Trabajo Fin de Grado en Grado de Ingeniería de Computadores

Evaluación del rendimiento de un planificador de tareas sobre una plataforma heterogénea ARM+DSP de Texas Instruments

Realizado por

Bermúdez Blanco, Javier

Director

Igual Peña, Francisco Daniel

Autorización de difusión

El abajo firmante *Javier Bermúdez Blanco*, alumno en el Grado de Ingeniería de Computadores, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando a su autor, el presente Trabajo de Fin de Grado: "Evaluación del rendimiento *de un planificador de tareas sobre una plataforma heterogénea ARM+DSP de Texas Instruments"*, realizado durante el curso académico 2015-2016, bajo la dirección de *Francisco Igual Peña*.

Así mismo autoriza a la Universidad Complutense de Madrid a que sea depositado en acceso abierto en el *repositorio institucional e-prints complutense* con el objeto de incrementar la difusión, uso e impacto del TFG en Internet y garantizar su preservación y acceso a largo plazo.

Índice de contenidos

Palabras clave

- OmpSs
- CUDA
- OpenCL
- Arquitecturas heterogéneas
- Paralelismo a nivel de tareas
- Consumo energético
- Procesadores Gráficos (GPUs)
- Procesadores Digitales de Señal (DSPs)

Keywords

- OmpSs
- CUDA
- OpenCL
- Heterogeneous architectures
- Task parallelism
- Energy consumption
- Graphics Processors (GPUs)
- Digital Signal Processors (DSPs)

Resumen

El presente trabajo estudia la viabilidad a la hora de aplicar un modelo de programación basado en la extracción de paralelismo a nivel de tareas sobre distintas arquitecturas heterogéneas basadas en un procesador multinúcleo de propósito general acelerado con uno o más aceleradores hardware. Se ha implementado una aplicación completa cuyo objetivo es la detección de bordes en una imagen (implementando el Algoritmo de Canny), y se ha evaluado en detalle su rendimiento sobre distintos tipos de arquitecturas, incluyendo CPUs multinúcleo de última generación, sistemas multi-GPU y una arquitectura objetivo basada en procesadores ARM Cortex-A15 acelerados mediante un DSP C66x de la compañía Texas Instruments. Los resultados experimentales demuestran la viabilidad de este tipo de implementación también para arquitecturas heterogéneas novedosas como esta última, e ilustran la facilidad de programación que introduce este tipo de modelos de programación sobre arquitecturas de propósito específico.

Abstract

This work studies the possibility of applying programming models based on the extraction of task parallelism on different heterogeneous architectures based on multi-core processors accelerated with one or more hardware accelerators. We have implemented a complete application for edge detection (Canny Algorithm), and we have evaluated in detail the performance on different architectures, including novel multi-core CPUs, systems equipped with multiple GPUs and a target architecture based on ARM Cortex-A15 processors accelerated through a C66x DSP manufactured by Texas Instruments. The expermiental results validate the usage of the aforementioned programming models also for novel heterogeneous architectures, and illustrate the ease of programming introduced by this kind of programming models on specific-purpose architectures.

1 Introducción y objetivos

En este capítulo se detalla la motivación principal del trabajo realizado, así como los principales objetivos planteados para su desarrollo.

Durante los últimos años, las exigencias computacionales dictadas por los problemas surgidos en ciencia e ingeniería han aumentado la capacidad de cálculo de los procesadores, con el fin de realizar cálculos y simulaciones cada vez más complejas, minimizando el tiempo de respuesta.

Tradicionalmente, la Ley de Moore [2], que determina el número de transistores que es posible integrar en una misma superficie de silicio, se ha complido hasta la fecha. Sin embargo, el incremento en frecuencia que posibilita, haciendo cada vez más rápidos los procesadores, se vio frenado en la pasada década, surgiendo como respuesta el concepto de procesadores multinúcleo. Este tipo de procesador replica la cantidad de unidades de procesamiento, haciendo posible que aquellos programas que puedan explotar este nivel de paralelismo vean aumentado su rendimiento generación tras generación.

Sin embargo, el uso de procesadores multinúcleo también ha visto frenado su desarrollo en los últimos años, siendo su consumo energético una de las principales barreras de cara a su evolución. En respuesta al creciente consumo energético de las arquitecturas de altas prestaciones, en los últimos años ha surgido un gran interés por el uso de plataformas heterogéneas, con especial énfasis no sólo en el rendimiento, sino en la reducción del consumo energético y, por tanto, en la mejora de la eficiencia energética de las arquitecturas.

El uso de arquitecturas heterogéneas es, por tanto, una tendencia creciente de cara a construir grandes supercomputadores que puedan responder a las demandas computacionales de la ciencia y la ingeniería. De entre este tipo de plataformas, destaca el uso de aceleradores hardware, que se adaptan de forma óptima a cierto tipo de aplicaciones, y aceleran el cómputo de ciertas partes de los algoritmos. Un ejemplo concreto es el uso de procesadores gráficos (GPUs), que en los últimos años ha emergido como un estándar a la hora de realizar implementaciones de alto rendimiento para cálculo de propósito general.

Aún así, aunque más eficientes desde el punto de vista computacional y energético, el uso de aceleradores y procesadores multinúcleo cada vez más potentes (y por tanto, consumiendo mayores potencias), ha hecho resurgir la preocupación por la imposibilidad de construir grandes supercomputadores con un coste energético asumible. De hecho, se calcula que, de seguir la tendencia actual en la construcción de supercomputadores basados en aceleradores hardware, el coste energético asociado a cada centro de datos en pocos años será sencillamente inasumible.

En respuesta a esto, se están estudiando nuevas tendencias a la hora de construir este tipo de plataformas que combinen, a la vez, gran eficiencia energética y prestaciones razonables. Una de las tendencias es el uso de procesadores y aceleradores de bajo coste y consumo, típicamente desarrollados para el mercado móvil, reutilizando y explotando sus características para un uso mucho más específico. Ejemplos de esta tendencia son los procesadores ARM, acelerados con procesadores gráficos de bajo consumo, u otro tipo de aceleradores como procesadores digitales de señal (DSPs). En la actualidad, existe gran interés en estudiar la viabibilidad de este tipo de

arquitecturas para construir supercomputadores con mayor eficiencia energética que los desarrollados actualmente.

De forma paralela, sin embargo, surge un problema adicional: la facilidad de programación. La cantidad de paradigmas de programación paralela existentes, y la dificultad a la hora de desarrollar códigos para arquitecturas específicas (por ejemplo, GPUs o DSPs), hace que se estén investigando nuevas técnicas, lenguajes y paradigmas de programación que permitan explotarlos de forma eficiente y sencilla de cara al desarrollador. Tecnologías como CUDA [3] u OpenCL [1] han surgido en respuesta a esta necesidad. Sin embargo, todavía resulta difícil realizar una computación realmente heterogénea de forma sencilla, sin que el programador esté a cargo de detalles de bajo nivel (por ejemplo, transferencias de datos, gestión de distintos espacios de memoria, adaptación de los códigos a cada arquitectura, etc.)

En este trabajo, se ha estudiado el uso de un paradigma de programación específico (programación paralela a nivel de tareas) sobre una arquitectura relativamente novedosa: un procesador heterogéneo equipado con núcleos ARM de propósito general y acelerado mediante un DSP multinúcleo. En este trabajo, se expondrán las ventajas de este tipo de paradigma sobre una aplicación concreta (una implementación del Algoritmo de Canny), evaluando su eficiencia y facilidad de programación sobre esta arquitectura en comparación con sistemas heterogéneos "clásicos" basados en procesadores multinúcleo de propósito general y GPUs de gran potencia.

Introduction and goals

During the last years, the computational requirements dictated by the problems encountered in science and engineering have increased the computing capacity of processors, in order to perform calculations and increasingly complex simulations, minimizing response time.

Traditionally, Moore's Law [2], which determines the number of transistors that can be integrated on the same silicon surface has been met. However, the increase in frequency that enables making ever faster processors, was slowed in the past decade, emerging as a response the concept of multicore processors. This type of processor replicates the number of processing units, making it possible for programs that can exploit this increased level of parallelism improve their performance generation after generation..

However, the use of multicore processors also been hampered its development in recent years, energy consumption being one of the main barriers facing their evolution. In response to the growing energy consumption of architectures high performance, in recent years there has been great interest in the use of heterogeneous platforms, with an emphasis not only on performance but on reducing energy consumption and, therefore, in improving the energy efficiency of architectures.

The use of heterogeneous architectures is therefore facing a growing trend to build large supercomputers that can meet the computational demands of science and engineering. Among these platforms, it emphasizes the use of hardware accelerators, which are adapted optimally to certain types of applications, and accelerate the computation of certain parts of the algorithms. A concrete example is the use of graphics processors (GPUs), which in recent years has emerged as a standard when Implementations high performance general purpose computing.

Still, although more efficient computationally and energy perspective, the use of accelerators and multicore processors increasingly powerful (and therefore consuming higher power), has revived concerns about the inability to build large supercomputers with asumible energy costs. In fact, it is estimated that, to follow the current trend in building supercomputers based hardware accelerators, the energy cost associated with each data center in a few years will simply be unaffordable.

In response to this, they are studying new trends in building such platforms that combine, at the same time, high energy efficiency and reasonable performance. One trend is the use of processors and accelerators low cost and consumption, typically developed for the mobile market, re-using and exploiting its features for a more specific use. Examples of this trend are the ARM processors, graphics accelerated with low-power processors, or other accelerators such as digital signal processors (DSPs). Currently, there is great interest in studying the viabibilidad of such architectures to build more energy-efficient supercomputers that developed currently.

In parallel, however, a further problem arises: the ease of programming. The amount of paradigms existing in parallel programming, and the difficulty of developing codes for specific architectures (eg, GPUs or DSPs) causes being investigated new techniques, languages and programming paradigms that allow to exploit efficiently and simple facing the developer. Technologies such as CUDA [3] or OpenCL [1] have emerged in response to this need. However, it is still difficult to make a truly heterogeneous computing easily, without the programmer is responsible for low-level details (for example, data transfers, management of different memory spaces, adaptation of codes to each architecture, etc.)

In this work, we have studied the use of a specific programming paradigm (task-level parallelism) on a relatively new architecture: a heterogeneous processor equipped with ARM core general purpose and accelerated through a multicore DSP. The advantages of this kind of paradigm for a particular application (an implementation of the Canny edge detector) will be exposed, assessing their efficiency and ease of programming on this architecture compared to "classical" heterogeneous systems based on multicore processors purpose general and powerful GPUs.

