Patrón pipeline aplicado a arquitecturas heterogéneas big.LITTLE

Antonio Vilches, Andrés Rodríguez, Ángeles Navarro, Francisco Corbera y Rafael Asenjo[1]

Abstract— **En este trabajo, proponemos una solución para permitir la ejecución de aplicaciones de tipo *streaming*, que constan de una serie de etapas, sobre arquitecturas heterogéneas con un *multicore* y una *GPU* integrada. Para ello, presentamos una *API* que permite especificar el nivel de paralelismo explotado en el *multicore*, la asignación de las etapas del *pipeline* a los procesadores (*CPU* y *GPU*), y el número de threads. Usando una aplicación real de tipo *streaming* como caso de estudio, evaluamos el impacto que tienen los parámetros anteriores sobre el rendimiento y la eficiencia energética de un procesador heterogéneo (*Exynos 5 Octa*) que se caracteriza por tener 3 tipos de procesadores: Una *GPU*, un *quad-core* ARM Cortex A15 y un *quad-core* ARM Cortex A7. Además, exploramos algunas optimizaciones de memoria y encontramos que el resultado de las mismas depende del tipo de la granularidad del trabajo, y que además ayudan a reducir el consumo de energía.**

Keywords— **Patrón pipeline, Chips heterogéneos, GPU integrada en chip, Eficiencia energética.**

I. Introducción

En este trabajo nos centramos en resolver el problema que surge a la hora de ejecutar aplicaciones de tipo *streaming* sobre arquitecturas heterogéneas. Las aplicaciones de tipo *streaming* se implementan típicamente como una serie de etapas en pipeline y son ampliamente usadas en cualquier sistema de cómputo, pero especialmente en los sistema portátiles [1] como *smartphones* y *tablets*, donde los chips heterogéneos dominan el mercado. Nuestra propuesta extiende la plantilla *pipeline* que ofrece la librería *Intel Threading Building Blocks* (*TBB*) [2] para permitir la ejecución de este patrón sobre sistemas heterogéneos. Con esta extensión de la plantilla se alcanzan tres objetivos: i) simplificar el trabajo de programación de arquitecturas con GPU y multicores; ii) optimizar la ejecución mediante un reparto balanceado de la carga entre los distintos dispositivos; y iii) reducir el consumo de energía permitiendo al programador elegir fácilmente la configuración del pipeline más adecuada a tal efecto.

Como caso de estudio y para mostrar un ejemplo de uso de nuestra plantilla, introducimos ViVid[1], una aplicación que implementa un algoritmo [3] de detección de objetos (caras), que se basa en una aproximación de ventana deslizante para la detección de objetos [4]. ViVid consiste en un *pipeline* de 5 etapas, tal y como muestra la Fig. 1. La primera y última etapas son las etapas de entrada y salida respectivamente (son etapas serie), mientras que las tres etapas intermedias son etapas paralelas (sin estado[2]).

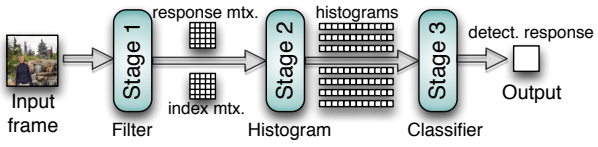


Fig. 1
Diagrama de flujo de la aplicación ViVid.

Cuando aplicaciones como ViVid son ejecutadas en arquitecturas heterogéneas integradas en chip, existen muchas posibles configuraciones. Por ejemplo, el usuario tiene que considerar la granularidad, el número de *items* del pipeline que se deben de computar simultaneamente, el procesador al que debe de ser asignada cada etapa y el número de *cores* de *CPU* a usar, para minimizar el tiempo de ejecución, el consumo de energía, o ambas (dependiendo de la métrica de interés). La granularidad determina el nivel al que el paralelismo es explotado en la *CPU*. En nuestra aproximación, el usuario puede especificar dos niveles de granularidad: grano grueso (Coarse Grain, CG) y grano medio (Medium Grain, MG). Si diferentes *items* pueden ser ejecutados simultaneamente en la misma etapa (sin estado) en la *CPU*, entonces la granularidad CG puede ser explotada. Por otro lado, si el cuerpo de una etapa es paralelizable, entonces un único *item* puede ser procesado en paralelo por varios *cores* de *CPU* en dicha etapa, y la granularidad MG puede ser explotada. En nuestra propuesta, la granularidad CG implica que hay tantos *items* procesándose simultáneamente en el pipeline como *threads* disponibles, mientras que en la granularidad MG hay como máximo dos *items* siendo procesados a la vez, uno por el *multicore* y otro por la *GPU*.

El mapeo del *pipeline* determina los procesadores en los que se pueden ejecutar las diferentes etapas del mismo. La Fig. 2 muestra todos los posibles mapeos que se pueden dar para las tres etapas paralelas de la aplicación ViVid.

Asumamos que un *pipeline* esta compuesto de S_1, S_2, ... S_n etapas paralelas. Usamos una n-tupla para especificar todos los posibles mapeos a la *GPU* y a los *cores* de *CPU*: $\{m_1, m_2, \ldots, m_n\}$. El i-ésimo elemento de la tupla, m_i, especifica si la etapa S_i puede ser mapeada a la *CPU* y *GPU*, ($m_i = 1$), o si ésta puede ser mapeada únicamente a la *CPU*,

[1]Dpto. de Arquitectura de Computadores, Univ. Málaga, Andalucía Tech. e-mail: {avilches, andres, angeles, corbera, asenjo}@ac.uma.es

[1]http://www.github.com/mertdikmen/vivid

[2]Una etapa es paralela o sin estado cuando el cómputo sobre un elemento en esa etapa no depende del cómputo de elementos anteriores.

($m_i = 0$). Si $m_i = 1$, cuando entra un *item* en la etapa S_i se comprueba si la *GPU* está libre. Si la *GPU* está libre, la etapa S_i procesa dicho *item* en la *GPU*; en otro caso, es procesado en la *CPU*. Si $m_i = 0$, el *item* sólo podrá ser procesado en la *CPU*. Por ejemplo, las posibles configuraciones de la aplicación ViVid de la Fig. 2 se corresponden con las siguientes tuplas: {1,1,1}, {1,0,0}, {0,1,0}, {0,0,1}, {1,1,0}, {1,0,1}, {0,1,1}, {0,0,0}. En nuestra implementación, el mapeo {1,1,1} representa un caso especial: si la *GPU* está libre cuando un nuevo *item* entra al *pipeline*, entonces todas las etapas para este *item* se mapean en la *GPU*.

