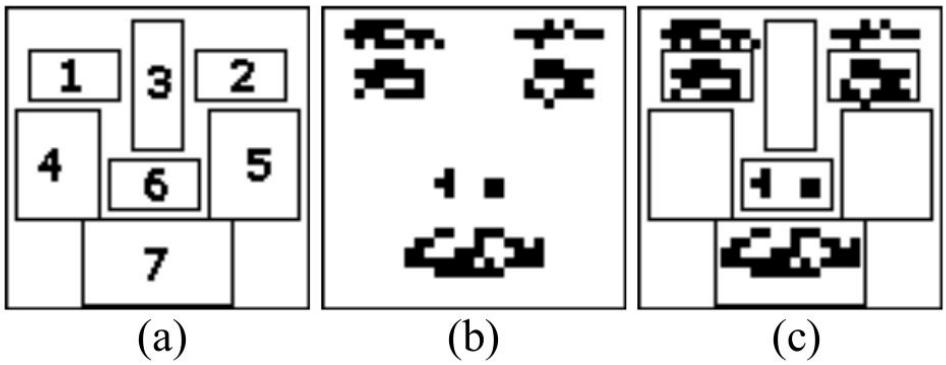
La segunda etapa

Sabemos que una cara tiene ojos, nariz, boca, cejas y que está cubierta por piel, y que algunos elementos de la cara son más oscuros que otros. La idea es aprovechar este invariante para comparar la intensidad en las regiones donde debería haber estos elementos y así determinar si la window realmente contiene una cara. Nótese que esta segunda etapa es parecida a la anterior, la diferencia es que en esta consultamos más píxeles y comparamos más regiones, con lo cual el coste es mayor pero también es más precisa. La buena noticia es que aplicaremos esta segunda etapa a muchas menos regiones que la primera etapa, pues el primer heurístico ya habrá descartado algunas windows donde podemos estar seguros de que no hay una cara.

El primer paso de esta segunda etapa es segmentar la imagen usando el conocido método de Sobel para encontrar aristas en dirección horizontal. La imagen resultante contendrá solo los ojos, cejas, boca y nariz. Usamos una máscara simple para evaluar la presencia o ausencia de estos elementos (véase la imagen de más abajo).

La imagen se reescala a 30x30 píxeles, que es el tamaño de la máscara. La máscara se divide en 7 regiones correspondientes a los elementos de la cara. Las regiones 1 y 2 corresponden a los ojos, 3 y 6 corresponden a la nariz, 4 y 5 corresponden a las mejillas y la 7 corresponde a la boca. Para cada región, se calcula la intensidad promedio de los píxeles contenidos en la región. En el caso de 1, 2, 6 y 7 el valor esperado debería estar cerca del 0 (intensidad baja, zona oscura), y en el caso de 3, 4 y 5 el valor esperado debería estar cerca del 255 (intensidad alta, zona brillante). También se evalúa la

simetría entre las regiones 1 y 2, y las regiones 4 y 5, comparando sus valores representativos.



(a) La máscara usada en la segunda etapa.
(b) Resultado de la región de 30x30 píxeles después de aplicar detección de aristas.
(c) Usando la máscara para evaluar si la posible cara es realmente una cara.

Así pues, para cada región se calcula la diferencia con el valor esperado y se acumulan las diferencias de cada región. Si la diferencia total está por debajo de un cierto umbral (determinado experimentalmente), se considera que la window contiene una cara.

Cabe destacar que en el paper no se especifican los píxeles que contenía cada región, así que lo hemos tenido que calcular aproximadamente a partir de la imagen anterior.

Cuando todo el proceso termina, normalmente se detecta una cara con windows de diferentes tamaños, pero solo se necesita una window por cara. Una ampliación del algoritmo consistiría en hacer clustering de las detecciones solapadas, y dejar aquella window con mejores valores en los heurísticos. Esto se aplicaría a cada cara.

Versiones de código CUDA

Como el algoritmo que hemos implementado es bastante complejo y tiene bastantes pasos y funciones auxiliares a implementar, en un primer momento decidimos centrarnos en hacerlo funcionar todo en secuencial.