Objetivos generales

El objetivo general del trabajo realizado es el estudio de la viabilidad, rendimiento y eficiencia energética del uso de un paradigma de programación paralela a nivel de tareas sobre una arquitectura heterogénea basada en procesadores ARM multinúcleo y acelerada por DSPs para una aplicación concreta.

Este objetivo general se divide en un conjunto de objetivos específicos:

- Selección de una aplicación de interés (Algoritmo de Canny) e implementación de un código secuencial básico, acelerado usando paradigmas CUDA y OpenCL.
- Modificación del algoritmo para su ejecución bajo un paradigma de paralelismo a nivel de tareas, identificando dependencias de datos entre las mismas.
- Evaluación del rendimiento de los distintos paradigmas sobre distintas arquitecturas heterogéneas.
- Evaluación de la eficiencia energética y rendimiento de un procesador heterogéneo ARM + DSP utilizando OmpSs, un modelo de programación paralela a nivel de tareas desarrollado por el Barcelona Supercomputing Center (BSC).

2 Modelos de programación para arquitecturas heterogéneas.

En este capítulo se introducen los tres distintos lenguajes de programación que se usan en el proyecto: CUDA, OpenCL y OmpSs, haciendo especial hincapié en las ventajas e inconvenientes de cada uno de ellos, así como en sus principales similitudes y diferencias.

CUDA

CUDA (*Compute Unified Device Architecture)* es un modelo de programación y una arquitectura de cómputo paralelo creado por NVIDIA en el año 2006. CUDA funciona en todas las GPUs Nvidia de la serie G8x en adelante (a diferencia de OpenCL, que funciona también en otras plataformas paralelas como CPUs o DSPs). Dado su éxito, CUDA se ha convertido en un estándar de facto en el desarrollo de códigos de propósito general sobre plataformas gráficas de propósito específico.

De cualquier modo, la mayoría de las tarjetas gráficas que soportan CUDA también soportan OpenCL; por lo tanto los programadores pueden escoger escribir código para cualquiera de las dos plataformas cuando desarrollan para hardware NVIDIA. Es habitual, sin embargo, realizar desarrollos en CUDA en estos casos, por su mayor adaptación a los recursos hardware subyacentes.

Como definición de arquitectura, la CPU es la unidad central de procesamiento de una computadora y la GPU es unidad de procesamiento gráfico. Al compartir la carga de procesamiento con el GPU (en lugar de sólo usar la CPU), CUDA permite a los programas mejorar su rendimiento. NVIDIA provee APIs de CUDA que permite desarrollar el software en C, C++, Python y Fortran.

Características

- Lenguaje de programación de alto nivel, basado en estándares abiertos
- MultiGPUs controlados, es decir, una sola CPU puede ser capaz de controlar varias GPU, lo cual permite un amplio campo de posibilidades en cuanto a la mejora de rendimiento.
- Cache de datos en paralelo, multiplicando el ancho de banda efectivo y reduciendo las latencias, al permitir que los grupos de procesadores trabajen juntos en el uso de la información contenida en la cache local.
- Transferencia de datos a través del bus PCI Express. Gracias a su elevado ancho de banda, especialmente en sus últimas versiones, las aplicaciones de cálculo pueden explotar unas tasas de transferencia de datos bastante elevadas, permitiendo unas lecturas más rápidas de y hacia la GPU.

Limitaciones

- Un código C válido puede ser rechazado debido a las limitaciones del propio hardware.
- Las primeras versiones de CUDA no admiten recursividad, punteros a funciones, y otras limitaciones se están desapareciendo.
- En ocasiones puede existir un cuello de botella (es decir, la capacidad de procesamiento del dispositivo es mayor a la capacidad del bus) entre la CPU y la GPU por los anchos de banda de los buses y sus latencias.
- La correcta selección del número y disposición de threads es crítica para obtener un elevado rendimiento.
- Una de los principales dificultades al trabajar en CUDA, es la transferencia de datos y más aún si se dispone de más de una GPU, especialmente en situaciones donde una GPU requiera datos residentes en otra GPU. Todos estas inconvenientes se deben tener presentes a la hora de programar en CUDA y son responsabilidad del programador. Es decir, no basta con enviar reservar datos, hay que saber qué dependencias existen entre tareas, y cómo reducir el número de transferencias.

CUDA como modelo de programación

El modelo de programación CUDA está creado para trabajar con implementaciones que exploten un nivel de paralelismo alto o muy alto. Toda función CUDA (llamada típicamente kernel) debe comenzar con la palabra clave __global__, identificándose como una función kernel.

```
__global__ void prueba(int a)
{
  // Cuerpo del kernel.
}
```

Estas funciones deben devolver void, y no soportan llamadas recursivas.

El objetivo de CUDA es conseguir un paralelismo alto, para ello el modelo de programación debe de aprovechar al máximo la cantidad de thread que se tiene. Por ejemplo si tenemos una multiplicación de matrices, cada thread podría ocuparse de una posición de la matriz resultante, así calcular todas las posiciones paralelamente.

Gestión de memoria y transferencias

La funciones de gestión de memoria en CUDA son:

- *cudaMalloc()*, reserva de espacio en la memoria global de la GPU, recibiendo dos

parámetros:

- Puntero.
- Tamaño de la memoria reservada.

- *cudaMemset(),* inicializa a un valor dado. Parámetros:
  - Dirección
  - Valor
  - Cantidad
- *cudaFree()*, liberta la memoria asociada a un puntero en el memoria global del dispositivo.
- *cudaMemcpy()*, copia (transfiere a través del bus PCIExpress) los datos de la CPU (RAM) a GPU (memoria global) y viceversa, recibiendo cuatro parámetros:
  - Puntero destino.
  - Puntero origen.
  - Número de bytes a copiar.
  - Tipo de transferencia:
    - cudaMemcpyHostToDevice: para realizar transferencias desde host a dispositivo.
    - cudaMemcpyDeviceToHost: para realizar transferencias desde dispositivo a host.
    - cudaMemcpyDeviceToDevice: para realizar copias de memoria entre distintas zonas de memoria global (memoria de dispositivo).

Ejemplo de reserva de memoria y transferencia de datos.

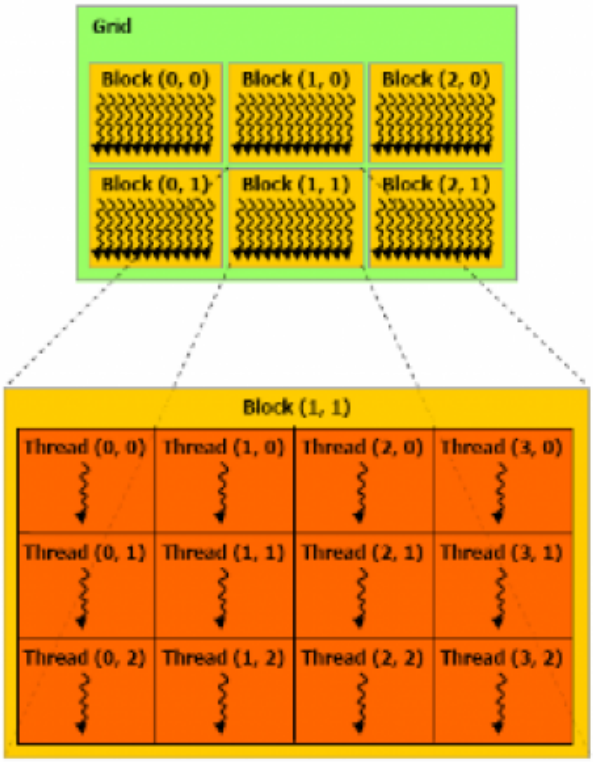
Se quiere reservar, inicializar y liberar espacio de una matriz de 32x32 elementos **int,** y transferir.

```
#define SIZE_BLOCK 32
int *array;
int *M;
int size = SIZE_BLOCK*SIZE_BLOCK*sizeof(int);
cudaMalloc( (void**)&array, size);  // Reserva de memoria en dispositivo (mem. global)
cudaMemset( array,0, size);         // Inicialización de memoria reservada.
cudaMemcpy(M,array, size, cudaMemcpyDeviceToHost); // Transferencia de datos a host.
cudaFree (array);                   // Liberación de memoria.
```

Jerarquía de threads

Las GPUs Nvidia constan de cientos (en algunos casos, miles) de unidades de cómputo, típicamente llamadas núcleos), cada uno dedicado a la ejecución de un flujo de ejecución independiente (hilo de ejecución). Por lo tanto, permiten, y explotan, un elevado grado de paralelismo de grano fino, siendo estas sus características:

- Existe un súper grupo de bloques de threads, llamado grid.
- Cada bloque de threads está formado por un conjunto de threads.
- Cada bloque dentro de un grid se identifica de forma única mediante un identificador uni, bi o tridimensional, en función de las necesidades del problema.



- Cada thread dentro de un bloque posee un identificador único, nuevamente uni, bi o tridimensional. De forma automática, cada thread instancia la variable *threadIdx,* cuyas componentes (x, y, z) identifican de forma unívoca al hilo dentro de su bloque. El identificador global del thread se puede obtener fácilmente a partir de dicha información, y del identificador del bloque al que pertenece; por ejemplo, trabajando en una dimensión, un identificador único para un determinado hilo puede obtenerse mediante expresiones sencillas de tipo:

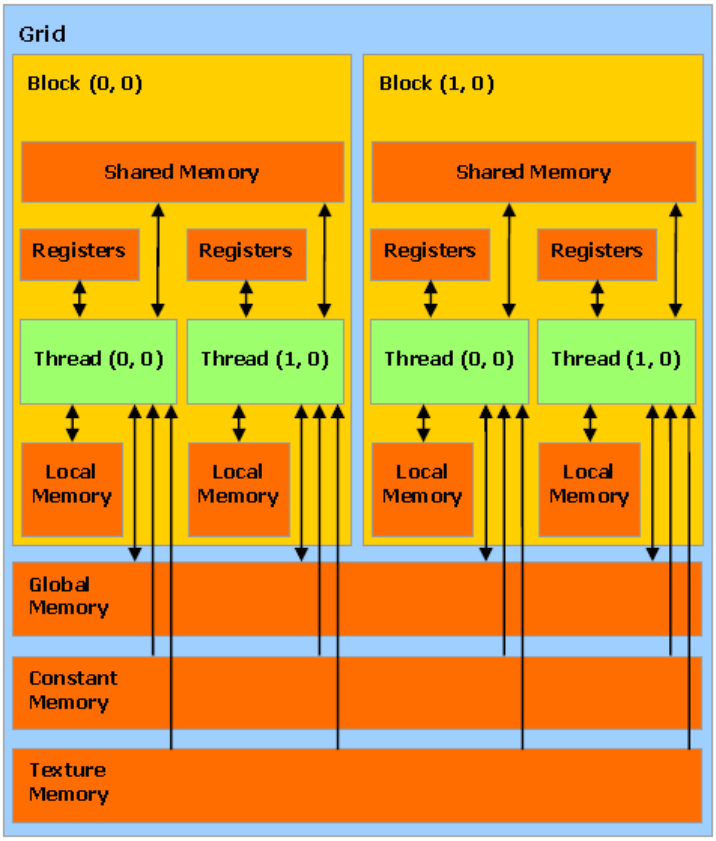
  *int threadX = threadIdx.x + blockDim.x * blockIdx.x;*

- Los threads de la GPU son ligeros, con poca o nula sobrecarga de planificación y presentan cambios de contexto rápidos; en cambio los threads de CPU son pesados, tiene sobrecarga de planificación y cambios de contexto mucho más lentos.
- Dado un identificador global único, típicamente se utiliza dicha información para realizar un reparto de aquellos datos o bloques de datos sobre los que trabajará el hilo de ejecución. Desde este punto de vista, las GPUs siguen un paradigma SIMD (Simple Instruction Multiple Data): cada hilo de ejecución ejecuta exactamente la misma instrucción en un determinado punto de la ejecución, pero trabajando sobre distintos datos en función de su identificador.