En este trabajo, primero se introduce la *API* de nuestro framework que permite especificar una aplicación basada en la plantilla *pipeline* junto con los parámetros de interés para el usuario como son: i) la granularidad que puede ser explotada en cada etapa por la CPU (Grano Grueso o Medio); ii) el mapeo de las etapas a los diferentes procesadores; y iii) el número de threads de *CPU* a utilizar (Sección II). Luego se presentan los resultados de distintas ejecuciones del código ViVid sobre una arquitectura heterogénea en chip, donde se analiza el impacto de la granularidad, el mapeo de las etapas y el número de threads sobre el rendimiento y el consumo de energía (Sección III). Finalmente, se presentan algunos trabajos relacionados (Sección IV) y conclusiones (Sección V).

II. Interfaz de la plantilla pipeline

En esta sección, se introduce la *API* de nuestra plantilla *pipeline*. Ésta proporciona un entorno de programación para el lenguaje *C++* que facilita la configuración y ejecución de un *pipeline*, oculta la implementación *TBB* subyacente y gestiona automáticamente las transferencias de memoria entre *CPU* y *GPU*.

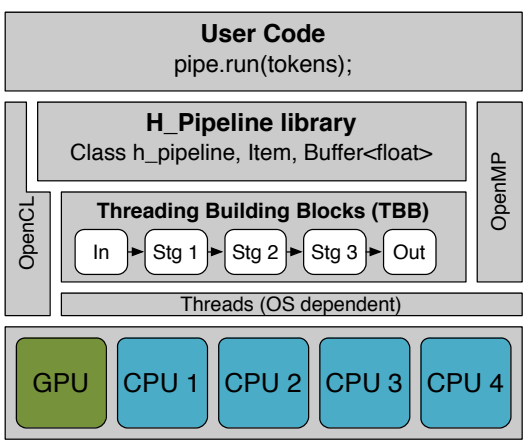


Fig. 3
Descripción de los componentes del Framework.

La Fig. 3 muestra todos los componentes en los que se apoya la plantilla *pipeline*. Básicamente, nuestra plantilla se basa en la implementación de la plantilla *pipeline* de *TBB* y ofrece la posibilidad de ejecutar tareas en la *GPU* mediante OpenCL. Además, dispone de un heurístico [5] que funciona en tiempo de ejecución que predice cual es la mejor configuración de la plantilla para optimizar el rendimiento, el consumo de energía o una combinación de ambas métricas. Por otro lado, la Fig. 4 muestra el lugar que ocupan las entidades (clases, objectos) de la plantilla, en el flujo de procesamiento del *pipeline*. El objeto *item* es la entidad que recorre el pipeline. Éste contiene referencias a los *buffers* de datos que son usados en las diversas etapas que componen el *pipeline*. Para crear una nueva instancia de la clase *pipeline*, el usuario tiene que definir una nueva clase que extienda a la clase *Item* proporcionada en el framework. Para facilitar la gestión de los *buffers* de datos, el framework proporciona una plantilla de clase *Buffer* que oculta las complejidades de las operaciones de reserva, liberación y movimientos de memoria entre *CPU* y *GPU*. El principal objetivo de esta clase *Buffer* es ofrecer una capa de abstracción que gestione los datos y los haga accesibles tanto a *CPU* como a *GPU* cuando el *item* esté siendo procesado.

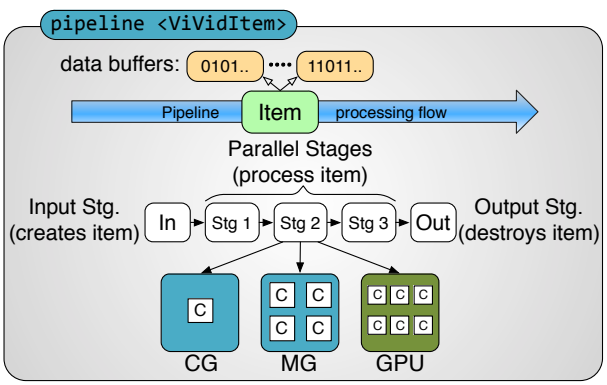


Fig. 4
Flujo de procesamiento en ViVid.

El programador puede proporcionar hasta tres implementaciones diferentes para cada etapa paralela del *pipeline*: una para implementar la etapa en *OpenCL* para la ejecución en la *GPU*, una segunda implementación para la ejecución sobre un *core* de *CPU* (granularidad CG), y una tercera implementación para ejecutar la etapa usando todos los cores de la *CPU* (granularidad MG).

La interfaz de programación (*API*) tiene 4 clases principales:

- *Items*: Son los objetos que recorren el pipeline con los *buffers* que almacenan los datos que son usados/generados en cada etapa.
- *Pipelines*: Representan a la entidad *pipeline* que está compuesta de n etapas. Los objetos de la clase *pipeline* pueden ser configurados estáticamente o usar un modo de autoconfiguración basado en modelo analítico [5] que dinámicamente encuentra la mejor configuración para una métrica dada.
- Etapas: Representan cada etapa de procesamiento del *pipeline*. En cada una de ellas se

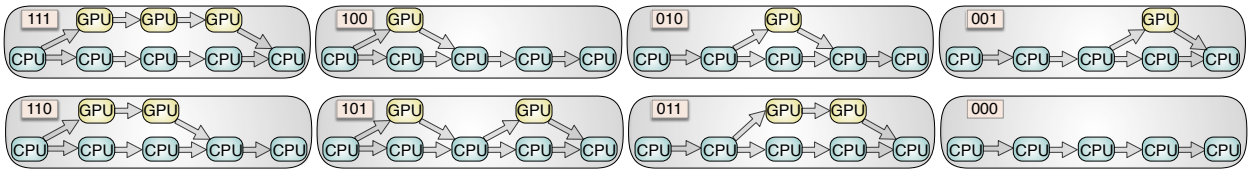


Fig. 2
Todos los posibles mapeos de etapas a GPU y/o CPU para la aplicación ViVid.

```
1  #include "h_pipeline.h"
2  using namespace h_pipeline;
3
4  class ViVidItem : h_pipeline::Item {
5    //Buffer Definitions
6    ...
7    ViVidItem(){...}   //Buffer allocation
8    ~ViVidItem(){...}  //Buffer deallocation
9  }
10
11 int main(int argc, char* argv[]){
12   int nTh = 4 ; //number of threads
13
14   h_pipeline::pipeline<ViVidItem> pipe(nTh);
15
16   //Set CG, MG and GPU functions for each
         stg.
17   pipe.add_stage(cg_f1, mg_f1, gpu_f1);
18   pipe.add_stage(cg_f2, mg_f2, gpu_f2);
19   pipe.add_stage(cg_f3, mg_f3, gpu_f3);
20
21   //Pipeline configuration: '101' and MG
22   pipe.set_conf({1,0,1},true);
23   pipe.run(numTokens);
24
25   //Dynamic dispatch of the pipeline
26   //pipe.run(numTokens, ENERGY, maxOverhead)
         ;
27 }
```

Fig. 5
Ejemplo de uso de la plantilla `pipeline`.

podrán especificar varias funciones que implementen distintas versiones del código para ser ejecutadas en *CPU* y/o *GPU*. La instancia del *pipeline* usará la función apropiada en cada etapa.