Una vez estuvimos satisfechos con los resultados en secuencial, fuimos paralelizando una a una las funciones del kernel principal hasta conseguir la versión totalmente paralelizada. Además hicimos una versión genérica para poder especificar con *defines* el número de devices a usar y si se quiere usar memoria pinned o no. De este modo nos ahorramos duplicar código y hacer varias versiones fue mucho más sencillo.

El primer *define* es *NUM_DEVICES*, que indica el número de GPUs que se van a usar. El segundo es *USE_PINNED*, que indica si se quiere usar *pinned memory* o no. Como hemos dicho, esto nos evita tener que duplicar código pero además cualquier corrección en una versión se aplica a todas automáticamente.

Cuando pasamos a usar 4 GPUs, vimos que no conseguíamos una mejora respecto a una sola GPU. Entonces cambiamos la manera de recuperar los resultados de las GPUs para que esto no se convirtiera en un cuello de botella.

Así, creamos las siguientes versiones del código:

1. face_detect_seq (versión secuencial)
2. face_detect_1gpu (una GPU sin usar pinned memory)
3. face_detect_4gpu_v1 (cuatro GPUs sin usar pinned memory)
4. face_detect_4gpu_v2 (transferencia de memoria mejorada)
5. face_detect_4gpu_pin_v1 (cuatro GPUs usando pinned memory)
6. face_detect_4gpu_pin_v2 (transferencia de memoria mejorada)

A continuación se describen en detalle cada una de las versiones.

face_detection_seq.cpp:

Como ya hemos comentado, esta versión fue la primera que implementamos. En ella no utilizamos la librería CUDA en ningún momento, pues nuestro objetivo era únicamente conseguir que nuestro programa detectara caras en la imagen, sin que esto fuera necesariamente eficiente.

Para abrir y guardar las imágenes, usamos la librería stbi. El resto de funcionalidades las hemos implementado nosotros mismos. Algunos ejemplos son:

1. Reescalado de una imagen (*resize*).
2. Obtención del histograma de una imagen (*getHistogram*).
3. Ecualización del histograma de una imagen (*histogramEqualization*).
4. Filtro Sobel de detección de aristas (*sobelEdgeDetection*).
5. Conversión de una imagen a escala de grises(*getGrayScale*).
6. Conversión de una imagen a blanco y negro (*toBlackAndWhite*).
7. Obtención de los heurísticos de acuerdo con el paper.
8. Guardar una imagen resaltando las regiones donde se han detectado caras (marco verde).

Después de conseguir que todos estos algoritmos y pasos funcionaran, escogimos varias imágenes de prueba para poder calibrar los heurísticos obteniendo los valores óptimos de los umbrales.

Una vez satisfechos con los resultados, pasamos a implementar la versión paralela con CUDA.

face_detect_1gpu:

Para implementar la versión la versión CUDA, decidimos hacerlo de una forma muy incremental. Es decir, para cada pequeña parte del kernel principal que modificábamos para paralelizarla, comprobábamos que los resultados fueran los esperados, es decir, los mismos que la versión secuencial. Esto nos permitió tener un mayor control sobre el código y nos facilitó la detección de errores.

La estrategia de descomposición de trabajo en bloques y threads que diseñamos fue la siguiente: cada *thread block* se encarga de procesar una window de la imagen, y los 1024 threads de cada bloque se reparten el trabajo de procesar cada window.

Decidimos redimensionar la imagen de entrada a un tamaño fijo de 1024x1024 píxeles con tal de hacer el código más sencillo. Como el máximo número de bloques disponibles es 65535 podemos usar 256 bloques en cada dimensión, y como cada bloque procesa una window, sabemos que estas están separadas entre ellas 4px (1024 px / 256 windows por dimensión). Gracias a las pruebas que hicimos en secuencial sabemos que esta precisión es más que suficiente para detectar bien las caras.

Por lo tanto, ocupamos todos los bloques (65535) y threads por bloque (1024) permitidos por nuestro nodo CUDA. Aún así, no todos los threads están todo el rato trabajando, ya que hay partes del kernel que son secuenciales y solo las puede ejecutar el primer thread del bloque.