Espacios de memoria

Las GPUs compatibles con CUDA presentan distintas zonas de memoria, cada una de ellas accesible a nivel de hilo, bloque o grid de ejecución:

- Registros: Disponible y accesible únicamente por el thread al que está asociada, en modo lectura/escritura.
- Memorial local: Disponible y accesible en modo lectura/escritura por todos los hilos que componen un mismo bloque de hilos. Su latencia y ancho de banda es similar a la de los registros, aunque su tamaño está limitado a pocos Kbytes por multiprocesador.
- Memoria global: Para lectura/escritura desde cualquier bloque. Típicamente de gran tamaño, permite comunicar hilos pertenecientes a distintos bloques.
- Memoria constante: Región de la memoria global, sólo para lectura y accesible desde cualquier bloque de hilos.
- Memoria textura: Región de la memoria global, sólo para lectura y accesible desde cualquier bloque de hilos, cacheable y utilizable a través de APIs específicas.



Ejemplo de código CUDA

Se muestra a continuación un ejemplo sencillo de ejecución de un código CUDA. El programa se divide en dos partes: programa principal, ejecutado en CPU y kernel, ejecutado en GPU por tantos hilos como se deseen. El programa principal, mostrado a continuación, se divide en tres partes principales:

1. Reserva de un buffer de memoria de tamaño MEM_SIZE bytes, tanto en RAM como en memoria global, a través de funciones específicas en CPU y GPU.
2. Configuración de la ejecución (que en este caso incluye un bloque de hilos formado por un único hilo), e invocación del kernel utilizando sintaxis CUDA.
3. Copia de los datos inicializados en GPU de vuelta a memoria RAM, previa a la impresión por pantalla del contenido del buffer.

Programa principal:

```
1 #include <stdio.h>
2
3 #define MEM_SIZE 128
4
5 int main(){
6
7     char buf[MEM_SIZE];
8     char buffer[MEM_SIZE];
9
10    buf = malloc(MEM_SIZE*sizeof(char));
11    cudaMalloc( string, MEM_SIZE*sizeof(char));
12
13    dim3 dimBlock(blocksize,1);
14    dim3 dimGrid(1,1);
15    hello<<<dimGrid, dimBlock>>>(buffer);
16
17    cudaMemcpy(buf,buffer,MEM_SIZE*sizeof(char),cudaMemcpyDeviceToHost);
18    cudaFree(buffer);
19
20    printf("%s\n",a);
21
22    return EXIT_SUCCESS;
23 }
```

Código GPU (kernel):

```
1 __global__ void hello(char* buffer){
2     buffer[0] = 'H';
3     buffer[1] = 'e';
4     buffer[2] = 'l';
5     buffer[3] = 'l';
6     buffer[4] = 'o';
7     buffer[5] = ',';
8     buffer[6] = ' ';
9     buffer[7] = 'W';
10    buffer[8] = 'o';
11    buffer[9] = 'r';
12    buffer[10] = 'l';
13    buffer[11] = 'd';
14    buffer[12] = '!';
15    buffer[13] = '\0';
16 }
```

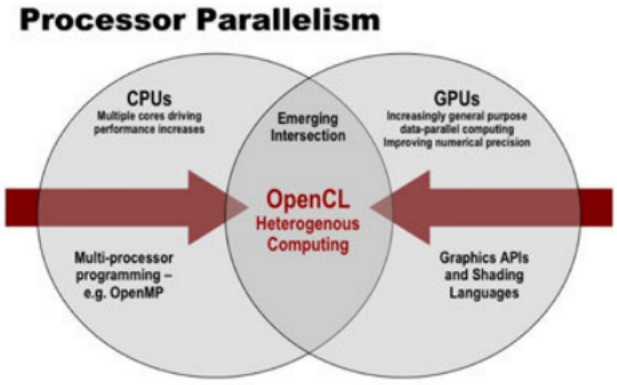
OpenCL

OpenCL [1] son las siglas de Open Computing Language, lenguaje de computación abierto. Es el primer estándar de programación verdaderamente abierto. Permite crear aplicaciones que pueden ejecutarse tanto en unidades centrales de procesamiento como unidades de procesamiento gráfico.

Para muchos es considerada la API que tiene mayor probabilidad de ejecutarse aplicaciones que funcionen usando las GPUs, al ser multiplataforma y no tener restricciones tanto de hardware como de sistema operativo.

La principal diferencia con CUDA es que OpenCL puede ser ejecutado en cualquier dispositivo que implemente el estándar,siendo abierto no es únicamente Nvidia. Es decir, que OpenCL además de poder ejecutarse únicamente en GPUs, puede ser ejecutado en CPUs.

OpenCL™ se desarrolló en un comité de estándares abiertos con representantes de los principales proveedores de la industria y les ofrece a los usuarios lo que han estado reclamando: una solución no de propiedad exclusiva, de varios proveedores, para acelerar las aplicaciones en las CPU, GPU y APU. AMD, un patrocinador inicial de OpenCL™ e innovador y proveedor líder de CPU, APU y GPU de alto rendimiento, está en una posición exclusiva en esta industria para ofrecer una plataforma de aceleración completa para OpenCL™.



Características

- Soporte del modelo de programación paralela a nivel de datos yde tareas
- Emplea un subconjunto del lenguaje de programación C99 + extensiones para programación paralela eficaz y segura.
- Permite la interacción eficiente con APIs gráficas como OpenGL, OpenGLES y DirectXentre otras.
- Define requisitos numéricos basados en el estándar IEEE 754.

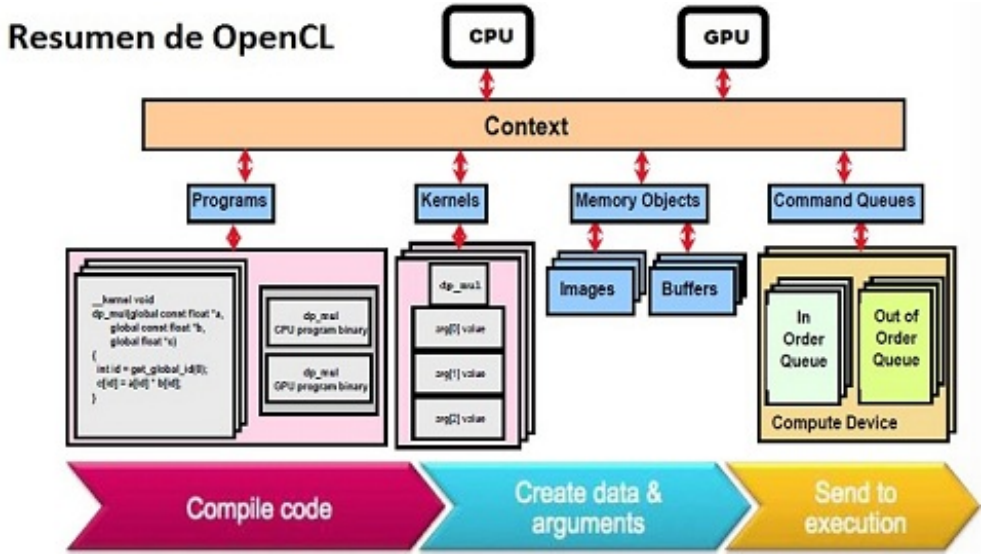
Programación en OpenCL

Todas las funciones de OpenCL se agrupan en los ficheros llamados kernels (al igual que en CUDA), siendo ficheros con la extensión "cl", siendo el lenguaje basado en C. Conceptualmente, el modelo de programación es muy similar a CUDA.

Todas las funciones OpenCl llevan el acrónimo __kernel para ser identificadas. Los punteros deben de llevar el acrónimo __global si apuntan a una zona de memoria global, o __local si son zonas de memoria local. Por ejemplo:

```
__kernel void prueba(int __global *a)
```

Los programas de OpenCL se compilan formando código objeto para la CPU, y para la GPU. El código objeto que se ejecuta en la CPU determina los kernels (cantidad mínima de código ejecutable) a ejecutar en cada una de las GPUs o dispositivos compatibles, compilándose éstos en tiempo de ejecución. Precisamente en tiempo de ejecución, OpenCL genera un contexto (Context) que se asocia a la unidad que se encargará de ejecutar el programa. Este contexto se encarga de manejar los programas, los kernels, los objetos de memoria y las colas de comandos, y está típicamente asociado a un dispositivo concreto.

