Programación de múltiples dispositivos heterogéneos: El caso de Optical Flow

Sergio Alonso Pascual[1], Arturo Gonzalez-Escribano[2]

Resumen— **Los sistemas paralelos actuales son cada vez más heterogéneos, con dispositivos de diferentes tipos y capacidades de cómputo. Explotar múltiples dispositivos diferentes para una misma aplicación sigue siendo un reto donde intervienen desde problemas técnicos relacionados con sincronizar y comunicar diferentes dispositivos hasta problemas de reparto de carga y flexibilidad para ajustar el cómputo a los recursos de la plataforma. En este trabajo estudiamos la programación y adaptación a plataformas heterogéneas de HSOpticalFlow, una aplicación de streaming orientada a estimar el movimiento aparente de objetos en una secuencia de imágenes. Partiendo del código original en CUDA, presentamos la metodología para implementarlo en forma de pipeline entre múltiples dispositivos utilizando el modelo de programación Controller, y para introducir un mecanismo de reparto de carga que permite el ajuste cuando las capacidades de cómputo son distintas. Esto permite no solo construir soluciones paralelas muy eficientes entre dispositivos similares, sino también aprovechar dispositivos de poca capacidad de cómputo para aliviar la carga y aumentar la productividad de dispositivos mucho más potentes. Presentamos resultados de un estudio experimental utilizando varias GPUs de NVIDIA de diferentes arquitecturas y generaciones que muestran que nuestra solución permite explotar de forma combinada varios dispositivos para reducir los tiempos de ejecución y conseguir un mejor ratio de fotogramas por segundo. En concreto, los resultados muestran aceleraciones de 2x utilizando dos GPUs NVIDIA V100 y hasta 1,37x con una GPU NVIDIA A100 y tres GPUs Titan Black, que son aproximadamente 8 veces más lentas para esta aplicación.**

Palabras clave— **Programación heterogénea, Streaming, Equilibrio de carga, Flujo óptico**

I. Introducción

ES cada vez más frecuente encontrar sistemas paralelos con un alto nivel de heterogeneidad. Tanto en grandes plataformas de cómputo como en sistemas integrados on-chip. Aunque disponemos cada vez de un mayor número de herramientas y propuestas para programar y manejar dispositivos heterogéneos desde un único código fuente (por ejemplo SYCL [1], Controller [2], o OpenH [3]), en la mayoría de los casos aún no están lo suficientemente maduras para explotar todo el potencial de estos sistemas con facilidad. Aún hay cuestiones complejas relacionadas con el movimiento eficiente y transparente de datos entre dispositivos de diferentes arquitecturas, la sincronización, el solapamiento de cómputo y transferencias de dato, y carencias importantes en el reparto de carga y asignación de recursos para ajustar el cómputo a la plataforma de ejecución. En general los modelos actuales nos aportan herramientas de programación de propósito general, que deben ser utilizadas por el programador para construir sus soluciones a los problemas de reparto y equilibrio de carga teniendo en cuenta la estructura de la aplicación y la plataforma objetivo.

Un tipo de aplicaciones especialmente interesante y exigente para explotar sistemas altamente heterogéneos son las aplicaciones de *streaming* o flujo de datos. Estas aplicaciones se caracterizan por recibir como entrada un *stream*, un canal por el que se reciben datos, que generalmente representan múltiples instancias de una estructura del mismo tipo. Por ejemplo, los frames o fotogramas de un flujo de vídeo. Estas aplicaciones realizan una serie de tareas sobre cada instancia de datos, que muchas veces tiene diferente granularidad o necesidades de cómputo. Adaptar estas aplicaciones a plataformas heterogéneas aprovechando sus recursos puede resultar complejo.

Hemos escogido como caso de estudio y motivación la aplicación HSOpticalFlow [4]. Se trata de una aplicación de streaming orientada a estimar el movimiento aparente de objetos en una secuencia de imágenes. Esta aplicación está desarrollada en CUDA y está incluida en los ejemplos de dominio específico suministrados con el toolkit de desarrollo [5]. HSOpticalFlow presenta un ejemplo sencillo pero característico de la estructura de aplicaciones basadas en ILS (Iterative Loop Stencil) multinivel, también conocidos como métodos *multi-grid* [6], [7]. Esta clase engloba todo un conjunto de aplicaciones científicas basadas en métodos finitos y resolución de ecuaciones diferenciales parciales con métodos iterativos en varios niveles, que presentan estructuras similares a HSOpticalFlow, o con recorridos de niveles más complejos. En concreto lo que se conoce como ciclos V o W, donde el flujo sube y baja a través de los niveles para conseguir una mayor precisión con una convergencia más rápida. En cada nivel se trabaja con estructuras de datos más y más grandes con un nivel de detalle más fino. Por tanto la carga de trabajo de las tareas depende del nivel. HSOpticalFlow es un buen ejemplo y caso de estudio para investigar el reparto de carga y la adaptación a sistemas heterogéneos de esta clase de programas aplicados a una secuencia de entradas.

En este trabajo presentamos la metodología y técnicas necesarias para introducir en aplicaciones de streaming y métodos multi-grid (como HSOpticalFlow) un mecanismo flexible de reparto y equilibrio de carga en plataformas heterogéneas con dispositivos de diferentes capacidades de cómputo. La solución está diseñada para integrarse con facilidad

[1]Dpto. de Informática, Universidad de Valladolid, e-mail: sergio.alonso.pascual@uva.es

[2]Dpto. de Informática, Universidad de Valladolid, e-mail: arturo@infor.uva.es

en la estructura de códigos de streaming ya existentes, explotando las posibilidades de solapamiento de cómputos y movimientos de memoria propios del pipeline paralelo típico en este tipo de aplicaciones. El mecanismo propuesto permite al programador decidir las etapas y carga que asigna a cada dispositivo simplemente cambiando valores en una matriz de asignación de recursos. La comunicación entre dispositivos y el solapamiento de cómputo y comunicación se realizan de forma transparente. La solución propuesta permite equilibrar la carga de trabajo no sólo en sistemas con dispositivos similares, sino también aprovechar dispositivos de poca capacidad de cómputo para aliviar la carga de dispositivos mucho más potentes, aumentando la productividad y el ratio de entradas por segundo procesadas.