Redimensionar la imagen implica que tengamos que adaptar las windows a la nueva relación de aspecto antes de ejecutar el kernel; al obtener los resultados, debemos deshacer esta transformación para poder dibujar el resultado final, con las windows con su relación de aspecto original.

Así, poco a poco fuimos paralelizando cada función del kernel. Durante este proceso, las partes que aún no habíamos paralelizado dejábamos que las hiciera sólo el thread 0, para simular una ejecución secuencial.

Cabe decir que la manera de recoger los resultados en la versión secuencial y paralela es diferente. Mientras que en la versión secuencial vamos guardando las caras detectadas en una estructura de resultados, en la versión CUDA no era eficiente hacerlo de este modo, ya que esto implicaría muchas transferencias entre GPU y CPU. Para evitarlo, decidimos usar una matriz de resultados por cada tamaño de window a procesar. Así, por cada window el kernel escribe un 1 en la celda correspondiente de la matriz de resultados si se ha detectado una cara, o un 0 en su defecto. Después de la ejecución del kernel, transferimos la matriz de resultados del device al host y así guardamos el resultado para dibujarlo posteriormente en la imagen final.

Cuando conseguimos que esto funcionase para un tamaño fijo de window, decidimos extenderlo a varios tamaños. Inicialmente estos tamaños eran cuadrados, pero más adelante, y con el objetivo de aumentar la carga de trabajo, hicimos que los tamaños pudieran tener una relación de aspecto no cuadrada haciendo el producto cartesiano de una serie de alturas y anchuras. Es importante notar que tenemos que crear en el host tantas matrices de resultado como tamaños de windows se van a computar.

face_detect_4gpu_v1:

Más adelante, decidimos exprimir todo el potencial que nos proporciona el nodo de CUDA y usar las 4 *Tesla K40* a la vez. Como la cantidad de tamaños de window a computar es muy grande, cada gráfica puede procesar un subconjunto de tamaños de window en paralelo.

En esta primera versión, decidimos reservar solo una matriz de resultados en cada device. Para ello, tuvimos que añadir los correspondientes *cudaSetDevice* y cambiar las llamadas de copiar memoria entre host y device por sus versiones asíncronas.

Entonces, al terminar de computar un tamaño de window, cada device envía la matriz de resultados de vuelta al host. Esto implica tener que hacer una transferencia *DeviceToHost* por cada tamaño de window, es decir, 400 transferencias en total.

face_detect_4gpu_v2:

En esta versión decidimos cambiar la manera de transferir los resultados del device al host para ver si esto introducía alguna mejora.

Si en la primera versión solo reservábamos una matriz de resultados en cada device y hacíamos una transferencia *DeviceToHost* por cada tamaño de window computado, en esta versión reservamos en cada device la memoria necesaria para almacenar todas las

matrices de resultados de los tamaños de window que calcula el device. Esto nos permite hacer una única transferencia *DevideToHost* cuando el device ha acabado de computar todos los tamaños de window asignados. Es decir, en total se realizan solo 4 transferencias *DeviceToHost*, pero cada una envía muchos más bytes que en la versión anterior.

Cabe destacar que solo veremos una mejora de rendimiento respecto a la versión anterior si hay un *overhead* considerable por cada transferencia *DeviceToHost*, ya que el número total de bytes a transferir es el mismo en ambas versiones. En la siguiente sección veremos si realmente es así.

face_detect_4gpu_pin_v1:

Añadimos el uso de la memoria pinned, que simplemente consistió en cambiar las llamadas a *cudaMalloc* por llamadas a *cudaMallocHost*.

face_detect_4gpu_pin_v2:

Nuevamente, añadimos el uso de memoria pinned tal y como hemos hecho con la versión *face_detect_4gpu_pin_v1*.

Vale la pena mencionar que en esta versión estamos reservando una cantidad mucho mayor de memoria pinned, ya que ahora reservamos en cada device la memoria necesaria para almacenar todas las matrices de resultados de los tamaños de window asignados al device.