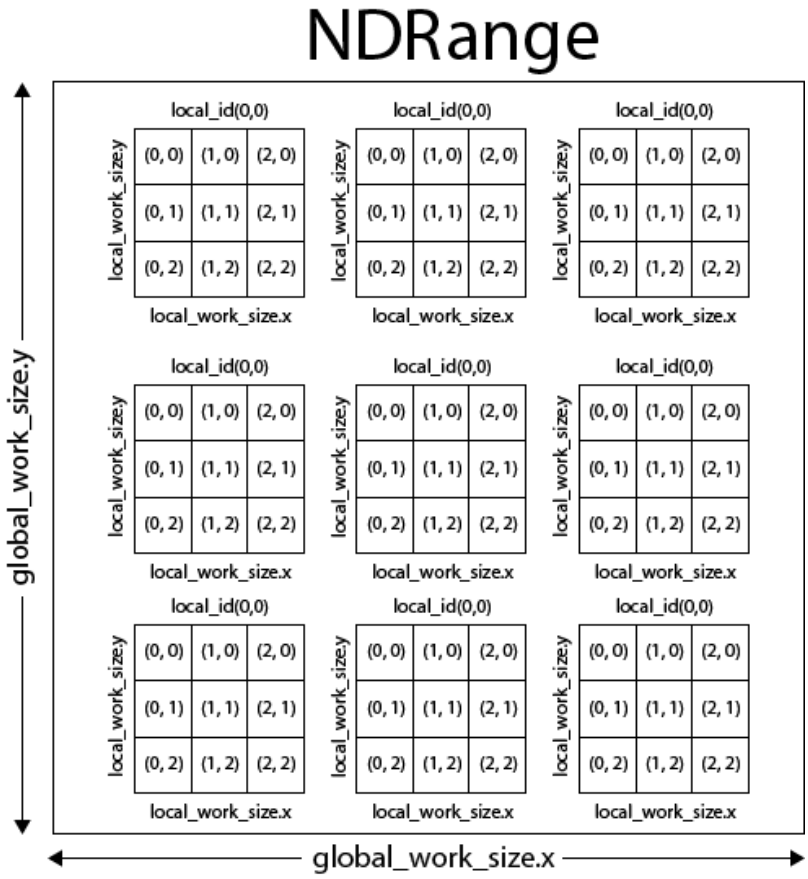


Jerarquía de threads

A diferencia de CUDA, OpenCL permite programar en todo tipo de dispositivos, en concreto si tenemos una GPUs (pudiendo ser o no, Nvidia),constan de cientos de núcleos los cuales tiene el procesamiento de un hilo cada uno, por lo tanto permitiendo un grado de paralelismo bastante alto, siendo estas sus características:

- Cada hebra es un work-item.
- Cada work-item se ejecuta en paralelo en un núcleo siguiendo un paradigma SIMD.

- El conjunto de work-items de un mismo núcleo se llama work-group (es decir, equivalente al concepto de bloque de hilos en CUDA).
- Cada work-group comparte memoria local y permite comunicar y sincronizar los work-items que lo componen.
- Cada work-item posee un identificador único en la configuración global de ejecución (es decir, no es necesario computarlo explícitamente, como sí ocurría en CUDA).

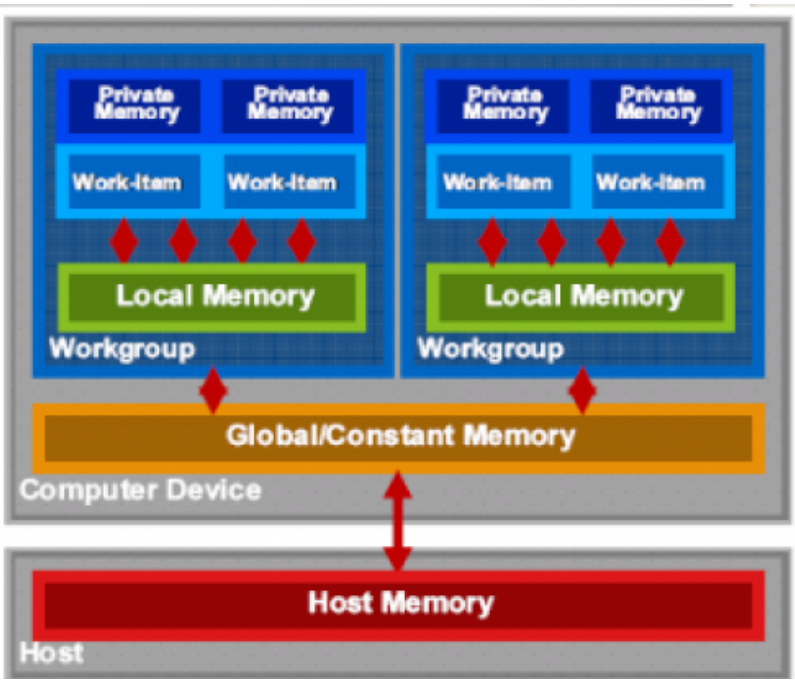


Gestión de memoria

La división por work-group permite tener memoria privada para cada uno de ellos y a su vez memoria compartida con los demás, siendo estas sus características:

- Memoria privada: Disponible y accesible únicamente por el work-item al que está asociada.
- Memorial local: Para lectura y escritura, accesible desde un único work-group (variables compartidas por work-items dentro de un work-group).

- Memoria global: Para lectura/escritura desde cualquier work-item o work-group. Representa a la memoria de cada dispositivo.
- Memoria constante: Región de la memoria global, sólo para lectura desde work-items. Es constante durante la ejecución del kernel, y puede ser leída y escrita por la aplicación host.
- Memoria del Host: Memoria asociada a la CPU que actúa como host.



Código de ejemplo OpenCL: programa principal.

Nótese la complejidad del código OpenCL en su parte host equivalente al desarrollado en CUDA. Esta complejidad demuestra el compromiso entre portabilidad del código, que ahora es compatible con cualquier plataforma paralela con soporte OpenCL, y facilidad de programación. Aunque fuera del alcance del presente trabajo, cabe destacar la cantidad de invocaciones a la API de OpenCL desde el host para configurar contextos de ejecución, dispositivos, colas de comandos, creación de buffers, transferencias de datos y configuraciones de ejecución del kernel.

```c
#include <stdio.h>
#include <stdlib.h>
#include <CL/cl.h>

#define MEM_SIZE 128
#define MAX_SOURCE_SIZE 0x100000

int main(){
    cl_device_id device_id = NULL;
    cl_context context = NULL;
    cl_command_queue command_queue = NULL;
    cl_mem memobj = NULL;
    cl_program program = NULL;
    cl_kernel kernel = NULL;
    cl_platform_id platform_id = NULL;
    cl_uint ret_num_devices;
    cl_uint ret_num_platforms;
    cl_int ret;

    char buffer[MEM_SIZE];

    FILE *fp;
    char fileName[] = "./kernel.cl";
    char *source_str;
    size_t source_size;

    /* Load the source code containing the kernel*/
    fp = fopen(fileName,"r");
    if (!fp){
        fprintf(stderr, "Failed to load kernel.\n");
        exit(1);
    }
    source_str = (char*)malloc(MAX_SOURCE_SIZE);
    source_size = fread(source_str,1,MAX_SOURCE_SIZE,fp);
    fclose(fp);

    /* Get Platform and Device Info */
    ret = clGetPlatformIDs(1,&platform_id,&ret_num_platforms);
    ret = clGetDeviceIDs(platform_id,CL_DEVICE_TYPE_DEFAULT,1,&device_id,&ret_num_devices);

    /* Create OpenCL context */
    context = clCreateContext(NULL,1,&device_id,NULL,NULL,&ret);

    /* Create Command Queue */
    command_queue = clCreateCommandQueue(context,device_id,0,&ret);

    /* Create Memory Buffer */
    memobj = clCreateBuffer(context,CL_MEM_READ_WRITE,MEM_SIZE*sizeof(char),NULL,&ret);

    /* Create Kernel Program from the source */
    program = clCreateProgramWithSource(context,1,(const char **)&source_str,(const size_t *)&source_size, &ret);

    /* Build Kernel Program */
    ret = clBuildProgram(program,1,&device_id,NULL,NULL,NULL);

    /* Create OpenCL Kernel */
    kernel = clCreateKernel(program,"hello",&ret);

    /* Set OpenCL Kernel Parameters */
    ret = clSetKernelArg(kernel,0,sizeof(cl_mem),(void *)&memobj);

    /* Execute OpenCL Kernel */
    ret = clEnqueueTask(command_queue,kernel,0,NULL,NULL);

    /* Copy results from the memory buffer */
    ret = clEnqueueReadBuffer(command_queue,memobj,CL_TRUE,0,MEM_SIZE*sizeof(char),buffer,0,NULL,NULL);

    /* Display Result */
    puts(buffer);

     /* Finalization */
     ret = clFlush(command_queue);
     ret = clFinish(command_queue);
     ret = clReleaseKernel(kernel);
     ret = clReleaseProgram(program);
     ret = clReleaseMemObject(memobj);
     ret = clReleaseCommandQueue(command_queue);
     ret = clReleaseContext(context);

     free(source_str);

     return 0;
}
```

Código ejemplo OpenCL: Kernel

Nótese la similitud del código de kernel desarrollado con respecto al equivalente CUDA (ver sección anterior):

```
1 __kernel void hello(__global char* buffer){
2     buffer[0] = 'H';
3     buffer[1] = 'e';
4     buffer[2] = 'l';
5     buffer[3] = 'l';
6     buffer[4] = 'o';
7     buffer[5] = ',';
8     buffer[6] = ' ';
9     buffer[7] = 'W';
10    buffer[8] = 'o';
11    buffer[9] = 'r';
12    buffer[10] = 'l';
13    buffer[11] = 'd';
14    buffer[12] = '!';
15    buffer[13] = '\0';
16 }
```

OmpSs

Como se ha detallado en las anteriores secciones, tanto CUDA como OpenCL permiten programar, de forma relativamente sencilla, dispositivos aceleradores compatibles. Aunque su introducción como modelos de programación ha facilitado y popularizado en gran medida el uso de este tipo de hardware, todavía presentan problemas graves relacionados con la facilidad de programación:

1. Ambos modelos de programación requieren una intervención explícita por parte del programador a la hora de gestionar tanto la reserva y liberación de memoria, como la transferencia de datos entre espacios de memoria.
2. OpenCL requiere el uso de APIs complejas para la configuración previa a la ejecución, típicamente ocupando decenas de líneas. Esto suele conllevar errores de programación en muchos programas, independientemente de su sencillez.
3. El uso de varios aceleradores simultáneamente es complejo utilizando tanto OpenCL como CUDA.
4. La gestión eficiente de las transferencias de datos entre espacios de memoria, intentando reducir aquellas transferencias innecesarias resulta responsabilidad del programador, y por tanto suele ser específica para un problema en concreto, y típicamente subóptima.

En respuesta a estas limitaciones han surgido nuevos modelos de programación, de entre los que destaca OmpSs. OmpSs [4] es un modelo de programación basado en la extracción y explotación de paralelismo a nivel de tareas desarrollado por el Barcelona Supercomputing Center (BSC). El

objetivo principal de OmpSs es facilitar la programación paralela de aplicaciones, delegando la ejecución paralela a un componente software (comúnmente denominado runtime o planificador de tareas) que, de forma automática, analiza las tareas anotadas por el usuario a través de #pragmas en el código, así como sus dependencias de datos, y las ejecuta de forma concurrente en los distintos procesadores disponibles sin ningún tipo de intervención por parte del usuario.