Presentamos un estudio experimental comparando nuestra implementación prototipo con la versión de referencia implementada directamente en CUDA. Se realizan comparaciones en una plataforma con dos GPUs iguales y en otra con GPUs de diferentes generaciones y capacidades de cómputo. Los resultados muestran en concreto mejoras sobre la aplicación de referencia de 2x utilizando dos GPUs NVIDIA V100 y hasta 1,37x con una GPU NVIDIA A100 y tres GPUs Titan Black, que son aproximadamente 8 veces más lentas para esta aplicación, obteniendo eficiencias superiores al 90 % en la explotación del paralelismo entre dispositivos.

El resto del trabajo se organiza de la siguiente forma. La sección 2 comenta trabajo relacionado. La sección 3 presenta la propuesta y la implementación de un prototipo usando el caso de estudio HSOpticalFlow. En la sección 4 se describe el estudio experimental desarrollado y se discuten los resultados del mismo. La sección 5 presenta las conclusiones y posible trabajo futuro.

II. Background y trabajo relacionado

Existen en la actualidad diversas propuestas de modelos de programación paralela heterogénea de alto nivel. Estos modelos intentan facilitar la programación de aplicaciones portables a diferentes tipos de dispositivos y simplificar la utilización combinada de varios de ellos. Algunas propuestas parten de la idea de utilizar un código único que se compila y adapta a diferentes plataformas. Por ejemplo, propuestas como el estándar SYCL [1] se están consolidando gracias a la evolución de los compiladores que lo implementan, como AdaptiveCpp [8] o el lenguaje DPC++ integrado en Intel oneAPI [9]. Sin embargo, estos compiladores no ofrecen aún un soporte completo del estándar y la eficiencia de los mecanismos utilizados, especialmente los relacionados con la migración o interoperabilidad entre dispositivos de diferente naturaleza aún no está garantizada. Un caso parecido lo presentan las implementaciones de las últimas versiones del estándar OpenMP, que en general necesitan extensiones conceptuales y nuevos sistemas de ejecución específicos para sistemas heterogéneos [10]. Otras propuestas menos conocidas o del ámbito académico incluyen por ejemplo el modelo Controller [2]. Todos estos modelos proveen de las herramientas necesarias para programación de propósito general y soportan múltiples dispositivos heterogéneos. Sin embargo, es responsabilidad del programador analizar la estructura de sus aplicaciones y generar el código adecuado para repartir y equilibrar la carga, adaptándola a un sistema con diferentes tipos de dispositivos, en especial con diferentes capacidades de cómputo.

Algunos modelos, como FastFlow [11] o SkePU [12], se focalizan en generar desde expresiones de alto nivel el código para aplicaciones con diferentes patrones paralelos. Algunos de estos modelos, como FastFlow, incluyen patrones basados en la estructura de pipeline y streaming. Sin embargo, el reparto y equilibrio de carga para estructuras de tipo pipeline multinivel sobre dispositivos heterogéneos de diversa naturaleza sigue siendo problemático.

Algunos modelos de programación heterogénea, como Sigmoid [13], OpenH [3] o las extensiones anteriormente comentadas para OpenMP [10], incluyen soluciones para el reparto y equilibrio de carga integrados en el propio modelo o mecanismo de ejecución. En general, estas soluciones genéricas están orientadas a equilibrar la carga partiendo el cómputo en subtareas y repartiéndolas de forma dinámica bajo demanda con en esquema maestro-trabajadores o granja de tareas. En muchos casos, la sobrecarga del sistema dinámico de reparto y el trasiego de datos asociados a cada subtarea puede ser menos eficiente que una solución adaptada a la aplicación o plataforma. Especialmente en el caso de aplicaciones de streaming donde existe una estructura de ejecución y dependencias preestablecida y conocida.

Uno de los objetivos de este trabajo es que la solución propuesta pueda integrarse con facilidad en la estructura de códigos de streaming ya existentes, utilizando tareas y transferencias con la misma granularidad del código original y explotando las posibilidades de solapamiento de cómputos y movimientos de memoria propios de la estructura de pipeline paralelo típico en estas aplicaciones. Para ello nos apoyaremos en un modelo de programación heterogénea de propósito general, en concreto Controller. Este modelo ofrece funcionalidades parecidas a SYCL, pero utiliza un mecanismo para integrar directamente kernels escritos en los modelos de programación de más bajo nivel suministrados por el vendedor de los dispositivos, cómo por ejemplo CUDA, OpenCL, Hip, u OpenMP. El modelo incluye un sistema para detectar en tiempo de ejecución las dependencias entre tareas y realizar de forma transparente las transferencias de memoria entre dispositivos necesarias para mantener la coherencia. Su sistema de ejecución incluye un mecanismo muy eficiente de gestión de las sincronizaciones y transferencias de memoria entre dispositivos de diferente naturaleza [2], [14]. Controller es un sistema de programación muy adecuado para construir sobre sus funcionalidades mecanismos de reparto y equilibrio de carga, que pueden integrar-