Rendimiento de la aplicación

A continuación analizaremos el rendimiento de nuestra aplicación con cada una de las versiones. Debido a la naturaleza de nuestra aplicación, no nos es posible tomar como medida de rendimiento los GFLOPS ni el ancho de banda. Esto es porque es muy difícil calcular cuántas operaciones de coma flotante se realizan en una ejecución. Por lo tanto, usaremos el tiempo de ejecución como medida de rendimiento para evaluar las mejoras introducidas.

Decidimos fijar el número de tamaños diferentes de window a 400. Es un número bastante elevado, pero nos servirá para acentuar las mejoras de rendimiento en las diferentes versiones. Para cada versión del código ejecutamos la aplicación con 2 imágenes diferentes, y para cada imágen realizamos 3 ejecuciones, de las cuales nos quedamos con el mínimo tiempo de ejecución. A continuación se muestran los resultados (en segundos) de las diferentes pruebas:

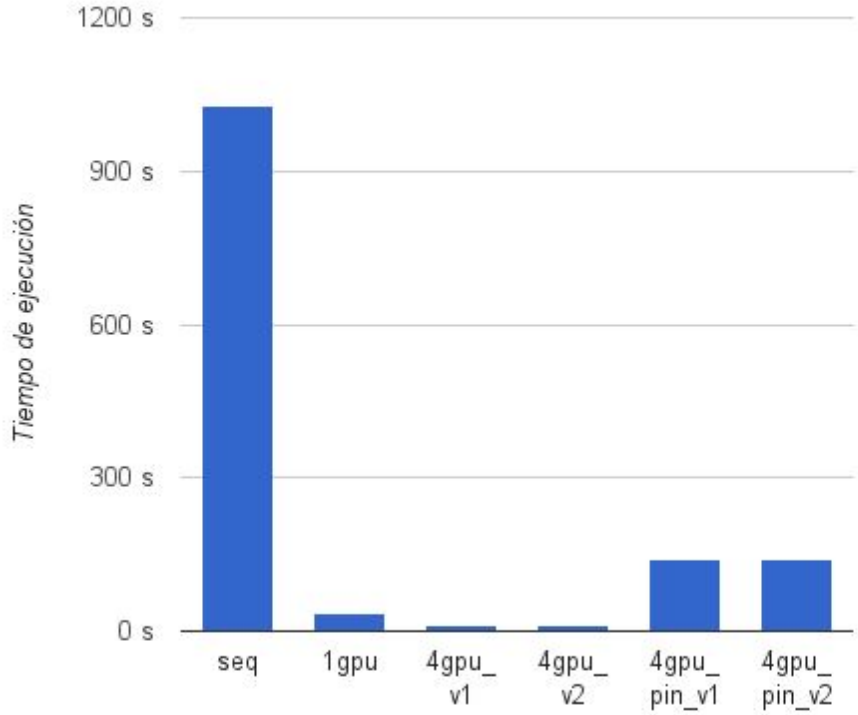
	img1_kernel	img1_tx	img1_total	img2_kernel	img2_tx	img2_total
seq	1.010,90	0,00	1.010,90	1.045,16	0,00	1.045,16
1gpu	35,33	0,05	35,38	36,56	0,05	36,61
4gpu_v1	9,69	0,05	9,74	10,02	0,05	10,07
4gpu_v2	9,64	0,02	9,66	9,97	0,02	9,99
4gpu_pin_v1	98,57	41,26	139,83	100,65	41,76	142,41
4gpu_pin_v2	98,50	40,75	139,25	100,50	41,50	142,00

Tabla de tiempos obtenidos en las diferentes pruebas. Cada fila representa una versión del programa, y cada columna un tiempo para cada imagen (tiempo kernel, tiempo transferencias, tiempo total).