Las ventajas de este tipo de paradigma son múltiples. En primer lugar, la labor del desarrollador se limita a identificar qué partes del código (por ejemplo, funciones) son candidatas a ser consideradas tareas, así como a indicar qué datos de entrada recibe la tarea y qué datos genera. A partir de es punto, el planificador de tareas decide, en tiempo de ejecución, cuándo ejecutar cada tarea (gestionando de forma automática las dependencias de datos) y sobre qué plataforma ejecutarla.

En sistemas heterogéneos, además, OmpSs se encarga, de forma transparente, de gestionar las transferencias de datos entre espacios de memoria siempre que sea necesario. En general, pues, un código secuencial puede ser traducido a OmpSs sin un gran esfuerzo por parte del programador. Además, mediante los parámetros específicos del planificador, es posible modificar las políticas de uso de procesador, algoritmos de planificación o utilización concurrente de distintos tipos de arquitecturas, todo ello sin modificar el código. Conseguir una funcionalidad similar realizando una programación de menor nivel (por ejemplo, exclusivamente basada en CUDA) requeriría un esfuerzo mucho mayor.

Modelo de programación

Como se ha descrito, el modelo de programación de OmpSs se basa en añadir pequeñas anotaciones en forma de #pragmas al código secuencial, de modo que se informa al planificador de tareas de la existencia de una tarea. Una tarea es la unidad mínima de planificación sobre cada tipo de procesador disponible en el sistema (por ejemplo, un núcleo o una GPU). De hecho, ya que dichos pragmas son ignorados por el compilador sin soporte para OmpSs, cualquier código anotado con pragmas puede funcionar de forma secuencial sin ninguna modificación, y viceversa.

OmpSs se basa en dos componentes fundamentales:

1. Compilador (Mercurium). Se trata de un compilador específico fuente-a-fuente (es decir, transforma el código del usuario en un código con similares características, pero ampliado para dar soporte al modelo de programación). Básicamente, inserta invocaciones a rutinas implementadas en el planificador (por ejemplo, para añadir nuevas tareas a la cola de tareas, realizar sincronizaciones, etc.)
2. Planificador (Nanox). Se trata de un software sofisticado que, enlazado con nuestro programa y a través de las invocaciones a su API introducidas por Mercurium, es capaz de planificar de forma dinámica y en tiempo de ejecución las tareas anotadas por el usuario.

A continuación se muestran los pragmas principales soportados por OmpSs, así como una breve descripción del funcionamiento del planificador de tareas.

Pragmas básicos

OmpSs explota el paralelismo de forma asíncrona, construyendo, en tiempo de ejecución, un grafo de tareas en el que cada nodo representa una tarea específica, y las aristas representan dependencias de datos entre ellas. Sólo cuando dichas dependencias son satisfechas, una tarea puede pasar a ejecución. Por tanto, es responsabilidad del programador informar de la existencia de dichas tareas, así como de las dependencias entre ellas. Para esto, se utilizan #pragmas o anotaciones asociadas a partes concretas del código (típicamente funciones). Estas anotaciones se derivan directamente del lenguaje OpenMP, en el que OmpSs se basa.

Anotación de tareas (*#pragma omp task*)

Dada una determinada función, es posible anotar dicha función como una tarea utilizando la anotación #pragma omp task. Dicha anotación puede añadirse previa a una definición, declaración o invocación de una determinada función. Imaginemos el siguiente programa:

```
void inicializa( int * x, int n )
{
  // Inicializa cada elemento de X a un valor determinado.
}
void incrementa( int * x, int n )
{
  // Incrementa cada elemento de X en una unidad.
}

int main( void )
{
  int x[10];
  int y[10];

  inicializa( x, 10 );
  incrementa( x, 10 );

  inicializa( y, 10 );
  incrementa( y, 10 );
}
```

Lógicamente, cada una de las invocaciones a función se ejecuta secuencialmente, una tras otra. Sin embargo, aquellas que afectan al vector x son independientes de las que afectan al vector y (no existe relación de dependencia entre ellas) y por tanto podrían ser ejecutadas en paralelo. No así con cada una de las funciones que trabajan con un mismo vector, entre las que sí existe una relación de dependencia.

En primer lugar, el programador es capaz de identificar dichas invocaciones como tareas modificando mínimamente el código:

```
#pragma omp task
void inicializa( int * x, int n )
{
  // Inicializa cada elemento de X a un valor determinado.
}

#pragma omp task
void incrementa( int * x, int n )
{
  // Incrementa cada elemento de X en una unidad.
```

```
}

int main( void )
{
  int x[10];
  int y[10];

  inicializa( x, 10 );  // Tarea A.
  incrementa( x, 10 );  // Tarea B.

  inicializa( y, 10 );  // Tarea C.
  incrementa( y, 10 );  // Tarea D.
}
```

Esta pequeña modificación hará que el planificador sea consciente de la existencia de dos tipos de tareas (inicializa e incrementa). Sin embargo, todavía no se ha añadido ninguna información sobre las dependencias entre ellas.

Gestión de dependencias (input, output, inout)

Para gestionar las dependencias, se utilizan tres cláusulas distintas en OmpSs para la anotación #pragma omp task:

- input: identifica un dato de entrada a la tarea.
- output: identifica un dato de salida generado por la tarea.
- inout: identifica un dato de entrada/salida para la tarea.

Así, el código anterior añadiendo gestión de datos resultaría:

```
#pragma omp task output( x[0:n-1] )
void inicializa( int * x, int n )
{
  // Inicializa cada elemento de X a un valor determinado.
}

#pragma omp task inout( x[0:n-1] )
void incrementa( int * x, int n )
{
  // Incrementa cada elemento de X en una unidad.
}

int main( void )
{
  int x[10];
  int y[10];

  inicializa( x, 10 );  // Tarea A.
  incrementa( x, 10 );  // Tarea B.

  inicializa( y, 10 );  // Tarea C.
  incrementa( y, 10 );  // Tarea D.
}
```

Nótese que el tamaño de los datos de entrada o salida se especifica utilizando la notación [inicio:tamaño]. Con esta información, el planificador de tareas detectaría una dependencia de datos entre las tareas A y B, y otra entre C y D. Sin embargo, el conjunto A-B y C-D son independientes, y podrían ejecutarse en paralelo si se dispone de dos procesadores libres.

Sincronización.

Es necesario destacar que, en el ejemplo anterior, las tareas no se ejecutan necesariamente en el orden en el que aparecen en el código, sino en el que el planificador considera óptimo; por tanto, se habla de ejecución fuera de orden dictada por las dependencias de datos (data-flow parallelism). Esto significa que las tareas no se ejecutan instantáneamente, sino que su invocación es traducida por el compilador en el código necesario para introducirse en un grafo de dependencias de tareas.

Para introducir un punto de sincronización se utiliza el pragma omp taskwait. Este pragma detiene el flujo de ejecución hasta que todas las tareas anotadas previamente han sido efectivamente ejecutadas. Esta funcionalidad es necesaria, por ejemplo, para temporizar códigos, como se ve en el siguiente fragmento de código:

```
#pragma omp task output( x[0:n-1] )
void inicializa( int * x, int n )
{
  // Inicializa cada elemento de X a un valor determinado.
}

#pragma omp task inout( x[0:n-1] )
void incrementa( int * x, int n )
{
  // Incrementa cada elemento de X en una unidad.
}

int main( void )
{
  int x[10];
  int y[10];

  double t1 = toma_tiempo();

  inicializa( x, 10 );  // Tarea A.
  incrementa( x, 10 );  // Tarea B.

  inicializa( y, 10 );  // Tarea C.
  incrementa( y, 10 );  // Tarea D.

  #pragma omp taskwait

  double t2 = toma_tiempo();

}
```

Sin la introducción de dicho punto de sincronización, el tiempo medido sería simplemente el

necesario para introducir las tareas en el grafo de dependencias, no el tiempo transcurrido para su ejecución.

Gestión de dispositivos en sistemas heterogéneos.

Por último, otra de las ventajas de OmpSs de las que se hará uso en el presente trabajo es la gestión de ejecuciones sobre arquitecturas heterogéneas. Para etiquetar una tarea como ejecutable sobre un determinad tipo de plataforma aceleradora (por ejemplo, CUDA u OpenCL), se utiliza la etiqueta #pragma omp target device(tipo_dispositivo), siendo tipo_dispositivo uno de entre cuda, opencl o smp.

Centrándonos en arquitecturas aceleradoras CUDA, por ejemplo, el anterior ejemplo podría ser acelerado fácilmente utilizando un kernel CUDA con simples modificaciones (para el uso de kernels OpenCL, bastaría sustituir la cláusula device(cuda) por device(opencl)):

```
#pragma omp target device(cuda) ndrange(1,n,1)
#pragma omp task output( x[0:n-1] )
__global__ void inicializa( int * x, int n )
{
  // Kernel CUDA para incialización.
}

#pragma omp target device(cuda) ndrange(1,n,1)
#pragma omp task inout( x[0:n-1] )
__global__ void incrementa( int * x, int n )
{
  // Kernel CUDA para implemento.
}

int main( void )
{
  int x[10];
  int y[10];

  double t1 = toma_tiempo();

  inicializa( x, 10 );  // Tarea A.
  incrementa( x, 10 );  // Tarea B.

  inicializa( y, 10 );  // Tarea C.
  incrementa( y, 10 );  // Tarea D.

  #pragma omp taskwait

  double t2 = toma_tiempo();

}
```

Lógicamente, será necesario proporcionar códigos CUDA (u OpenCL) para las implementaciones de las tareas `inicializa` e `incrementa`. Sin embargo, es necesario destacar la principal ventaja de OmpSs aquí: no es necesario realizar reservas de memoria, ni transferencias de datos, ni liberaciones de memoria. Además, en el caso de disponer de más de una GPU compatible con CUDA, por ejemplo, sería posible ejecutar los kernels de forma paralela en ambas, sin absolutamente ninguna intervención por parte del usuario. La cláusula `ndrange` simplemente

controla el número de hilos (o work-items) total lanzados en el acelerador.

Pragmas avanzados. Implements.

En los anteriores códigos, únicamente se han mostrado tareas que son ejecutadas exclusivamente en CPU, o bien en el acelerador correspondiente (compatible con CUDA u OpenCL). Sin embargo, sería deseable ejecutar tareas concurrentemente en todas las unidades disponibles, sean o no aceleradores.