Listado 1: Pseudocódigo del algoritmo usado para calcular el HSOpticalFlow. Las llamadas a funciones equivalen a lanzamientos de kernel (salvo swap). Los argumentos en azul son de entrada y los rojos de salida.

```
// Crear versiones de menor resolución de ambas imagenes (src y tgt)
for(lvl = nlvls - 1; lvl > 0; lvl--){
  Downscale(src[lvl], src[lvl-1]);
  Downscale(tgt[lvl], tgt[lvl-1]);
}
// La estimación inicial (u, v) comienza en 0
for(lvl = 0; lvl < nlvls; lvl++){
  for(warp = 0, warp < nwarps; warp++){
    // Distorsionar imagen objetivo en base a estimación actual (u, v)
    WarpImage(tgt[lvl], u, v, dist);
    // Calcular matrices de la ecuación a resolver
    ComputeDerivatives(src[lvl], dist, Ix, Iy, Iz);
    // Resolver la ecuación para du, dv
    for(i = 0; i < nsolves; i++){
      JacobiSolve(du0, dv0, Ix, Iy, Iz, du1, dv1);
      swap(du0, du1)
      swap(dv0, dv1)
    }
    // Actualizar estimación actual
    add(u, du0, u);
    add(v, dv0, v);
  }
  if(lvl < nlvls - 1){
    // Escalar solución (u, v) para usar en el siguiente nivel
    Upscale(u, nu);
    Upscale(v, nv);
    swap(u, nu);
    swap(v, nv);
  }
}
```

se en las aplicaciones de partida con poco esfuerzo de desarrollo.

III. Propuesta de solución

Esta sección describe la propuesta para incluir una abstracción simple y extrapolable a otros casos que permita al programador escoger cómo se realiza en un sistema heterogéneo el reparto de las tareas de una aplicación de streaming en la que cada instancia en el flujo de datos se procesa con métodos iterativos, potencialmente con múltiples kernels y niveles de iteración.

A. Caso de estudio

En este trabajo hemos escogido como caso de estudio y motivación la aplicación HSOpticalFlow, incluída en los ejemplos de dominio específico suministrados con el toolkit de desarrollo de CUDA. Es en una implementación de un método de flujo óptico en 2D conocido como Hierarchical Horn and Schunck [4]. En el listado 1 se muestra el pseudocódigo del algoritmo aplicado por HSOpticalFlow.

Este algoritmo minimiza una función de energía mediante una aproximación de diferencias finitas de la ecuación de Euler-Lagrange correspondiente a este problema específico. Emplea una estrategia multinivel, de grano grueso a fino, calculando primero la solución para versiones de menor resolución de las imágenes y luego escalando sucesivamente la solución obtenida en cada nivel para ser usada como punto de partida en el cálculo del siguiente, con imágenes de mayor resolución. Para cada uno de estos niveles de

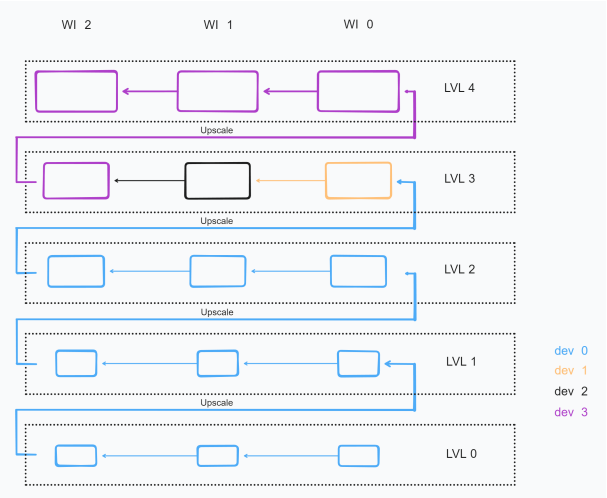

Fig. 1: Diagrama de flujo de HSOpticalFlow.

resolución se realizan lo que en la terminología del problema se denominan *iteraciones de warp*. En cada una de ellas partimos de una estimación inicial de la solución y calculamos el incremento necesario para mejorar esta estimación. El cálculo consiste en un número fijo de iteraciones de un método de Jacobi, implementado como un ILS (Iterative Loop Stencil). Se aplica de forma paralela en cada punto un cálculo que depende de los valores anteriores en los puntos vecinos.

En la figura 1 se puede ver una representación del flujo de tareas para comparar un par de fotogramas, utilizando cinco niveles de resolución y tres iteraciones de warp en cada nivel. En el caso de la implementación de referencia, las iteraciones de Jacobi que realiza cada iteración de warp se puede configurar como argumento del programa. El cómputo comienza en el nivel de resolución más bajo (nivel 0), donde realiza de forma encadenada las iteraciones de warp con ese nivel de resolución. Se utiliza una función *prolongation* o *upscale* para generar, a partir del resultado de la última iteración de warp, una imagen de mayor resolución. Esta última es la entrada de la primera iteración de warp del siguiente nivel.

B. Reparto de carga

La estructura iterativa y multinivel de HSOpticalFlow, cuando trabaja sobre una secuencia de imágenes, proporciona oportunidades para repartir la carga de trabajo de diferentes formas entre varios dispositivos.

La secuencia de imágenes crea un flujo de datos continuo o *pipeline* a través de la estructura de niveles e iteraciones de warp de HSOpticalFlow. Esta estructura de pipeline puede distribuirse entre varios dispositivos, repartiendo el trabajo a realizar para cada par de fotogramas entre ellos. De esta forma, cuando un dispositivo termina con su parte le pasa el resultado al siguiente dispositivo y puede comenzar con su parte de cálculo para el siguiente par de fotogramas. Un adecuado reparto de la estructura del pipeline permite además que el cómputo y los movimientos de las estructuras de datos entre dispo-

Listado 2: Ejemplo de la matriz de reparto de carga correspondiente a la asignación mostrada en la figura 1.

```
int lw2cid[nlvls][nwarps] = {
  {1, 1, 1},
  {1, 1, 1},
  {1, 1, 1},
  {2, 3, 0},
  {0, 0, 0}};
```

Listado 3: Pseudocódigo del pipeline de fotogramas usado. Las llamadas a funciones equivalen a lanzamientos de kernel (salvo swap). Los argumentos en azul son de entrada y los rojos de salida.

```
tgt = loadFrame( 0 );
for(i = 1; i < nFrames; i++){
  src = tgt;
  tgt = loadFrame( i );
  ComputeFlow(src, tgt, u, v, ...);
  Norm(u, v);
}
```

sitivos se solapen usando comunicaciones asíncronas. De esta forma, una vez que el pipeline está lleno las latencias de los movimientos de datos pueden quedar parcial o totalmente ocultas. En la figura 1 se muestra un posible ejemplo de reparto de carga asignando diferentes iteraciones de warp de diferentes niveles a cada dispositivo. Cada nivel tiene una carga de trabajo diferente y creciente con el índice del nivel.