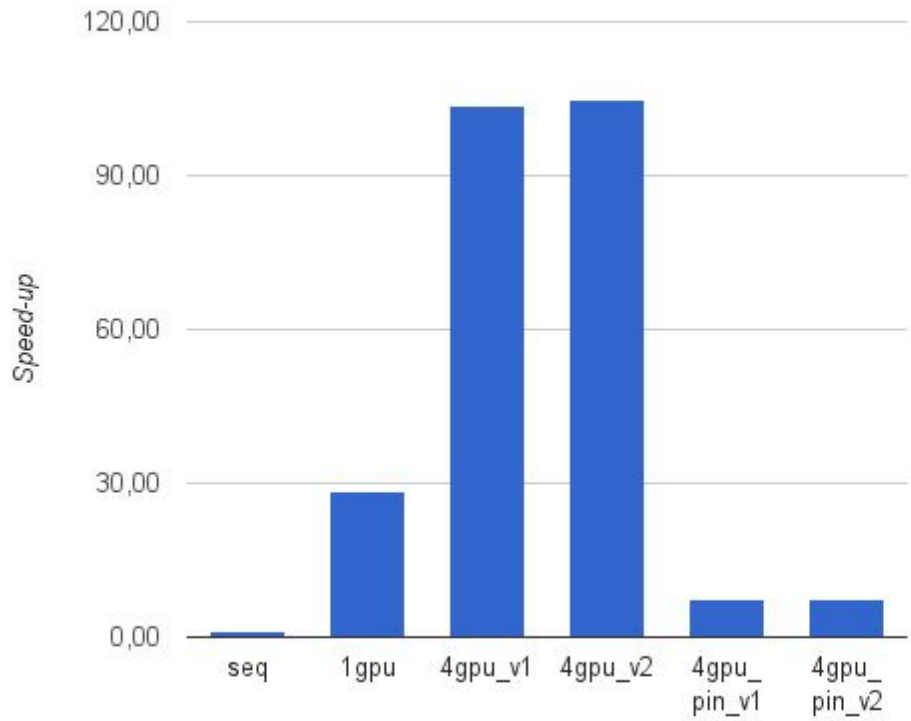
	Tiempo promedio	Speed-up respecto a la versión secuencial	Speed-up respecto a la versión anterior
seq	1.028,03	1,00	1,00
1gpu	35,99	28,56	28,56
4gpu_v1	9,91	3,63	103,78
4gpu_v2	9,82	14,36	104,64
4gpu_pin_v1	141,12	0,07	7,28
4gpu_pin_v2	140,63	0,07	7,31

Tabla de tiempo promedio para cada versión, el speed-up respecto a la versión secuencial, y el speed-up respecto a la versión anterior, para ver el incremento (o decremento) de rendimiento en cada versión.

Tiempo de ejecución de cada versión:



Speed-up de cada versión respecto a la versión secuencial:



face_detect_seq:

Esta es la versión del programa sin paralelizar. Tarda unos 20 minutos en detectar caras en una imagen para los 400 tamaños de window especificados. Sin embargo, muchas operaciones del programa se pueden paralelizar de manera independiente. Es aquí donde se hace útil el uso de CUDA y las GPUs. Como veremos, las mejoras de rendimiento obtenidas con la paralelización son muy significativas.

face_detect_1gpu:

Aunque usamos solo una GPU, esta versión ya supone una mejora abismal respecto a la versión secuencial (del orden de 33x). Esto es porque ahora todo el cómputo de comprobar si hay una cara en la imagen para un tamaño de window determinado se hace en paralelo. Pero aún tenemos la secuencialización de todos los tamaños de window, lo cual arreglamos en las siguientes versiones con el uso de las 4 GPUs.

face_detect_4gpu_v1:

En esta versión repartimos los tamaños de window a computar entre 4 GPUs, de modo que cada tarjeta recibe una cuarta parte de los tamaños. Después de realizar las ejecuciones, obtenemos un speedup de 3,63 (casi 4!) respecto a la versión con una sola GPU, y un speedup de 103,78 respecto a la versión secuencial. Esto muestra que la versión CUDA de nuestra aplicación tiene una gran escalabilidad, ya que usando *n* devices obtenemos un speedup de casi *n*.

face_detect_4gpu_v2:

Después de realizar las ejecuciones, vemos que la nueva manera de transferir los resultados de device a host mejora ligeramente el tiempo de ejecución respecto a la versión anterior, pero no demasiado. Tal y como comentábamos en la sección anterior, solo veremos una mejora de rendimiento si hay un *overhead* considerable por cada transferencia *DeviceToHost*. Si observamos los resultados, el tiempo de transferencia se reduce a la mitad, por lo que podemos concluir que el *overhead* implícito en cada transferencia es importante. A pesar de esto, el tiempo de transferencia es del orden de milisegundos, negligible en comparación con el tiempo de ejecución del kernel, por lo que la mejora en rendimiento es casi inapreciable.