- *Buffers*: Contienen *arrays* de n-dimensiones que serán utilizados por las distintas implementaciones de las *Etapas* tanto si son ejecutadas en la *CPU* como en la *GPU*.

Debido a limitaciones de espacio, en este trabajo solo se muestran algunos detalles de especificación y uso de la plantilla *pipeline*. El lector que esté interesado en más detalles puede consultar el trabajo [6].

La Fig. 5 muestra un ejemplo de como definir y usar nuestra propuesta de plantilla *pipeline*. En primer lugar, se incluye el fichero de cabecera `h_pipeline.h` (línea 1) para hacer accesible la interfaz de programación (encapsulada dentro del *namespace* `h_pipeline`). El programador tiene que declarar una clase que extienda a la clase `h_pipeline::Item` (a modo de ejemplo, en línea 4 de la figura se define la clase `ViVidItem`). En esta clase hay que definir tantos *buffers* como sean necesarios para la ejecución de todas las etapas del *pipeline*. Los métodos de clase, constructor y destructor, son los encargados de reservar/adquirir y liberar los *buffers* de memoria respectivamente. El método constructor debe obtener el espacio de memoria necesario para mantener los *buffers*. Para esto, puede hacer una reserva de memoria nueva por cada *buffer* de cada *item* o usar un pool de *buffers* que son creados antes de la ejecución del pipeline y reusados por distintos *items* durante la ejecución del mismo. En este último caso, el método constructor utilizará el método de adquisición proporcionado por la clase *buffers*. El método destructor de la clase *item* deberá liberar los *buffers* (que implicará una simple liberación de memoria o pasar el buffer de nuevo al pool). La optimización por el uso del pool de *buffers* se presenta y evalúa en la secciónIII-D. La línea 14 muestra la declaración de un objeto de la clase *h_pipeline* instanciada para la clase de item anteriormente definida (`ViVidItem`). El argumento que recibe el constructor de la variable `pipe` (`nTh`) define el número de *threads* que ejecutarán simultáneamente el *pipeline*. Las etapas de Entrada y Salida (etapas serie) son creadas automáticamente por el template e invocan automaticamente al constructor y al destructor de la clase `ViVidItem`, respectivamente. Antes de poder ejecutar el *pipeline*, el programador debe de definir las funciones que se ejecutarán en la *CPU* (granularidad CG y/o MG) y en la *GPU* para cada una de las etapas paralelas (en el caso de la aplicación ViVid son tres, una para cada etapa paralela del *pipeline*: líneas 17 a 19). La implementación de nuestro template del pipeline se encarga de seleccionar y ejecutar la implementación apropiada para cada etapa, ya sea la que utiliza un solo *core* de *CPU* (`cg_f`), la que utiliza todo el *multicore* (`mg_f`), o la que utiliza la *GPU* (`gpu_f`). Cada función recibe como argumento de entrada un puntero al *item* que tiene que ser procesado en la etapa a la que pertenece dicha función. A su vez, cada *item* contiene punteros a los *buffers* que almacenan los datos de entrada y salida de cada función. De forma automática y trasparente, el usuario podrá acceder a los punteros en el espacio de memoria de *CPU* o de *GPU*, dependiendo del dispositivo donde se va a procesar el *item*.

Antes de ejecutar nuestro *pipeline*, puede ser configurado para que presente un comportamiento estático mediante el uso del método `pipe.set_conf()` (línea 22). Este método define cuales son las potenciales etapas que se pueden mapear en la GPU y

la granularidad (MG o CG) que se tiene que explotar en el `multicore`. En la figura, el primer argumento {1,0,1} representa la n-tupla para el mapeo de etapas en la GPU, definida según lo presentado en la sección anterior. Por otro lado, el segundo argumento `true`, indica que la granularidad MG será explotada cuando cualquier *item* sea procesado en la *CPU* (`false` indicaría granularidad CG). Una vez que el *pipeline* ha sido configurado, el usuario puede ejecutarlo utilizando el método `run` de la plantilla (línea 23), pasándole como argumento el número de *items* que pueden ser procesados a la vez en el *pipeline* (`numTokens`). Alternativamente, en la línea 26 se muestra como usar una implementación alternativa del método `run()`. Esta implementación tiene un comportamiento dinámico que encuentra la mejor configuración del *pipeline* (mapeo de etapas en la GPU y granularidad de la implementación de las funciones de CPU) basándose en un modelo analítico descrito en [5]. En este caso, el usuario tiene que seleccionar el criterio a optimizar (`THROUGHPUT, ENERGY, THROUGHPUT_ENERGY`) y el máximo *overhead* permitido a la plantilla (como ratio entre [0,1]) en la búsqueda de la mejor configuración.

A. Etapas del pipeline

Uno de los principales componentes de la plantilla `pipeline<T>` es la clase `parallel_stage`. Cada vez que se invoca el método `add_stage()` (ver Fig. 5 líneas 17 a 19) se crea un objeto de la clase `parallel_stage`. Esta clase contiene variables de instancia muy importantes: tres de ellas son punteros a las funciones CG (`cgFunc`), MG (`mgFunc`) y/o GPU (`gpuFunc`). Además, hay otras dos variables de instancia, `runOnGPU` y `grain`, que son usadas para decidir si la etapa tiene que ser ejecutada en *CPU* o *GPU* (decidido en tiempo de ejecución), o si se debe usar la granularidad MG o CG para la ejecución en *CPU*, respectivamente. La variable `runOnGPU` se inicializa de acuerdo a la n-tupla que se pasa como argumento en la invocación del método `set_conf()` (ver línea 22 en la Fig. 5) para una configuración estática, o es establecida automáticamente en la ejecución dinámica del *pipeline* (línea 26). El método principal de la clase `parallel_stage` es el método `operator()`. Este método se invoca automáticamente cuando un *item* alcanza dicha etapa paralela y recibe como parámetro de entrada un puntero al *item* que tiene que ser procesado. Dentro de este método se decide que función va a ser la encargada de procesar el *item*: si `runOnGPU` es `true` y la *GPU* está libre, el *item* es procesado en la *GPU* por la función `gpuFunc`. En otro caso, dependiendo de la variable `grain`, se invocará la función `mgFunc` o `cgFunc`. A la función seleccionada se le pasará el puntero al *item* para que pueda acceder a los *buffers* necesarios en dicha etapa.