La decisión más importante para esta aproximación es elegir los puntos de corte en los que se cambia de un dispositivo a otro. Para ello hay básicamente tres opciones que implican una creciente complejidad de implementación pero también una granularidad más fina para el control del equilibrio de carga entre dispositivos.

1. Partir sólo entre niveles.
2. Partir entre iteraciones de warp en cualquier nivel.
3. Partir entre iteraciones de jacobi dentro de una iteración de warp.

Nuestras pruebas experimentales preliminares indican que repartiendo niveles enteros entre dispositivos de diferentes capacidades de cómputo la granularidad es demasiado gruesa y el desequilibrio entre niveles no permite conseguir un buen equilibrio de carga en muchas situaciones. Por otra parte, la complejidad de programar un cambio de dispositivo entre iteraciones de Jacobi es alta. En este trabajo estudiamos el reparto de iteraciones de warp completas (segunda opción) que demuestra ser un punto intermedio en cuanto a complejidad de implementación, pero permitiendo al mismo tiempo un control razonable de la carga en cada dispositivo.

En nuestra solución incluimos un sencillo mecanismo, a través de la declaración de una matriz de *mapping* o asignación de recursos, con el que el programador puede escoger en qué dispositivo se ejecuta cada iteración de warp de cada nivel. Para el caso de HSOpticalFlow la matriz tiene dos dimensiones. La primera tiene la cardinalidad del número de niveles y la segunda el número de iteraciones de warp por nivel. En el listado 2 se muestra un ejemplo de dicha matriz, en la que cada valor es el índice de uno de los dispositivos disponibles en la máquina objetivo.

C. Movimiento de datos

Las iteraciones de warp que se ejecutan en dispositivos diferentes deben trabajar sobre espacios de memoria diferentes. Esto permite que puedan solaparse las diferentes partes del pipeline. Sin embargo, en los puntos de corte del pipeline, cuando dos iteraciones de warp consecutivas se han asignado a dispositivos diferentes, los resultados parciales de la primera deben moverse o comunicarse de un dispositivo al siguiente.

En nuestra solución, las iteraciones de warp del mismo nivel asignadas al mismo dispositivo reutilizan la misma estructura de datos como entrada y salida. En un cambio de nivel dentro del mismo dispositivo la estructura de salida es diferente porque el tamaño es distinto, y es la estructura que se utiliza como entrada en el siguiente nivel. En el caso de que dos iteraciones de warp consecutivas estén asignadas a dispositivos diferentes, la salida de la primera es una estructura de datos diferente que se usa como entrada en la siguiente iteración. Estas estructuras de datos de paso entre dispositivos son las únicas que tendrán duplicado su espacio de memoria en ambos dispositivos y en el host. Esta última se utilizará como intermediario para realizar la transferencia de datos entre las imágenes de memoria de los dos dispositivos. En la figura 2 se muestra un ejemplo.

Para integrar esto en el código de una forma sencilla, utilizamos una matriz de punteros con valores para todos los niveles e iteraciones de warp. Cada iteración utiliza el puntero correspondiente a su nivel y número de iteración como entrada y el puntero de su nivel y siguiente número de iteración como salida. En el caso de un cambio de nivel la salida es el puntero correspondiente a la primera iteración del siguiente nivel. Al crear las estructuras de datos se inicializan estos punteros para que apunten a la estructura de datos correspondientes.

D. Implementación usando Controller

Para construir un prototipo experimental que pueda trabajar fácilmente con múltiples dispositivos heterogéneos hemos trabajado con el modelo de programación Controller [2].

D.1 Preparación

Partimos del código original de HSOpticalFlow provisto por NVIDIA en los ejemplos del toolkit de desarrollo de CUDA. La versión de referencia de CUDA utiliza extensivamente la memoria de texturas de las GPUs por sus beneficios relacionados con: (1) el modo de direccionamiento en espejo que solucionan problemas en el cálculo de derivadas y accesos en los contornos; (2) la interpolación bilineal que se utiliza en las operaciones de restricción, prolongación y *warping*; y (3) mejorar la localidad de datos gracias a las cachés de texturas [4]. Hemos incorporado al

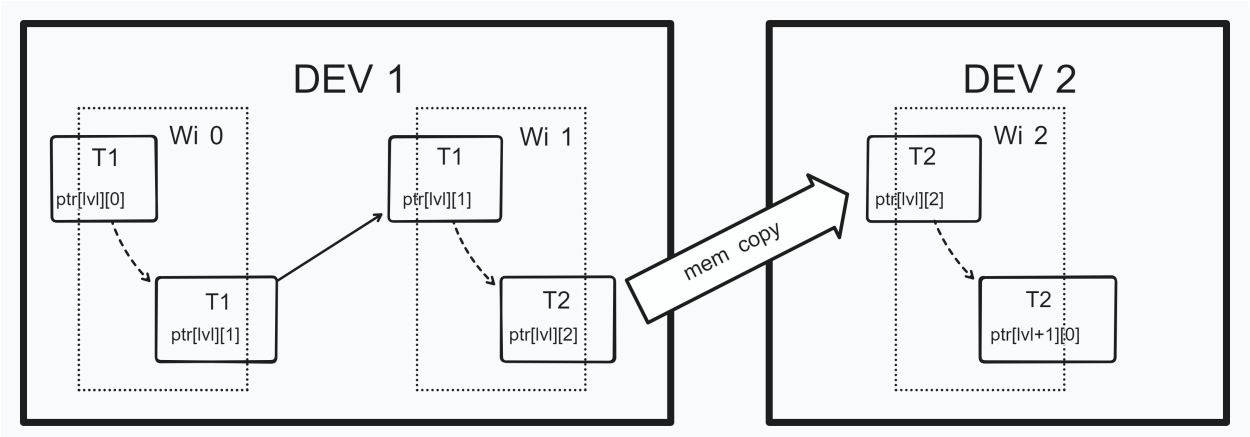

Fig. 2: Diagrama de las estructuras de datos usadas en cada iteración de warp cuando la siguiente iteración está asignada al mismo dispositivo y cuando está asignado a otro.