Para ello, OmpSs implementa una funcionalidad llamada versioning; la idea es proporcionar al planificador dos o más implementaciones distintas para cada tarea, y otorgarle libertad para usar una u otra en función de los recursos disponibles en un momento determinado de la ejecución. Para ello, bastaría con utilizar la cláusula implements ofrecida por OmpSs. Sobre el anterior código:

```
#pragma omp target device(smp)
#pragma omp task output( x[0:n-1] )
__global__ void inicializa( int * x, int n )
{
  // Kernel CUDA para incialización.
}

#pragma omp target device(smp)
#pragma omp task inout( x[0:n-1] )
__global__ void incrementa( int * x, int n )
{
  // Kernel CUDA para implemento.
}

#pragma omp target device(cuda) ndrange(1,n,1) implements( inicializa )
#pragma omp task output( x[0:n-1] )
__global__ void inicializa_gpu( int * x, int n )
{
  // Kernel CUDA para incialización.
}

#pragma omp target device(cuda) ndrange(1,n,1) implements( inicializa )
#pragma omp task inout( x[0:n-1] )
__global__ void incrementa_gpu( int * x, int n )
{
  // Kernel CUDA para implemento.
}

int main( void )
{
  int x[10];
  int y[10];

  double t1 = toma_tiempo();

  inicializa( x, 10 );  // Tarea A.
  incrementa( x, 10 );  // Tarea B.

  inicializa( y, 10 );  // Tarea C.
  incrementa( y, 10 );  // Tarea D.

  #pragma omp taskwait

  double t2 = toma_tiempo();
```

```
}
```

En el anterior ejemplo, se han proporcionado dos implementaciones distintas de las tareas inicializa e incrementa. Una de ellas tiene como objetivo smp (es decir, núcleos CPU), mientras que la otra, con sufijo _gpu, tiene como objetivo un dispositivo CUDA. Así, es el planificador quién, en función de parámetros como los dispositivos libres, o un historial de rendimiento de cada tarea sobre los dos tipos de arquitectura, elegirá, sin intervención del programador, una u otra implementación de la tarea.

Planificador de tareas (runtime)

Dados los códigos de ejemplo anteriormente descritos, y una vez compilados a través del compilador Mercurium, la ejecución del código deja de ser meramente secuencial. El planificador de tareas Nanox crea tantos hilos de ejecución (worker threads) como se especifiquen a través de variables de entorno. La labor de estos hilos de ejecución es simplemente consultar la existencia de tareas listas para ejecución (es decir, con todas sus dependencias de datos satisfechas), realizar las transferencias de datos necesarias, y ejecutar la tarea correspondiente dentro de cada plataforma.

Tanto el número de worker threads como los distintos parámetros del planificador pueden modificarse a través de variables de entorno. Más concretamente, la variable de entorno NX_ARGS contiene todos los parámetros que modifican el comportamiento del planificador. Por ejemplo:

```
NX_ARGS="--gpus=2 --smp-threads=14" ./programa
```

ejecutaría el programa utilizando dos GPUs y 14 cores SMP.

Ventajas e inconvenientes de cada modelo de programación

El uso de un planificador de tareas conlleva unas ventajas importantes, entre ellas:

- Eliminación de la gestión de dependencias/paralelismo a nivel de tareas, que es gestionada automáticamente por el planificador.
- Eliminación de la gestión de memoria: usando un runtime, no es necesario realizar una gestión explícita de memoria (reserva, transferencias, liberación).
- Uso sencillo de varios aceleradores simultáneamente: no es necesario realizar una gestión explícita de los lanzamientos sobre cada acelerador presente en el sistema. Estos son automáticamente detectados y gestionados por el runtime (en función de los parámetros introducidos por el usuario).

Por contra, la principal desventaja es la necesidad de adaptar los algoritmos al paradigma de paralelismo a nivel de tareas (cosa no siempre posible), y el menor control sobre la ejecución, ya que ésta es gestionada automáticamente por el planificador, con mínima intervención para el usuario.

Ejemplos de códigos OmpSs

Ompss + CUDA

```
#include <stdio.h>
#include "omp.h"

#define MEM_SIZE 128

#pragma omp target device(cuda) copy_deps ndrange(1,1,1,1)
#pragma omp task inout(buffer[0:MEM_SIZE])
__global__ void hello(char* buffer);

int main(){
    char buffer[MEM_SIZE];

    buf = (char *) calloc(MEM_SIZE,sizeof(char));

    hello(buffer);

    printf("%s\n", buffer);

    return EXIT_SUCCESS;
}
```

Ompss + OpenCl

```
#include <stdio.h>
#include "omp.h"

#define MEM_SIZE 128

#pragma omp target device(cuda) copy_deps ndrange(1,1,1,1) file(kernel.cl)
#pragma omp task inout(buffer[0:MEM_SIZE])
__kernel void hello(char* buffer);

int main(){
    char buffer[MEM_SIZE];

    buf = (char *) calloc(MEM_SIZE,sizeof(char));

    hello(buffer);

    printf("%s\n", buffer);

    return EXIT_SUCCESS;
}

```

3 Problema objetivo. Detección de bordes

En este capítulo se explica el algoritmo de Canny (etapa por etapa) y la motivación de realizarlo.

Descripción del algoritmo y motivación

El algoritmo usado para nuestras pruebas es Canny. El programa ha sido creado totalmente desde 0, en código C, y usando las herramientas atrás descritas.

El algoritmo de Canny fue desarrollado por John F.Canny en 1986, donde se utiliza varias etapas para detectar la mayoría de bordes en una imagen dada. El propósito de este algoritmo es el de descubrir bordes en las imágenes que se llegaran a analizar con este algoritmo. Con la técnica de reducción significativa de datos en una imagen, preservando las propiedades estructurales de la imagen.

Este algoritmo esta enfocado en los siguientes puntos:

- *Buena detección:* El algoritmo debe marcar el mayor número real en los bordes de la imagen como sea posible.
- *Buena localización:* Los bordes de marca deben estar lo más cerca posible del borde de la imagen real.
- *Respuesta mínima:* El borde de una imagen sólo debe ser marcado una vez, y siempre que sea posible, el ruido de la imagen no debe crear falsos bordes

Desde el punto de vista del modelo de progrmación OmpSs, este algoritmo resulta interesante, puesto que:

1. Presenta distintos tipos de tareas asociadas a cada etapa del algoritmo.
2. Presenta dependencias de datos entre tareas no triviales.
3. Las implementaciones de cada tarea son (relativamente) sencillas de implementar.
4. Cada tarea es altamente paralela a nivel de datos, por lo que permite una correcta explotación de los aceleradores hardware.

Etapas

- Suavizar: Desenfoque de la imagen para eliminar el ruido.
- Encontrar gradientes: los bordes deben estar marcados en los gradientes de la imagen que tiene magnitudes grandes.

- No supresión máxima: Sólo los máximos locales se debe marcar como bordes.
- Umbralización doble: Los posibles bordes deben estar determinados por umbralización.
- Seguimiento por histéresis: Los bordes finales se determinan mediante la supresión de todas las aristas que no están conectados a una muy determinada borde (fuerte).

Procesamiento por bloques. Descripción de las tareas.

Para fomentar el paralelismo, la imagen es divida en bloques totalmente independientes entre si en una misma etapa, de tal forma que dos trozos de imagen distintas se puedan realizar simultáneamente.

El tamaño de bloque es dinámico, es decir, el programa es capaz de descomponer la imagen en cualquier tamaño de bloque dado por el usuario. Cada bloque de la imagen es procesado por cada función, siendo a su vez una tarea.

Algunas etapas como por ejemplo la primera, dado un pixel, utiliza los de su alrededor y lo multiplica por una matriz dada para obtener el valor resultado de ese pixel. Este paso puede dar error en los pixeles cercanos a los limites de la imagenes, para ello hemos agrandado la imagen rellenado de ceros todos los limites de la imagen. Es decir, si la imagen es de 15x15, la hemos agrandado a 16x16 con ceros.

Etapas

Se muestran a continuación las etapas o fases principales que componen el procesamiento de la imagen. En nuestro caso, existe una primera fase de preprocesado en la que la imagen a color es transformada en una imagen en escala de grises, cuyos detalles de implementación se obvian en la siguiente descripción:

La siguiente fase consiste en limpiar la imagen de ruidos. Es inevitable que todas las imágenes tomadas desde una cámara contengan cierta cantidad de ruido; el objetivo de esta etapa es evitar que el ruido introducido se confunda con bordes. Para ello se le aplica un filtro de Gauss (en nuestro caso, un filtro de Gauss con una desviación estándar de σ = 1,4). El filtro de Gauss variar en tamaño; en nuestro caso de estudio se ha elegido un filtro de tamaño 5x5:

$$\mathbf{B} = \frac{1}{159}\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix} * \mathbf{A}$$

La siguiente figura muestra un ejemplo de aplicación del anterior filtro sobre una imagen de entrada en escala de grises:

A continuación, el algoritmo de Canny encuentra básicamente bordes, considerando éstos como aquellas zonas de la imagen con variación más rápida de la intensidad. Estas áreas se encuentran mediante la determinación de los gradientes de la imagen. Los gradientes en cada píxel en la imagen suavizada se determinan mediante la aplicación de lo que se conoce como filtro de Sobel: el primer paso es aproximar el gradiente en las direcciones x e y respectivamente, aplicando el operador de Sobel:

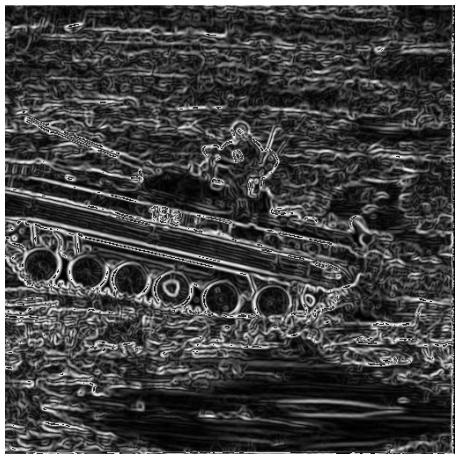
$$K_{GX} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

$$K_{GY} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Las magnitudes del gradiente (también conocidos como los puntos fuertes de borde) se pueden determinar calculando la arcotangente entre los puntos resultados de x e y:

$$\theta = \arctan\left(\frac{|G_y|}{|G_x|}\right)$$

La siguiente imagen muestra el resultado de la aplicación del filtro de Sobel sobre la imagen de entrada resultante de la fase anterior:



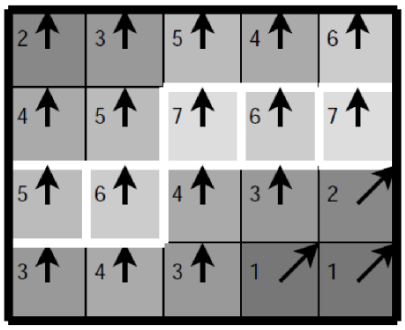
Por último, se aplica una fase llamada de de no supresión máxima. El propósito de este paso es convertir los "enmascarados" bordes en la imagen de las magnitudes del gradiente a los "fuertes" bordes.