III. Resultados experimentales

A. Entorno de experimentación

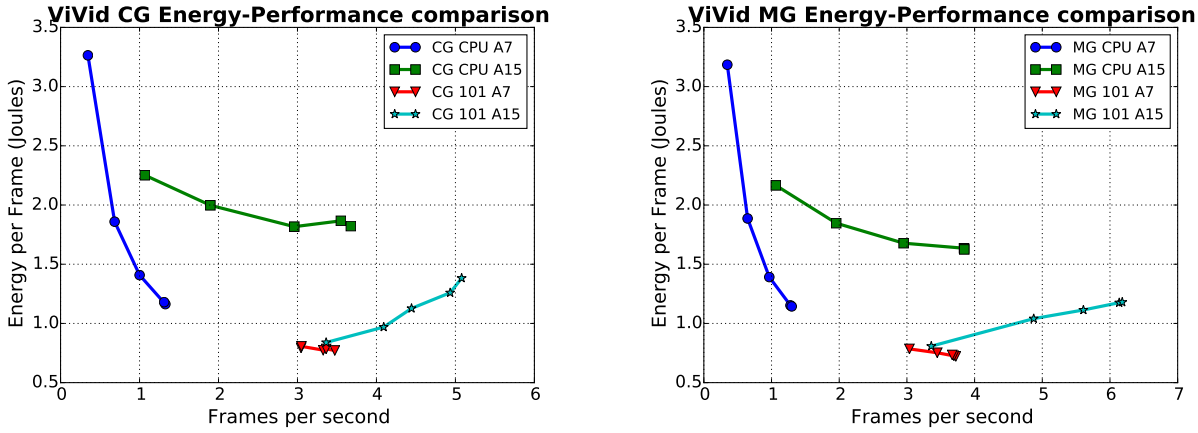
Para ejecutar nuestros experimentos hemos usado la placa Odroid XU3. Esta placa incluye un *SoC* (*System-on-Chip*) Samsung Exynos Octa (5422) que dispone de un *quad-core* Cortex-A15 que funciona a una frecuencia de 2.0 *Ghz* y de un *quad-core* Cortex-A7 (con una arquitectura big.LITTLE de ARM). El *quad-core* Cortex-A15 dispone de una cache L1 privada para cada core de 32 KB (Instrucciones) / 32 KB (Datos) y una cache L2 compartida de 2 MB. Por otro lado, el *quad-core* Cortex-A7 tiene una cache L1 de 32 KB (Instrucciones) / 32 KB (Datos) y una cache L2 compartida de 512 KB. Esta plataforma incorpora unos monitores de corriente/potencia basados en un chip INAI231 de Texas Instruments, que permite la lectura de la potencia instantánea consumida por los procesadores A15, por los A7, por la memoria principal y por la GPU. El *SoC* también incluye la *GPU* Mali-T628 MP6. Hemos elegido este procesador heterogéneo porque presenta una interesante característica que permite dos niveles de heterogeneidad: dos dispositivos con un *ISA* diferente (una *GPU* y una *CPU multicore*), y dos tipos de arquitecturas diferentes dentro del *multicore* de la *CPU*. De esta forma, este *SoC* presenta tres tipos de procesadores con diferentes capacidades de cómputo que además comparten el acceso a memoria principal.

La placa funciona con una versión de Linux Ubuntu 14.04.1 LTS y una librería [7] (desarrollada en nuestro grupo) para medir el consumo de energía. Se ha usado el compilador gcc 4.8 con las opciones de compilación -O3. Para la ejecución de los kernels en la *GPU* se ha usado el *SDK* de OpenCL v1.1.0 para la familia Mali. Los códigos de *CPU* han sido vectorizados usando los intrínsecos NEON. Para implementar la granularidad MG, se ha usado OpenMP mediante el pragma `#omp parallel for` en cada etapa. La librería Intel Threading Building Blocks (*TBB*) proporciona la gestión interna de la plantilla *pipeline*. Para cada experimento, se han medido el tiempo y la energía consumida en cinco ejecuciones distintas y se ha tomado la mediana de todas ellas. El conjunto de entrada ha consistido en 100 frames de video.

En la siguiente sección se explora y cuantifica el impacto que tienen la granularidad de las etapas (CG y MG), el mapeo de las etapas en los diferentes procesadores y el número de *threads*, sobre la eficiencia energética y el rendimiento. Posteriormente, se explora el impacto de varias optimizaciones de memoria.

B. Estudio del impacto de la granularidad

Primero vamos a presentar un estudio sobre el rendimiento y la eficiencia energética de ejecuciones homogéneas (solo un tipo de *cores* de *CPU*) *vs.* ejecuciones heterogéneas (se incorpora la *GPU*) donde se ejecuta la aplicación ViVid, descrita en la sección I. Para las ejecuciones heterogéneas se han explorado todos los posibles mapeos de etapas en los procesadores junto con los dos tipos de granularidad, Grano



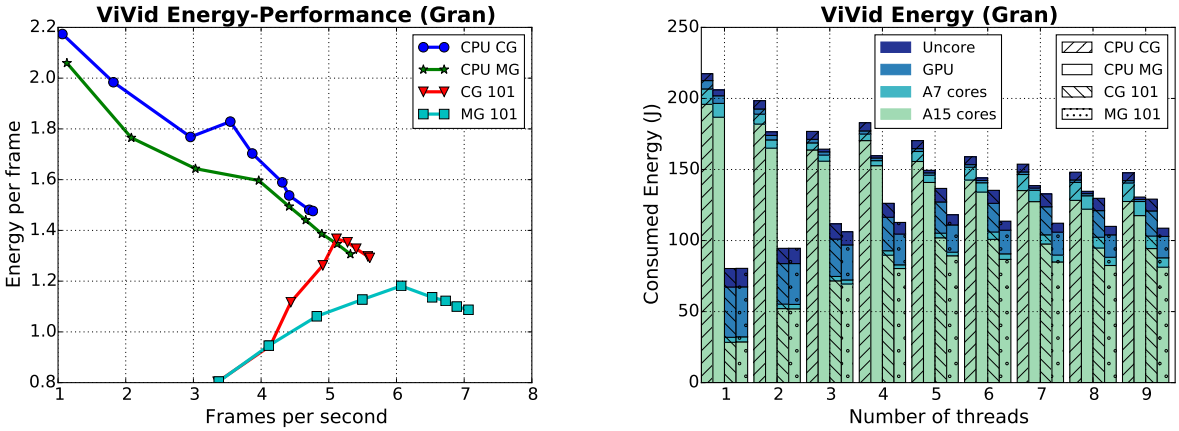
(a) (b)

Fig. 6
Estudio del impacto de las granularidades CG y MG en ViVid para los experimentos A7 vs A15 vs GPU+A7 vs GPU+A15.