modelo Controller funcionalidades para trabajar con este tipo de memorias.

El código original está preparado para ejecutarse sólo con un par de fotogramas de entrada. Para tener la aplicación de streaming completa que trabaja sobre una secuencia de imágenes, añadimos un bucle externo para leer fotogramas (ver el listado 3). Para medir con más exactitud los tiempos de trabajo del algoritmo principal, en lugar de escribir los resultados en un fichero, simplemente calculamos la norma del resultado de cada par de fotogramas. Esto permite que en cualquier versión que se modifique a partir de esta se pueda comprobar la corrección con un buen nivel de confianza haciendo una sencilla comparación con los resultados de las normas en la versión de referencia.

D.2 Portar kernels

Para portar un kernel de CUDA al modelo Controller lo primero es cambiar los punteros a estructuras de datos que representan las matrices en el kernel por *HitTiles*. Este tipo representa una estructura utilizada en el modelo Controller para manejar estructuras de datos junto con metadatos que las describen e información sobre el estado de la memoria en el host o el dispositivo. Esto permite al modelo Controller detectar la necesidad de movimientos de datos para mantener la coherencia entre las diferentes copias de la memoria de un HitTile en el host o en diversos dispositivos. En los prototipos de los kernels en el modelo Controller se indica información sobre el rol de entrada o salida de un parámetro HitTile para poder detectar de forma implícita la necesidad de dichos movimientos. El segundo paso consiste en sustituir los índices nativos de CUDA por su contrapartida en Controller y lo mismo para los accesos a la memoria, que en Controller se realizan con unas macro funciones que reciben como parámetros el HitTile y los índices. En general, esto resulta en una ligera simplificación del código del kernel al estar los tamaños de los espacios de memoria integrados en las HitTiles y los índices de hilo globales calculados implícitamente por Controller.

D.3 Portar código de host

Portar el algoritmo principal a Controller es bastante directo. Controller provee de un objeto para acceder y manejar cada dispositivo disponible en la plataforma que se haya declarado visible en el momento de ejecutar el programa. Se crean objetos HitTile para cada estructura de datos, asociando imágenes de memoria en el host y en cada dispositivo donde sea necesaria para realizar cómputos.

Para cada par de fotograma tenemos dos operaciones que se ejecutan en el host (la carga del nuevo fotograma y el cálculo de la norma). Para asegurar que estas operaciones sean independientes y que las operaciones del pipeline se puedan solapar debidamente, lanzamos una de estas operaciones como *host task* y la otra como un kernel de CPU en modo tarea. Este mecanismo de Controller permite utilizar parte de los cores de la CPU como un dispositivo más, independiente del flujo de ejecución del host.

Implementar el mecanismo de reparto de carga en la versión de Controller es sencillo. Con un bucle que recorre la matriz de reparto se van creando los HitTiles guardando punteros a los mismos en un array de dos dimensiones con tamaños similares a la matriz de reparto. Los HitTiles se generan con imágenes en memoria sólo en un dispositivo, o en el host y dos dispositivos cuando haya una transición entre dos iteraciones de warp asociadas a dos dispositivos diferentes. Las llamadas a los kernels usan el array de punteros y los índices adecuados para escoger los parámetros de entrada y salida adecuados al nivel e iteración de warp en cada llamada. El mecanismo de movimiento de datos implícito de Controller realiza automáticamente los movimientos de memoria necesarios de forma asíncrona para permitir el solapamiento de cómputo y comunicaciones de una forma eficiente [2].

IV. Estudio experimental

En esta sección se describe un estudio experimental realizado para verificar la eficiencia de la solución propuesta.

0	0	0
0	0	0
0	0	0
0	0	0
0	1	1

1	1	1
1	1	1
1	1	1
2	3	0
0	0	0

1	1
1	1
1	1
2	3
0	0

Fig. 3: Matrices de reparto de carga utilizadas en la experimentación. Las filas son niveles y las columnas iteraciones de warp. El número indica el identificador de dispositivo. En manticore los 2 dispositivos son V100 y en gorgón el 0 es la A100 y el resto Titan Blacks

A. Entorno y diseño de experimentos

El estudio experimental se ha llevado a cabo en las máquinas *manticore* y *gorgón* del clúster de investigación del grupo Trasgo, que tienen las siguientes características:

- **Manticore**:
  - Procesador: 2x Intel(R) Xeon(R) Platinum 8160 CPU @ 2,10 GHz
  - Memoria RAM total: 512 GB DDR4.
  - 2x NVIDIA Tesla V100 32 GB HBM2 GPU.
  - 2x AMD Vega 10 XT Radeon PRO WX 9100 GPU.
- **Gorgón**:
  - Procesador: 2x AMD EPYC 7713 CPU @ 2,0 GHz
  - Memoria RAM total: 512 GB DDR4.
  - 1x NVIDIA A100 40 GB HBM2 GPU.
  - 3x NVIDIA GeForce GTX Titan Black 6 GB GDDR5 GPU.