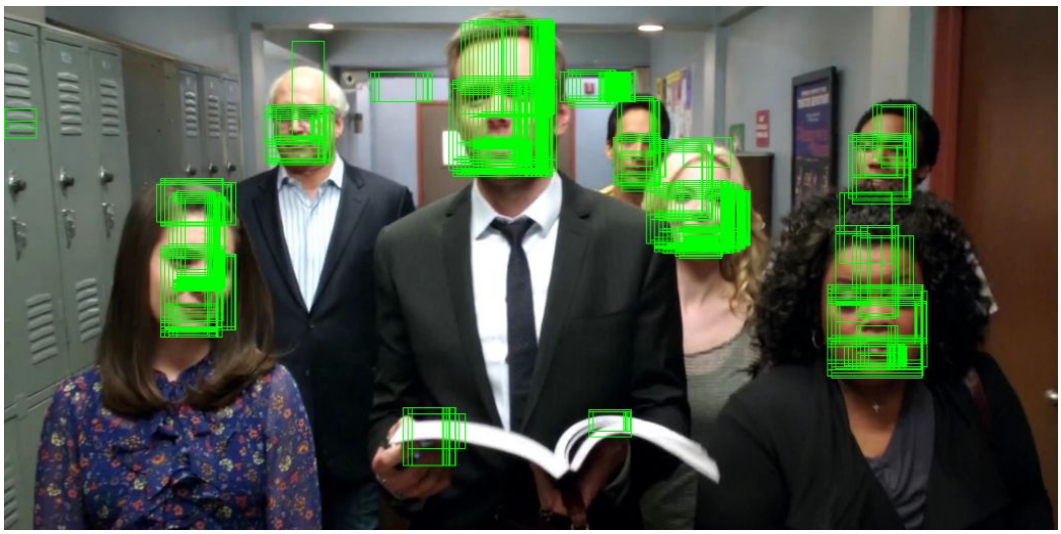
face_detect_4gpu_pin_v1:

Después de realizar las ejecuciones, observamos que el rendimiento de la aplicación usando *pinned memory* es mucho peor que sin usarla (de hecho, es peor aún que con una sola GPU). Investigando un poco, creemos que esto se puede deber a que hacemos un sobreuso de la *pinned memory*, ya que se realizan muchas transferencias *DeviceToHost* cuando se recogen los resultados de cada uno de los tamaños de window, y estas transferencias son de muy pocos bytes, con lo que no sale a cuenta usar *pinned memory*.

face_detect_4gpu_pin_v2:

Nuevamente, vemos que el rendimiento de esta segunda versión de la aplicación usando *pinned memory* es mucho peor que sin usarla. Esta vez, sin embargo, creemos que se puede deber al hecho de que reservar *pinned memory* es una operación muy costosa, y con la nueva manera de transferir los resultados de device a host tenemos que reservar una gran cantidad de memoria en el device, ya que hacemos solo una transferencia por device una vez ha computado todos los tamaños de window asignados.

Anexo de resultados:




CBS
CBS
CBS


HBO

Referencias

Artículo en el que nos hemos basado para implementar el algoritmo de detección de caras en una imagen:
https://www.researchgate.net/profile/Olac_Fuentes/publication/221463049_Heuristic-based_Automatic_Face_Detection/links/0f3175327255e58d38000000.pdf

Algoritmo para implementar la ecualización de histograma de una imagen:
http://www.math.uci.edu/icamp/courses/math77c/demos/hist_eq.pdf

Pinned memory y razón por la que creemos que baja tanto nuestro rendimiento usándola:
http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#pinned-memory
https://www.cs.virginia.edu/~mwb7w/cuda_support/pinned_tradeoff.html