Grueso (CG) y Grano Medio (MG). Se ha observado que para ViVid, el mapeo óptimo que alcanza el ratio más alto de *frames* por segundo (*pipeline throughput*) es {1,0,1} para ambas granularidades. Esto indica que las etapas 1 y 3 pueden ser potencialmente mapeadas en la *GPU* mientras que la etapa 2 solo puede ser mapeada en la *CPU*. Este resultado no es sorprendente ya que la etapa 2 ofrece un rendimiento pobre en la *GPU*. Así, las siguientes figuras muestran los resultados de ejecutar ViVid con este mapeo, representado con la etiqueta `101`. A modo de comparación, las figuras también muestran los resultados de los experimentos cuando solo trabaja la *CPU*, representados con la etiqueta `CPU`.

La Fig. 6 muestra como se comportan los diferentes procesadores bajo las granularidades CG y MG, cuando ViVid se ejecuta en un entorno homogéneo (solo *cores* de *CPU*) y en un entorno heterogéneo (*GPU* + *cores* de *CPU*). Para cada experimento, las figuras 6(a) y (b) muestran la energía por *frame* (Julios) *vs.* el ratio de *frames* (rendimiento medido en *frames* por segundo o fps) para cada granularidad. En otras palabras, estas figuras muestran la eficiencia energética respecto del rendimiento. Cada una de las líneas de las figuras tiene 5 puntos: cada punto representa la energía consumida por *frame vs.* los *frames* por segundo cuando el experimento es ejecutado usando 1 *thread* (el punto más a la izquierda) hasta 5 *threads* (el punto más a la derecha). Para realizar los experimentos en el A7 (A15), los threads son obligados a ejecutarse sobre el *multicore* A7 (A15) usando el comando `taskset`. En los experimentos homogéneos (`CG CPU A7`, `CG CPU A15`, `MG CPU A7` y `MG CPU A15`) los primeros 4 puntos representan el resultado de ejecutar de 1 a 4 *threads* sobre el *multicore* A7 o A15 con la granularidad CG y MG. El último punto representa un escenario de oversubscription en cada *multicore* (5 *threads* sobre 4 *cores*). Cuando la *GPU* es incorporada en los experimentos heterogéneos (`CG 101 A7`, `CG 101 A15`, `MG 101 A7` y `MG 101 A15`), el primer punto representa la ejecución con 1 *thread* donde sólo se utiliza la *GPU* para procesar los `items`. Los siguientes 4 puntos (de 2 a 5 *threads*) representan la contribución de los *multicores* A7 y A15 con granularidad CG y MG respectivamente.

Observando las Figs. 6(a) y (b) se puede apreciar que para los experimentos CG y MG, cuando se incrementa el número de threads, y de esta forma el número de *cores* activos, se produce un incremento del rendimiento. En cuanto a la eficiencia energética, para el caso de las ejecuciones homogéneas, podemos observar que mejora conforme se incrementa el número de threads en ambas granularidades. Para estos experimentos, las ejecuciones sobre el *quad-core* A7 son claramente más eficientes energéticamente (cuando el número de *threads* es mayor que 1), mientras que las ejecuciones sobre el *quad-core* A15 alcanzan un mayor rendimiento. Las figuras también muestran que cuando se incorpora la *GPU* (`CG 101 A7`, `CG 101 A15`, `MG 101 A7` y `MG 101 A15`) la energía consumida disminuye. En particular, usando únicamente la *GPU* (ejecución heterogénea con 1 *thread*), se observa una mejora en la eficiencia energética de 4.7x y 3.1x con respecto a usar un *core* del A7 (primer punto de la curva CG CPU A7) y un *core* del A15 (primer punto de la curva CG CPU A15), respectivamente. En ambas configuraciones de granularidad (CG y MG), el mínimo consumo de energía se alcanza en la ejecución heterogénea `101 A7` con 5 *threads*. Para el caso MG, es 1.5x veces más eficiente energéticamente que el caso homogéneo `MG CPU A7` con 5 *threads*. Es interesante resaltar que mientras las ejecuciones heterogéneas `CG 101 A7` tienden a aumentar ligeramente el rendimiento a la vez que se reduce marginalmente el consumo de energía cuando se incorporan más *threads*, las ejecuciones `CG 101 A15` incrementan de forma más apreciable el rendimiento, aunque pagando el precio de aumentar también el consumo



(a) (b)

Fig. 7
Estudio del impacto de la granularidad CG y MG en ViVid para los experimentos A15+A7 vs GPU+A15+A7.

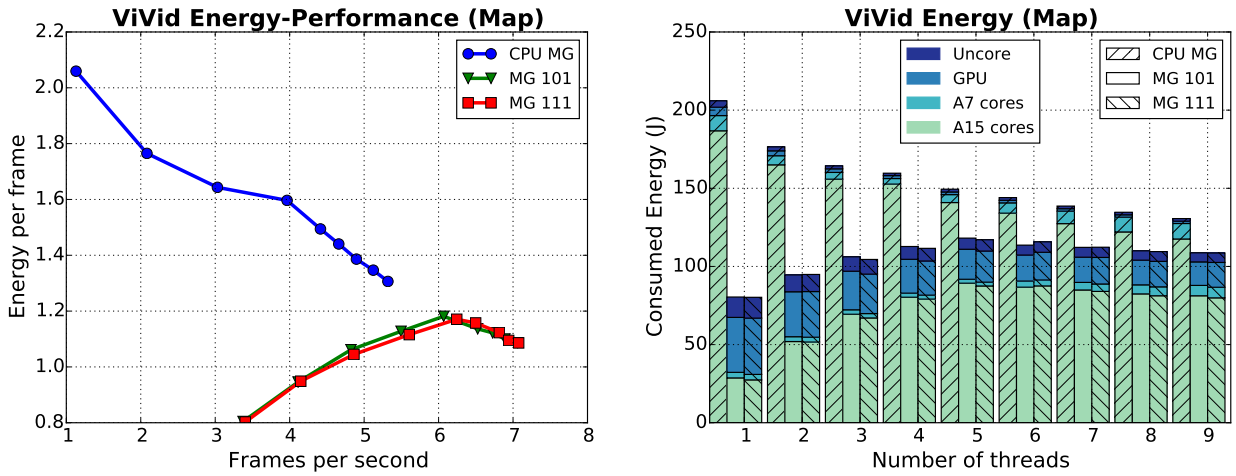
de energía. Se observa una tendencia similar para las ejecuciones MG heterogéneas. Otra observación interesante es que se alcanza un rendimiento similar en la configuración `CG CPU A15` con 4 *threads* y `CG 101 A7` con 5 *threads* (A15 *quad-core vs. GPU*+A7 *quad-core*), aunque el segundo es 2.7x veces más eficiente energéticamente que el primero. En cualquier caso, para un número de threads determinado, las ejecuciones MG heterogéneas proporcionan un mayor rendimiento y menor consumo de energía que sus ejecuciones CG equivalentes, para los experimentos sobre el A7 y el A15.