Básicamente esto se hace mediante la preservación de todos los máximos locales en la imagen de gradiente, y la eliminacion del resto de información. El algoritmo para cada píxel de la imagen de gradiente es el siguiente:

1. Alrededor de la dirección del gradiente theta más cercano a 45º.
2. Comparar la resistencia del borde del píxel actual con la resistencia de los bordes en dirección al píxel en el gradiente positivo y negativo. Es decir, si la dirección del gradiente

es el norte (theta= 90º), por ejemplo, comparar con los píxeles hacia el norte y el sur.

3. Si la resistencia del borde del píxel actual es el más grande; preservar el valor de la resistencia de los bordes. Si no es así, suprimir (es decir, eliminar) el valor.



La siguiente figura muestra un ejemplo de aplicación de dicha fase sobre el resultado de la aplicación del operador Sobel:

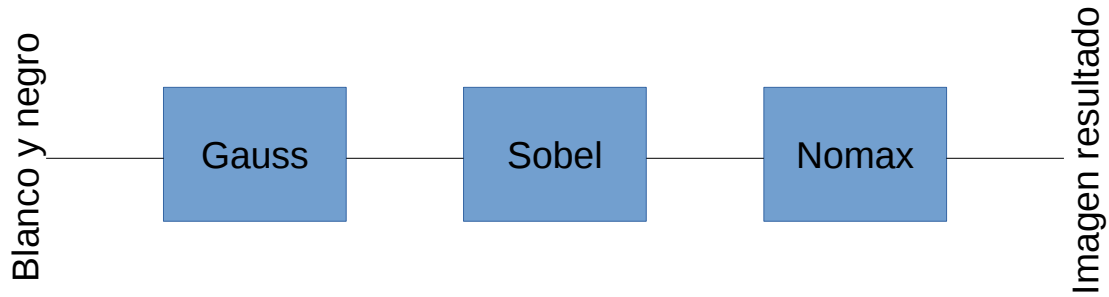
Implementación utilizando OmpSs

En este capítulo se introduce de forma de más precisa el desarrollo del programa desarrollado y su adaptación a un paradigma de paralelismo a nivel de tareas, explicando en detalle el tratamiento de la imagen desde su origen hasta su destino.

Visión general de la implementación

El programa parte de una imagen en colores, y tiene como objetivo detectar los bordes de la misma. Como se ha detallado anteriormente, la imagen es previamente transformada a escala de grises, para a continuación aplicar las distintas fases del algoritmo de Canny de forma secuencial. En nuestro caso y como se explicó anteriormente, las dos últimas fases se omitirán en la descripción, al no ser estrictamente necesarias (a partir de la tercera fase, no_máximos, la detección de bordes ha sido realizada, siendo las restantes fases de mejora de calidad de los bordes).

Para dar una visión general del funcionamiento de la implementación, imaginemos que cada fase es una caja negra, la cual esta conectada a la siguiente fase, y que cada etapa no puede continuar si la etapa anterior no ha terminado.



Cada etapa recibe un buffer de entrada y devuelve un buffer de salida, que contiene el resultado de la aplicación del tratamiento correspondiente. Más concretamente:

- Primera fase (*filtro gaussiano*):
  - Recibe como buffer de entrada, la imagen en escala de blanco y negro.
  - La imagen es tratada y devuelta en un buffer de salida, el cual llamaremos buffer_gaussiano.
- Segunda fase (*filtro Sobel*):
  - Recibe como buffer de entrada, la imagen tratada por la fase gaussiana (buffer_gaussiano).
  - La imagen es tratada y devuelta en un buffer de salida, el cual llamaremos buffer_sobel.
- Tercera fase (*supresión de máximos*):
  - Recibe como buffer de entrada, la imagen tratada por la fase sobel (buffer_sobel).

  - La imagen es tratada y devuelta en un buffer de salida, el cual llamaremos buffer_no_max.

- Por último, se escribe la imagen resultante a disco.

Paralelismo a nivel de tareas y dependencias de datos

Como se ha visto, cada fase requiere datos de la fase anterior. Más concretamente, en una implementación no orientada a bloques, cada fase requiere que la etapa anterior haya finalizado completamente (sobre toda la imagen) para proceder. Este proceso ralentiza el procesado de la imagen en sistemas con múltiples procesadores, al ser necesario esperar a que la imagen sea tratada completamente por la fase anterior. Para optimizar el trabajo, se ha optado por un procesamiento orientado a bloques: la imagen se divide en bloques de filas de tamaño configurable, que se identificarán como tareas a través de los mecanismos proporcionados por OmpSs y se asignarán, en tiempo de ejecución, a las distintas unidades de proceso existentes en el sistema.

Este grado de paralelismo permite ejecutar, de forma concurrente, varias tareas asociadas a una misma fase, e incluso pertenecientes a fases distintas, siempre que las dependencias de datos hayan sido satisfechas. Además, este esquema permite ejecutar cada tarea en distintos núcleos del procesador y en los aceleradores disponibles. Por ejemplo, si disponemos de seis tareas listas para ser ejecutadas, podría pontencialmente lanzarse cuatro de ellas a núcleos de CPU, y dos restantes a GPUs disponibles (o a cualquier otro tipo de acelerador).

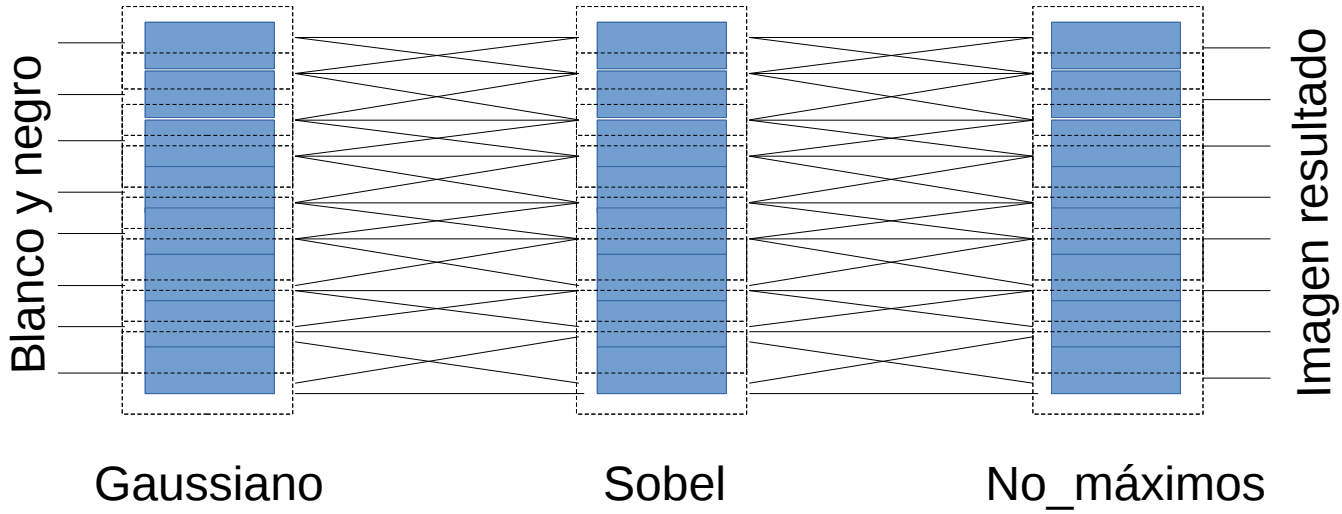
Toda tarea que es lanzada a un núcleo de CPU procesa un bloque de la imagen de manera secuencial, pixel a pixel; en cambio las GPUs (u otros aceleradore) poseen múltiples núcleos, siendo aprovechables para aumentar el nivel de paralelismo. Para aprovechar este hecho, cada tarea que es lanzada a la GPU, es procesada en paralelo, es decir, cada pixel del bloque de la imagen es procesado por un núcleo de la GPU, aprovechando ya no solo el paralelismo a nivel de tareas (mediante OmpSs), sino también de datos (mediante CUDA u OpenCL). Desde este punto de vista, los aceleradores son vistos por OmpSs como “cajas negras”, o unidades mínimas de asignación de tareas; es el código interno de cada tarea quien extraerá paralelismo a nivel de datos de forma interna.

Algunas de las fases descritas requieren, para calcular el valor de un pixel, los valores de los pixeles cercanos a él calculados en la fase anterior. Esto puede suponer un problema al calcular los píxeles en los extremos de la imagen, ya que pueden ser necesarios valores de pixeles que estén fuera del rango de la imagen. La solución que se ha implementado crea un halo (también conocido como padding o relleno) alrededor de la imagen o borde correspondiente.

Al aplicar esta técnica surgen dependencias entre tareas mayores. Como se dijo anteriormente cada bloque depende de algunos bloques anteriores. Al introducir padding, cada bloque va a ser mayor, abarcando valores de otros bloques de su misma fase. Veamos un ejemplo: la aplicación del filtro de

Sobel sobre el primer bloque de la imagen requerirá no sólo los datos de entrada correspondientes a dicho bloque (y obtenidos tras la aplicación del filtro gaussiano sobre el primer bloque de la imagen), sino también ciertas filas adicionales computadas tras aplicar el filtro gaussiano al segundo bloque de la imagen. Por tanto, existe una dependencia de datos entre la tarea que aplicará el filtro Sobel al primer bloque de la imagen, y las dos primeras tareas que aplicarán el filtro gaussiano sobre los dos primeros bloques de la imagen.

A continuación se muestra un ejemplo de dependencias entre bloques, para una imagen de 1024x1024, en bloques de 128, es decir, 8 bloques por fase:



La gestión de este tipo de dependencias de datos es realmente compleja en el caso de ser realizada a mano, e ilustra las ventajas de utilizar un modelo de programación como OmpSs, como se verá a continuación.