El sistema operativo de las máquinas es la distribución de Linux CentOS versión 7.9.2009. Todos los programas se compilan con NVCC y GCC v10.3 y se lanzan desde un frontend usando slurm. La versión de CUDA instalada es 11.3. El estudio experimental se realiza sobre secuencias de diferente longitud en fotogramas de un video en resolución 8K (7680×4320), midiendo el tiempo que se tarda en calcular el flujo óptico entre pares de fotogramas consecutivos.

Se realizan pruebas usando las diferentes versiones de la aplicación y con diferentes configuraciones de dispositivos que se detallan a continuación:

- Ref: Versión de CUDA basada en la referencia, usa un solo dispositivo.
- Ref Pinned: Igual que la anterior, pero reservando la memoria del host como pinned para obtener transferencias de memoria más rápidas
- Ctrl: Versión utilizando el modelo Controller. Estudiamos ejecuciones con un solo dispositivo y con múltiples dispositivos realizando reparto de carga.

Para los experimentos se utilizan primero los valores de niveles e iteraciones de warp por defecto de la aplicación original, cinco y tres respectivamente, y 500 iteraciones de Jacobi en cada iteración de warp. En un segundo experimento se usan 5 niveles, 2 iteraciones de warp y 750 iteraciones de Jacobi. Para los casos de múltiples dispositivos se han usado estas matrices de reparto indicadas en la figura 3.

Para cada escenario se ejecutan 30 repeticiones,

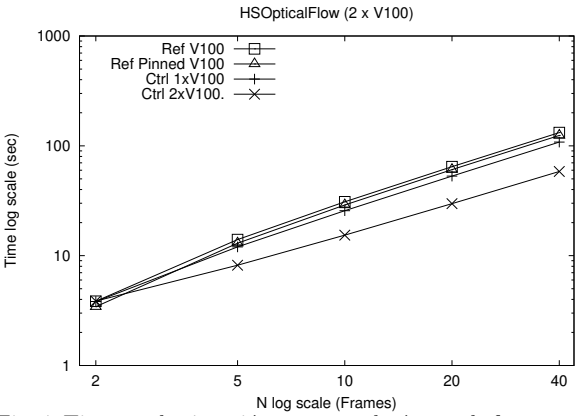

Fig. 4: Tiempos de ejecución respecto al número de fotogramas en manticore usando las V100

ya que este es un tamaño de muestra suficientemente grande para que el Teorema Central del Límite se considere aplicable. Los resultados presentados son la media de los tiempos de ejecución tras haber eliminado los outliers, es decir, aquellos resultados por debajo o por encima de la media $\pm 1,5 \times IQR$ (rango intercuartílico).

En esta aplicación y para la resolución usada, la ocupación y uso de los recursos de una GPU es prácticamente completa. Hemos comprobado experimentalmente que intentar usar la misma GPU para computar varios pares de fotogramas simultáneamente no reporta beneficios.

B. Resultados

En esta sección discutimos las observaciones obtenidas a la vista de los resultados. Para mostrarlos de forma resumida presentamos gráficas con el número de fotogramas de la secuencia en el eje x y tiempo de ejecución (con escala logarítmica) en el eje y.

En la figura 4 se muestran los tiempos de ejecución de las diferentes versiones de HSOpticalFlow en la máquina manticore, dotada con dos GPUs NVIDIA V100. Las versiones de referencia pueden aprovechar sólo uno de los dos dispositivos. La versión de referencia que usa memoria pinned siempre es ligeramente más eficiente que la original, ya que las transferencias de memoria son menos costosas. Esto se observa en las ejecuciones de la versión de referencia en todos los escenarios. Se observa que la versión con Controller es ligeramente menos eficiente para un sólo par de fotogramas, ya que la construcción del pipeline es innecesario y no hay posibilidades de solapar ningún cómputo. Sin embargo, en cuanto hay un mayor número de fotogramas en la secuencia, incluso la versión de Controller que sólo explota un dispositivo es ligeramente más eficiente que la referencia con memoria pinned, hasta un 5 % menos de tiempo de ejecución total. Al contrario que en la versión de referencia, con el modelo de controller se están solapando las tareas de lectura de fotograma y cálculo de la norma de resultado en el host con el cómputo en el dispositivo. Además, Controller usa internamente un mecanismo de comunicaciones asíncronas en diferentes streams controlados por eventos que es muy

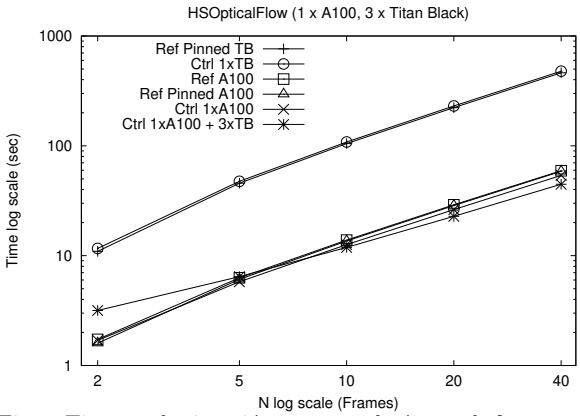

Fig. 5: Tiempos de ejecución respecto al número de fotogramas en gorgón usando una A100, una Titan Black, o la A100 con tres Titan Black