A continuación, se estudian los efectos de incorporar todos los dispositivos computacionales del *SoC* (A15, A7 y *GPU*) para las granularidades CG y MG. La Fig. 7 muestra un estudio de la granularidad CG y MG cuando se utilizan ambos *multicores* (el A15 y el A7). El estudio incluye experimentos donde sólo trabaja la *CPU* (`CG CPU` y `MG CPU`) frente a ejecuciones heterogéneas con la *GPU* (`CG 101` y `MG 101`). Nuestro objetivo es entender como los diferentes procesadores con sus características particulares influyen en el rendimiento y en la eficiencia energética.

La Fig. 7(a) muestra la energía consumida por frame *vs.* el ratio de *frames* por segundo. Cada línea muestra los resultados de ejecutar los experimentos con 1 *thread* (el punto situado más a la izquierda) hasta 9 (el punto situado más a la derecha). En los experimentos homogéneos (solo *CPU*), los primeros 4 puntos (desde 1 a 4 *threads*) representan la contribución de los *cores* del A15, los siguientes 4 puntos (desde 5 a 8 *threads*) representan la contribución de los *cores* del A7, y el último punto representa un escenario con *oversubscription* (9 *threads* sobre 8 *cores*). Cuando la *GPU* es incorporada en los experimentos heterogéneos `MG 101` y `CG 101`, el primer punto representa una ejecución donde únicamente la *GPU* está trabajando (1 *thread*), los 4 puntos siguientes (desde 2 a 5 *threads*) representan la contribución de los *cores* del A15, y los últimos 4 puntos (desde 6 a 9 *threads*) la contribución de los *cores* del A7.

En la Fig. 7(a) se puede apreciar que en los experimentos donde únicamente trabaja la *CPU* (`CPU CG` y `CPU MG`), la eficiencia energética y el rendimiento (*frames* por segundo) mejoran a medida que se incrementa el número de *threads*. Esta figura también muestra que la incorporación de la *GPU* (`CG 101` y `MG 101`) produce una reducción del consumo de energía. El mínimo consumo de energía se alcanza con 1 *thread* en la ejecución heterogénea (solo *GPU*), el cual es 2.7x y 2.6x veces más eficiente energéticamente que las ejecuciones con 1 *thread* de *CPU* (`CPU CG` y `CPU MG`) respectivamente. Sin embargo, tanto `CG 101` como `MG 101` tienden a incrementar el consumo de energía cuando se incorporan los *cores* del A15 (de 2 a 5 *threads*, como se puede apreciar entre los puntos 2 y 5 de ambas líneas). Tanto CG como MG mejoran el rendimiento cuando se incrementa el número de *threads*, pero CG consume más energía y muestra un rendimiento inferior a MG. Es interesante hacer notar que cuando se añaden los *cores* del A7 a los experimentos `CG 101` y `MG 101` (de 6 a 9 *threads*), las dos granularidades mejoran el rendimiento a la vez que reducen ligeramente el consumo de energía. En cualquier caso, para la aplicación ViVid, `MG 101` ofrece el mayor rendimiento (con 9 *threads*) pagando el coste de degradar el consumo de energía en un 32 % respecto al consumo mínimo (1 *thread MG 101*, ejecución solo *GPU*).

Por otro lado, la Fig. 7(b) muestra un desglose del consumo de energía cuando incrementamos el número de *threads* (de 1 a 9): se muestra la contribución del A15, del A7, de la *GPU* y de las unidades *uncore* del *SoC*. Las dos primeras barras representan experimentos donde solo trabaja la *CPU*: la primera barra representa el experimento `CPU CG` y la segunda representa el experimento `CPU MG`. La tercera y cuarta representan experimentos donde trabajan la *CPU* y la *GPU*: `CG 101` y `MG 101`, respectivamente. La figura muestra que para los experimentos de *CPU* a medida que se incrementa el número de *threads* la componente de consumo de energía de los A15 se



(a) (b)

Fig. 8
Estudio del impacto del mapeo en ViVid para los experimentos A15+A7 vs GPU+A15+A7.

reduce significativamente, mientras que la de los A7 aumenta ligeramente. Además, para cualquier número de *threads*, la componente del consumo de energía de los A15 es siempre menor en `CPU MG` que en `CPU CG`. Esto es debido a un mejor uso de la localidad en los niveles L1/L2 de las *caches*. Por otro lado, para los experimentos `CG 101` y `MG 101`, el consumo de energía de los A15 se incrementa de 1 a 5 *threads*. Sin embargo, el consumo de energía de la *GPU* se reduce porque la *CPU* está computando `items` que antes eran procesados por la *GPU*. El resultado global del sistema se traduce en un incremento del consumo de energía debido a que el A15 es menos eficiente energéticamente que la *GPU*. Si se añaden más *threads* a esos experimentos, empiezan a incorporarse los `cores` del procesador A7, lo que produce una ligera reducción del consumo de energía del A15 (ya que parte del proceso antes realizado por el A15 ahora lo realiza el A7), mientras que se incrementa marginalmente el consumo en el A7. En la figura también se puede apreciar que el consumo de energía de la *GPU* y de la componente *Uncore* están correlacionadas: ambas disminuyen de 1 a 5 *threads*, y posteriormente se estabilizan. El experimento `MG 101` muestra un consumo de energía menor en el A15, debido a un mejor uso de las *caches* bajo esta granularidad. En cualquier caso, para 9 *threads*, `MG 101` es un 16.2 % más eficiente energéticamente que `CG 101`.

C. Análisis del mapeo de etapas

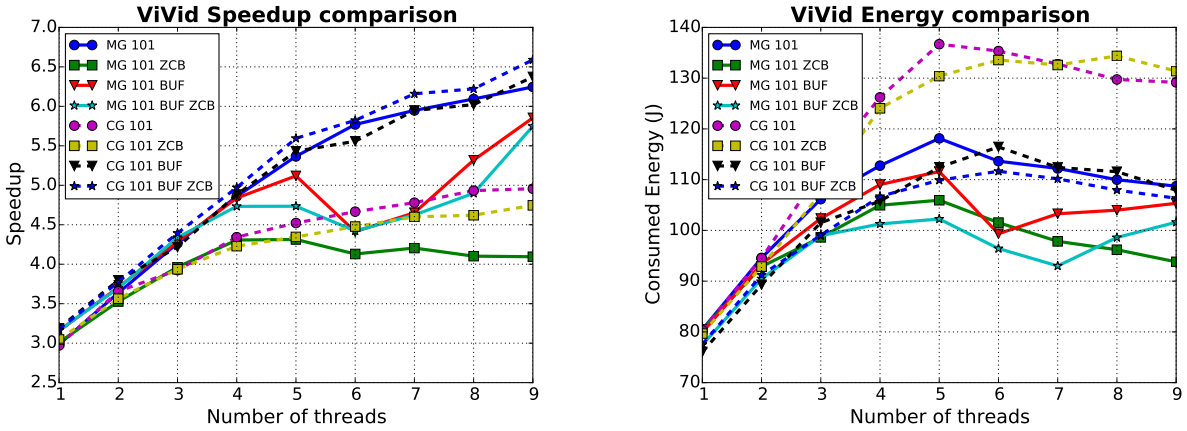
Una vez que hemos identificado que la mejor granularidad para este código y *SoC* es la MG, vamos a estudiar el impacto que tiene el mapeo de las etapas en los procesadores disponibles para dicha granularidad. Vamos a comparar el caso que consigue el mayor rendimiento (mapeo {1,0,1}) con el caso en el que se optimizan las comunicaciones entre *GPU* y *CPU* (mapeo {1,1,1}, donde todas las etapas pueden ser mapeadas en la *GPU* si el *item* que entra en el pipeline encuentra la *GPU* libre). La Fig. 8(a) muestra una comparación entre los experimentos `MG 101` y `MG 111`. En esta figura, como referencia, se muestran ademas los resultados de una ejecución homogénea solo *CPU* (`CPU MG`). La Fig. 8(b) muestra el desglose del consumo de energía para esos experimentos.

En este caso, ambos mapeos se comportan de una forma similar: al incorporar los `cores` del A15 (de 2 a 5 *threads*) se incrementa el rendimiento a costa de incrementar el consumo de energía (debido al aumento del consumo de energía del A15, como muestra la Fig. 8(b)). Cuando se añaden los `cores` del A7 (de 6 a 9 *threads*), se mejora el rendimiento y se reduce ligeramente el consumo de energía. De cualquier forma, con 9 *threads* el mapeo `MG 101` proporciona un ligero rendimiento superior (no apreciable en la figura) y un menor consumo de energía que `MG 111`. Las diferencias entre ambos mapeos son pequeñas debido a que la etapa 2 (etapa en la que difiere su mapeo) tiene una carga computacional muy pequeña: ésta representa menos del 5 % de la carga computacional total de un *item*.

D. Optimizaciones de memoria

El *SoC* Exynos utilizado en los experimentos no tiene una memoria *cache* compartida en su ultimo nivel entre la *CPU* y *GPU*. Esto hace que cuando ambos dispositivos trabajan simultáneamente, las operaciones de memoria compiten por el bus de memoria (AHB). En esta sección se explora una optimización que trata de minimizar el volumen de copia de datos entre *buffers* para reducir el tráfico en el *bus* de memoria. Además, se exploran otras optimizaciones para reducir el número de operaciones de gestión de memoria, lo cual reduce la sobrecarga del sistema. Se estudian las optimizaciones para las granularidades CG y MG, y para los mapeos en los que cada granularidad alcanzan el máximo rendimiento: `CG 101` y `MG 101`.

La Fig. 9 compara la aceleración y la energía consumida en los experimentos `CG 101` y `MG 101` (las



(a) (b)

Fig. 9
Estudio del impacto de diferentes optimizaciones de memoria en ViVid.

versiones base) con otros experimentos en los que se usan una serie de optimizaciones (las cuales pueden ser activadas en nuestro *framework* usando una opción en la inicialización de los *buffers* [6]). Los experimentos con las optimizaciones son los siguientes:

- `CG 101 ZCB` y `MG 101 ZCB`: Usan una opción (*Zero-Copy-Buffer*) disponible en los *buffers* de OpenCL para alojar los datos que son compartidos entre *CPU* y *GPU* en una zona de memoria accesible por ambos dispositivos. Esta optimización reduce el tráfico de memoria y la cantidad de copias entre *buffers* en la memoria física compartida.
- `CG 101 BUF` y `MG 101 BUF`: Usan un *pool* de *buffers* que son creados (y destruidos) una única vez antes de comenzar el cómputo (y a la finalización) del *pipeline*. Estos *buffers* son usados por la plantilla *pipeline* para pasar los datos entre las etapas del *pipeline*. Esta optimización evita la reserva y el desalojo de memoria cada vez que un *item* es procesado en el *pipeline*, eliminando la sobrecarga debida a llamadas del sistema.
- `CG 101 BUF ZCB` y `MG 101 BUF ZCB`: Usan las dos optimizaciones anteriores.

En la figura Fig. 9(a) se puede apreciar que si se aplica únicamente la optimización ZCB el rendimiento de la aplicación se degrada para ambas granularidades (CG y MG). La degradación del rendimiento para la granularidad MG, comparada con la versión base MG, es significativa, llegando al 34 % con 9 *threads*. Para la granularidad CG, la degradación sobre la versión CG base es menor, por debajo de un 4.2 % con 9 *threads*. Esta pérdida de rendimiento se debe a un problema de localidad: cuando la opción ZCB es utilizada, la implementación de OpenCL emplea un protocolo de acceso remoto a memoria directo, que evita que los datos sean cacheados (hemos observado que las ejecuciones de *CPU* también sufren una degradación de rendimiento con ZCB). Esta opción afecta en mayor medida a la granularidad MG, que pierde la ventaja que tenía al explotar la localidad de *cache*.

Sin embargo, la optimización BUF tiene un impacto positivo en la granularidad CG, mejorando el rendimiento de la versión CG base en un 28 %. En el caso de MG, esta optimización introduce un problema de desbalanceo de carga entre el A15 y el A17 con 6 y 7 *threads*. Esta perdida de rendimiento se empieza a recuperar con 8 y 9 *threads*, pero no lo suficiente para compensar la pérdida por desbalanceo de carga. Por ejemplo, con 9 *threads*, la pérdida de rendimiento de la versión BUF respecto de la versión base de MG es de un 6.2 %.