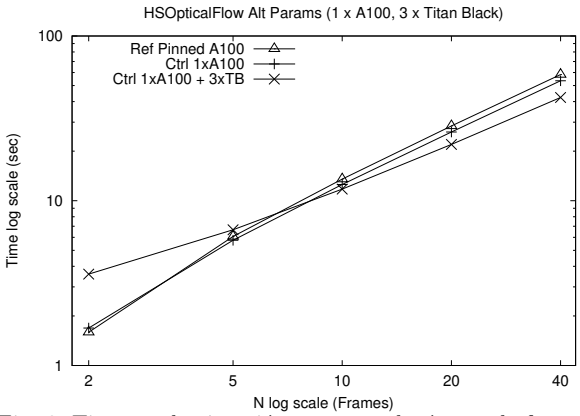

Fig. 6: Tiempos de ejecución respecto al número de fotogramas, versión con parámetros alternativos (5 niveles 2 warp 750 solves) en gorgón

eficiente. En la tabla izquierda de la figura 3, se muestra la matriz de asignación de dispositivos utilizada en los experimentos con las dos V100 de manticore. En la figura 8 se muestran la aceleración obtenida al usar las 2 V100 en comparación con la versión de Controller que usa una sola GPU, se observa como a partir de un mínimo número de fotogramas, aproximadamente cinco, el pipeline ya está lleno y a partir de ese momento la versión que explota los dos dispositivos consigue un muy buen equilibrio de carga y un muy buen solapamiento de cómputo y transferencias de memoria. Se consigue una aceleración de 1,85x con los dos dispositivos, lo que indica una eficiencia del 92,5 % en la explotación del paralelismo entre dispositivos. Comparado con la versión de referencia con memoria pinned se consigue una mejora de más de 2x, debido a las mejoras introducidas por el solapamiento de las tareas de host con el cómputo del dispositivo y el mecanismo interno de Controller para la gestión de los streams y eventos.

En la figura 5 se muestran los tiempos de ejecución de las diferentes versiones de HSOpticalFlow en la máquina gorgón, dotada con una GPU NVIDIA A100 y tres antiguas GPUs NVIDIA Titan Black de arquitectura Kepler. En los resultados se puede observar que tanto en la versión de referencia como en la de Controller la ejecución en una única Titan Black es unas 8 veces más lenta que en la A100. Concretamente, la referencia con la Titan Black tarda $462,38$ segundos y con la A100 tarda $58,64$ segundos para 40 fotogramas.

Al igual que en el caso de utilizar una única V100 en manticore, la ejecución de la versión de Controller con sólo la A100 en cuanto hay un número mínimo de fotogramas en la secuencia para llenar el pipeline es ligeramente más eficiente que las versiones de referencia. En el caso de intentar utilizar tanto la A100 como las tres Titan Black con el reparto indicado en ta tabla central de la figura 3, para sólo dos fotogramas el resultado es, como era esperable, mucho peor. Se están procesando etapas en las GPUs Titan Black, mucho más lentas que la A100, sin posibilidad de explotar en paralelo o solapar ninguna otra acción. De nuevo, en cuanto hay una secuencia mínima (cinco fotogramas o más) el pipeline está lleno y se pueden explotar las posibilidades de ejecución paralela y mecanismos de solapamiento de cómputo y transferencias de memoria. Como se puede ver en la figura 7, esto resulta en una aceleración sostenida con respecto a la versión de Controller que usa la A100 de 1,2x. Al ser las Titan Black unas 8 veces más lentas que la A100, el pico teórico de aceleración sería aproximadamente 1,35x. La mejora obtenida no es ideal, con una eficiencia del 89 %. Esto se debe a que el equilibrio de carga no es demasiado bueno, ya que para el grado de granularidad escogido la carga de las iteraciones de warp en cada nivel no permite dividir la computación de forma perfectamente equilibrada.

Se puede aplicar un pequeño cambio en la configuración de los niveles e iteraciones de warp por nivel, como se muestra en la tabla de reparto de carga que se muestra a la derecha de la figura 3. Con esta configuración se consigue un nivel de convergencia similar a la obtenida con la configuración anterior, sin alterar significativamente los tiempos de ejecución de la referencia (menos del $0,5\,\%$). En la figura 6 se muestran los tiempos de ejecución de las diferentes versiones de HSOpticalFlow en la máquina gorgón con la nueva configuración de niveles e iteraciones de warp por nivel. Con esta configuración se puede conseguir un mejor equilibrio de carga. En la figura 7 podemos ver que con estos parámetros alternativos se alcanza una aceleración de 1,26x en comparación con usar solo la A100 en Controller, lo que corresponde a una eficiencia del 93,5 %. Esto equivale a una mejora de 1,37x con respecto a la aplicación de referencia usando la A100 y memoria pinned.

V. Conclusiones

En este trabajo se presenta una sencilla metodología y mecanismos para introducir la explotación de múltiples dispositivos heterogéneos, potencialmente con diferentes capacidades de cómputo. Esta propuesta permite equilibrar la carga de trabajo en una clase representativa de aplicaciones de streaming o flujo de datos basadas en métodos multi-grid. Se ha utilizado como caso de estudio la aplicación HSOpticalFlow, que estima el movimiento aparente de obje-

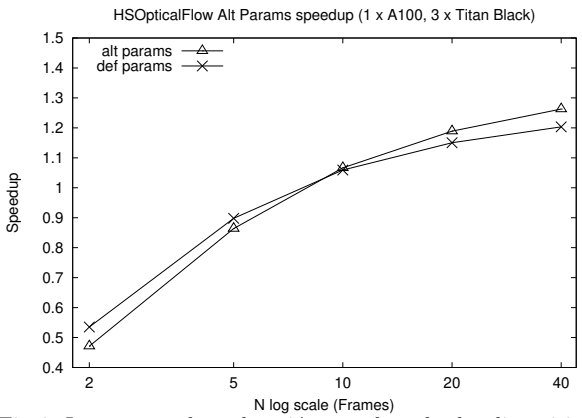

Fig. 7: Incremento de aceleración usando todos los dispositivos vs. usar solo la A100, con diferente número de fotogramas en la versión de Controller, en gorgón usando los parámetros por defecto o los parámetros alternativos.