Es interesante hacer notar que la combinación de la optimización BUF con ZCB es la que ofrece el mayor rendimiento para CG: con 9 *threads*, se mejora el rendimiento en un 32.9 % respecto a la versión base. Para MG, la reducción en la sobrecarga de llamadas al sistema de BUF no compensa la pérdida de localidad que introduce ZCB: como resultado, el rendimiento de BUF+ZCB para 9 *threads* se reduce en un 7.9 % respecto de su versión base.

La Fig. 9(b) muestra el impacto de las optimizaciones sobre el consumo de energía. Todas las optimizaciones, excepto ZCB en CG, reducen el consumo de energía. En el caso de la granularidad MG y 9 *threads*, las optimizaciones ZCB, BUF y BUF+ZCB reducen el consumo de energía en un 13.7 %, 3.1 % y 6.5 % respectivamente, cuando se compara con la versión MG base. En particular, la optimización ZCB reduce la componente de consumo de energía del A15 e incrementa la componente de energía de la *GPU* respecto de la versión base (la *GPU* computa más *frames* lo que explica la mejora en el consumo de energía a pesar de la reducción del rendimiento). Para CG y 9 *threads*, ZCB incrementa ligeramente el consumo de energía en un 1.7 %, mientras que BUF y BUF+ZCF reducen la energía en un 16 % y 17.6 % respectivamente, cuando se compara con la versión base.

IV. Trabajos relacionados

Una aproximación para desarrollar aplicaciones de tipo *streaming* es mediante el uso de un lenguaje de programación con soporte para *streams*, como StreamIt [8]. Sin embargo, esos lenguajes no proporcionan soporte para sistemas heterogéneos CPU-GPU. La ejecución simultánea en *CPU* y *GPU* alcanza un mayor rendimiento que las ejecuciones "solo *CPU*" y "solo *GPU*" [9], pero los *frameworks* de programación que proporcionan soporte para realizar cómputo sobre arquitecturas heterogéneas, como Qilin [10], OmpSs [11] o StarPU [12], solo consideran el rendimiento como criterio a la hora de distribuir la carga computacional entre *CPU* y *GPU*. La diferencia entre los trabajos previos que hemos mencionado y nuestra propuesta, radica principalmente en que nuestra aproximación aborda aplicaciones de tipo *streaming* mientras que los trabajos previos se centran en el paradigma de paralelismo de datos. Además, en este trabajo también se estudia la eficiencia energética, aspecto no considerado en los trabajos previos mencionados.

V. Conclusiones

En este trabajo, se presenta la *API* de un framework que nos permite ejecutar aplicaciones de tipo *streaming* sobre procesadores heterogéneos integrados en el mismo chip. Nuestra plantilla permite la configuración de tres parámetros básicos: la granularidad explotada en cada etapa por el *multicore* (CG or MG), la asignación de las etapas a los procesadores (*CPU* or *GPU*), y el número de *threads*. Usamos este framework para evaluar como estos tres parámetros impactan sobre el rendimiento y el consumo de energía de un procesador heterogéneo. El procesador sobre el que hemos realizado todas las pruebas es un Exynos 5 Octa que se caracteriza por tener tres procesadores diferentes: una *GPU*, un *quad-core* A15 y un *quad-core* A7. Los resultados muestran que las ejecuciones heterogéneas son las que ofrecen un mayor rendimiento aunque se incrementa el consumo de energía del chip cuando los procesadores A15 comienza a trabajar. Teniendo en cuenta la eficiencia energética, los procesadores A7 permiten incrementar un poco el rendimiento mientras que se reduce ligeramente el consumo de energía. Los experimentos también muestran que la granularidad MG tiende a consumir menor energía en los A15 debido a una mejor explotación de la localidad cache. Además, se exploran varias optimizaciones de memoria que muestran una dependencia con el tipo de granularidad, aunque en general ayudan a mejorar la eficiencia energética.

Agradecimientos

Este trabajo ha sido financiado por los siguientes proyectos españoles: TIN2010-16144 y TIN 2013-42253-P por parte del Ministerio de Ciencia e Innovación, y P11-TIC-08144 por parte Junta de la Andalucía.

References

[1] J. Clemons, H. Zhu, S. Savarese, and T. Austin, "Mevbench: A mobile computer vision benchmarking suite," in *IISWC*, Nov 2011, pp. 91–102.

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

[3] M. Dikmen, D. Hoiem, and T. S. Huang, "A data driven method for feature transformation," in *Proc. of CVPR*, 2012, pp. 3314–3321.

[4] M. Jones and P. Viola, "Fast multi-view face detection," *Mitsubishi Electric Research Lab TR-20003-96*, vol. 3, 2003.

[5] A. Vilches, A. Navarro, R. Asenjo, F. Corbera, R. Gran, and M. Garzaran, "Mapping streaming applications on commodity multi-CPU and GPU on-chip processors," *IEEE Tran. on Parallel and Distributed Systems*, 2015, doi: 10.1109/TPDS.2015.2432809.

[6] A. Rodriguez, A. Vilches, A. Navarro, R. Asenjo, F. Corbera, R. Gran, and M. Garzaran, "Productive interface to map streaming applications on heterogeneous processors," Comput. Architecture Dept., Tech. Rep., 2015, http://www.ac.uma.es/ asenjo/research/.

[7] F. Corbera, A. Rodriguez, R. Asenjo, A. Navarro, A. Vilches, and M. J. Garzaran, "Reducing overheads of dynamic scheduling on heterogeneous chips," in *Workshop on High Performance Energy Efficient Embedded Systems (HIP3ES)*, 2015.

[8] R. Soule, M. I. Gordon, S. Amarasinghe, R. Grimm, and M. Hirzel, "Dynamic expressivity with static optimization for streaming languages," in *Intl Conf on Distributed Event-Based Systems*, June 2013.

[9] C. Yang *et al.*, "Adaptive optimization for petascale heterogeneous CPU/GPU computing," in *CLUSTER*, 2010, pp. 19–28.

[10] C.-K. Luk, S. Hong, and H. Kim, "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *MICRO 42*, 2009, pp. 45–55.

[11] J. Planas *et al.*, "Self-adaptive OmpSs tasks in heterogeneous environments," in *Proc. of IPDPS*, 2013.

[12] C. Augonnet *et al.*, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurr. Comput. : Pract. Exper.*, vol. 23, no. 2, Feb. 2011.