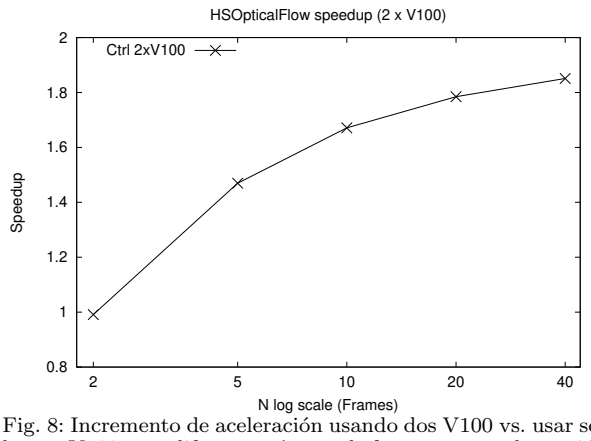

Fig. 8: Incremento de aceleración usando dos V100 vs. usar solo una V100, con diferente número de fotogramas en la versión de Controller, en manticore con los parámetros por defecto.

tos en una secuencia de imágenes. Se describe como modificar la aplicación para introducir los mecanismos propuestos e implementarla con el modelo de programación heterogénea Controller. La aplicación resultante permite ejecutar la aplicación como un pipeline, escogiendo con una sencilla configuración qué fases del cómputo se realizan en cada posible dispositivo. Esto permite al programador equilibrar la carga entre los diferentes elementos de cómputo. El mecanismo propuesto y el modelo de programación Controller crean un eficiente solapamiento de cómputo y comunicaciones de forma transparente.

Se presenta un estudio experimental usando como punto de partida la aplicación de referencia incluida en el toolkit de desarrollo de CUDA y una versión programada con Controller y los mecanismos propuestos. Los resultados muestran que la estructura de pipeline y el mecanismo de reparto de carga introducidos permiten conseguir una excelente explotación del paralelismo entre dispositivos a partir de un número muy reducido de fotogramas en la secuencia. Además, los mecanismos internos del modelo de programación Controller para la sincronización y movimientos de datos, no sólo son transparentes y simplifican el desarrollo de este tipo de soluciones, sino que además son muy eficientes y permiten aprovechar las posibilidades de solapamiento de cómputo y comunicaciones sin necesidad de una intervención por parte del programador.

Como trabajo futuro se ha planteado estudiar la aplicación de los mecanismos propuestos en otras aplicaciones y contextos, explorar su uso con otros modelos de programación heterogénea y considerar la introducción transparente de diferentes niveles de granularidad para permitir una mayor precisión en el equilibrio de carga.

Agradecimientos

El presente trabajo es parte de la actuación PID2022-142292NB-I00 (Proyecto de Generación de Conocimiento 2022), financiada por MICIU/AEI /10.13039/501100011033 y por FEDER, UE. También se ha recibido el soporte del Programa Investigo del Servicio Público de Empleo Estatal, Convocatoria para la Contratación de Personal Investigador, financiado por la Unión Europea-NextGenerationEU. Las plataformas de experimentación han sido financiadas parcialmente por el programa NVIDIA Academic Hardware Grant Program.

Referencias

[1] The Khronos SYCL Working Group, "Sycl 2020 specification (revision 8)," Tech. Rep., The Khronos Group, 2023.

[2] Yuri Torres, Francisco J. Andújar, Arturo González-Escribano, and Diego R. Llanos, "Supporting efficient overlapping of host-device operations for heterogeneous programming with ctrlevents," *J. Parallel Distributed Comput.*, 2023.

[3] Simon Farrelly, Ravi Reddy Manumachu, and Alexey L. Lastovetsky, "Openh: A novel programming model and api for developing portable parallel programs on heterogeneous hybrid servers," *IEEE Access*, 2024.

[4] M. Smirnov, "Optical flow estimation with cuda," White paper, NVIDIA, 2013.

[5] NVIDIA, "Cuda samples: Hsopticalflow," GitHub, on `https://github.com/NVIDIA/cuda-samples/tree/master/Samples/5_Domain_Specific/HSOpticalFlow`.

[6] M.T. Heath, *Scientific Computing: An Introductory Survey*, McGraw Hill, 1997.

[7] M.J. Quinn, *Parallel Computing: Theory and Practice*, McGraw-Hill, 1993.

[8] AdaptiveCPP contributors, "Adaptivecpp," GitHub, on `https://github.com/AdaptiveCpp/AdaptiveCpp`.

[9] James Reinders, Ben Ashbaugh, James Brodman, Michael Kinsner, John Pennycook, and Xin Tian, "Data parallel c++," *null*, 2023.

[10] Marc González Tallada and Enric Morancho, "Compute units in openmp: Extensions for heterogeneous parallel programming," *Concurrency and Computation*, 2023.

[11] Marco Danelutto, Marco Danelutto, Tiziano De Matteis, Tiziano De Matteis, Daniele De Sensi, Daniele De Sensi, Gabriele Mencagli, Gabriele Mencagli, Massimo Torquati, Massimo Torquati, Marco Aldinucci, Marco Aldinucci, Peter Kilpatrick, and Peter Kilpatrick, "The rephrase extended pattern set for data intensive parallel computing," *International Journal of Parallel Programming*, 2019.

[12] August Ernstsson, August Ernstsson, Lü Li, Lu Li, Lu Li, Lu Li, Lu Li, Christoph Keßler, and Christoph Kessler, "Skepu 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems," *International Journal of Parallel Programming*, 2018.

[13] Borja Pérez, Borja Pérez, Esteban Stafford, Esteban Stafford, José Luis Bosque, José Luis Bosque, Ramón Beivide, and Ramon Beivide, "Sigmoid: An auto-tuned load balancing algorithm for heterogeneous systems," *Journal of Parallel and Distributed Computing*, 2021.

[14] Sergio Alonso Pascual, "Modelo de ejecución y sincronización en múltiples dispositivos heterogéneos," Trabajo fin de máster (master thesis), Departamento de Informática, Facultad de Informática de A Coruña, Marzo 2023.