Esquema algorítmico y detalles de implementación

En una implementación secuencial, el esquema algorítmico básico procesaría cada una de las tres fases de forma consecutiva: la finalización del procesado de una de ellas supondría el inicio de la siguiente. En un procesado por bloques, la aplicación de cada fase se realiza a nivel de bloque de filas, también de forma secuencial, como muestra el siguiente código.

```
// ...

int *orig;
int *dest;
int *dest_conv;
int *dest_gaus;
int *dest_sobel;
int *dest_no_max;
int *dest_dir;

// Reserva de memoria.
malloc( ... );

for (int xoffset = 0; xoffset < height; xoffset = xoffset + size_block)
{
  // Modificacion de punteros para apuntar a bloque correspondiente.

  gaussiano (orig, dest_gaus, ... );

}

for (int xoffset = 0; xoffset < height; xoffset = xoffset + size_block)
{
  // Modificacion de punteros para apuntar a bloque correspondiente.

  sobel (dest_gaus, dest_sobel, ...);

}

for (int xoffset = 0; xoffset < height; xoffset = xoffset + size_block)
{
  // Modificacion de punteros para apuntar a bloque correspondiente.

  no_max (dest_sobel,dest, ...);

}

//...
```

Cabe destacar dos observaciones principales:

1. En este caso, no existe ningún tipo de paralelismo a nivel de tareas; es decir, cada tarea se ejecuta de forma exclusivamente secuencial, sin solapar su procesamiento con ninguna otra.
2. Las invocaciones a cada función de procesamiento de la imagen se ejecutan exclusivamente sobre CPU, sin utilizar en ningún caso ninguno de los posibles aceleradores disponibles.

Una migración de este código para ser acelerado mediante uno o varios aceleradores, requeriría una reescritura completa del código. Sin embargo, mediante el uso de OmpSs, es posible realizar una transformación del mismo con mínimos cambios. De hecho, los cambios principales serían básicamente dos:

1. Desarrollo de kernels específicos (CUDA u OpenCL) para la implementación de cada fase en el acelerador.

2. Etiquetado de cada tarea mediante pragmas, indicando sus datos de entrada y salida, y la plataforma o plataformas en las que debe ejecutarse.

Este último caso es de especial interés para el desarrollo del trabajo propuesto. A continuación, se detalla el mecanismo de anotación para una de las tareas (gaussiano) utilizando #pragmas OmpSs. Nótese como, si dicho pragma es eliminado o no soportado por el compilador, el programa seguiría funcionando de forma correcta (aunque secuencial):

```
#pragma omp target device(smp) copy_deps
#pragma omp task in( ( [M][N] orig)[origXoffset;h_sizeConv*2+bsX][0;N], \
                     mascara[0;(sizeConv*sizeConv)],                    \
                     number_gauss[0;1])                                 \
                out( ([M][N] dest)[destXoffset;bsX][0;N])
void gaussiano (int *orig,
                int *dest,
                int origXoffset,
                int origYoffset,
                int destXoffset,
                int destYoffset,
                int M,
                int N,
                int m,
                int n,
                int bsX,
                int bxY,
                int sizeConv,
                int h_sizeConv,
                int *mascara,
                int *number_gauss);
```

Analicemos cada uno de las cláusulas que componen cada #pragma:

- Indicamos que la tarea será ejecutada en CPU (device(smp)).
- Indicamos que orig (el buffer original) es un dato de entrada cuya primera dimensión es de tamaño (hSizeConv * 2 + bsX), es decir, el tamaño de bloque seleccionado más un halo superior y otro inferior, y cuya segunda dimensión es N (la anchura de la imagen).
- De forma similar, dest (el buffer destino), es un dato de salida cuya primera dimensión es únicamente bsX, y cuya segunda dimensión es N.

Una cualidad de OmpSs es la existencia de la opción *implements*, la cual al ser incluida en un pragma, indica que una función determinada es una implementación alternativa para una tarea, o incluso tiene como objetivo una plataforma diverente (por ejemplo, GPU o DSP contra CPU). Dependiendo de la disponibilidad de cada plataforma en tiempo de ejecución, se seleccionará la implementación adecuada para su ejecución. Por lo tanto, esto permite lanzar tareas de forma transparente sobre un sistema heterogéneo, siendo el planificador de OmpSs el que aprovechará sus componenetes de forma automática. Una vez más, tomando como referencia la tarea gaussiano, tendríamos un tipo de tarea alternativa, llamada gaussiano_gpu (e internamente implementada en CUDA u OpenCL), que implementaría la tarea original gaussiano:

```
#pragma omp target device(cuda) copy_deps ndrange(2,bsX+h_sizeConv*2,N,16,16) implements(gaussiano)
#pragma omp task in( ( [M][N] orig)[origXoffset;h_sizeConv*2+bsX][0;N],    \
                      mascara[0;(sizeConv*sizeConv)],                      \
                      number_gauss[0;1])                                   \
                 out( ([M][N] dest)[destXoffset;bsX][0;N])
__global__ void gaussiano_gpu (int *orig,
                               int *dest,
                               int origXoffset,
                               int origYoffset,
                               int destXoffset,
                               int destYoffset,
                               int M,
                               int N,
                               int m,
                               int n,
                               int bsX,
                               int bxY,
                               int sizeConv,
                               int h_sizeConv,
                               int *mascara, int *number_gauss);
```

Este procedimiento ha sido desarrollado para todos los tipos de tareas, usando tanto CUDA (con el fin de explotar los procesadores gráficos de un sistema que lo soporte), como OpenCL (con el objetivo final de evaluar el rendimiento de un sistema basado en DSPs) para la implementación de cada tarea. La siguiente sección muestra los resultados experimentales obtenidos tras la evaluación de cada uno de dichos códigos.

4 Resultados experimentales

En este capítulo mostrare los resultados de testeo con distintos tamaños para la misma imagen.

El testeo consistirá en extraer tiempos (en microsegundos), rendimiento (MegaBits por segundo) y consumo energético (julios) para distintas combinaciones de tamaños de imagen y tamaños de bloque. Donde sea posible, se experimentará con distintas configuraciones de ejecución (número de hilos de ejecución y número de aceleradores a utilizar). Se realizará sobre distintos plataformas hardware, todas ellas basadas en procesadores de propósito general (CPUs) y uno o varios aceleradores hardware, incluyendo procesadores gráficos y DSPs. La gestión de imágenes se ha desarrollado utilizando las facilidades de la biblioteca Cimg. Todos los códigos han sido desarrollados utilizando C++ y las últimas versiones del compilador Mercurium y el runtime Nanox pertenecientes al proyecto OmpSs, excepto aquellos a ejecutar sobre el acelerador, para los que se ha utilizado CUDA (en el caso de GPUs) u OpenCL (en el caso de los DSPs).

Descripción de las arquitecturas objetivo

En este proyecto hemos trabajo con dos máquinas:

- Bujaruelo: Máquina que consta de dos procesadores y tres GPUs.
- K2H: Máquina que consta de un procesador y un DSP.

Descripción de las CPU Intel Xeon (Bujaruelo)

- Tipo: Intel® Xeon® Processor E5-2695 v3 (64 bits)
- Núcleos: 28, con Hyperthreading (hasta 56 cores lógicos).
- Velocidad por núcleo: 2.3Ghz – 3.3Ggz
- Bus usado: PCI Express
- Potencia pico (TDP): 120W

Descripción de las GPU (Bujaruelo)

Bujaruelo consta de hasta tres tarjetas gráficas, dos de ellas son NVIDIA GeForce GTX 980, siendo mas avanzadas que la restante, la NVIDIA GeForce GTX 950.

La tarjeta gráfica NVIDIA GeForce GTX 980 es la más avanzada del mundo al basarse en la nueva arquitectura NVIDA, teniendo una rapidez y consumo óptimos

- Tipo: NVIDIA GeForce GTX 980

- Núcleos: 2048
- Velocidad por núcleo: 1126Mhz – 1216Mhz
- Bus usado: PCI-E 3.0
- Potencia pico (TDP): 165W

La tarjeta gráfica NVIDIA GeForce GTX 950 es de menor potencia que la 980, pero siendo aún así unas de las mejores del mercado. Está basada en la arquitectura NVIDA Maxwell y proporciona un rendimiento tres veces mayor a las tarjetas de la generación anterior.

- Tipo: NVIDIA GeForce GTX 950
- Núcleos: 768
- Velocidad por núcleo: 1024Mhz – 1188Mhz
- Bus usado: PCI-E 3.0
- Potencia pico (TDP): 90W

Descripción de las CPU ARM (K2H)

El procesador de alta eficiencia energética ARM Cortex-A15 está diseñado para una amplia gama de aplicaciones, que necesitan un alto rendimiento con las ventajas de la arquitectura de bajo consumo de ARM.

- Tipo: Cortex-A15 (32 bits)
- Núcleos: 4
- Velocidad por núcleo: 1Ghz – 2.5Ghz
- Potencia pico (TDP): 10W

Descripción del DSP (K2H)

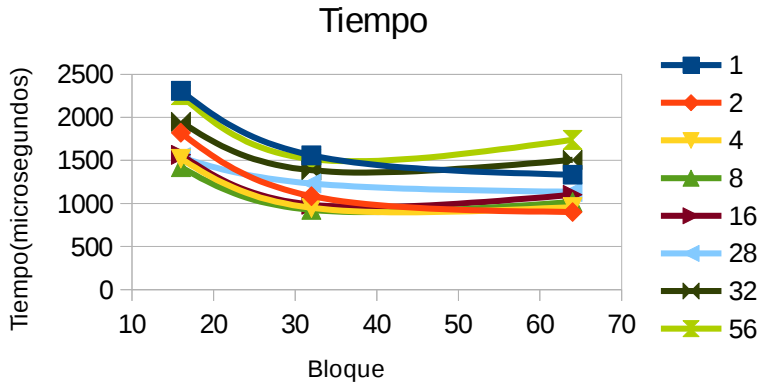
- Tipo: Familia C66x
- Núcleos: 8
- Velocidad por núcleo: 1Ghz – 1.25 Ghz
- Potencia pico (TDP): 10W

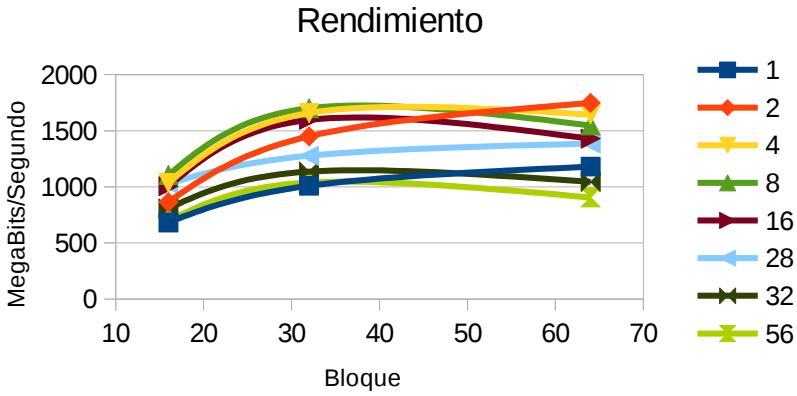
Debido a la falta de disponibilidad de entornos precisos de medición de consumo sobre K2H, se ha simplificado el cálculo del consumo energético suponiendo una potencia disipada constante coincidente con el TDP en todos los experimentos.

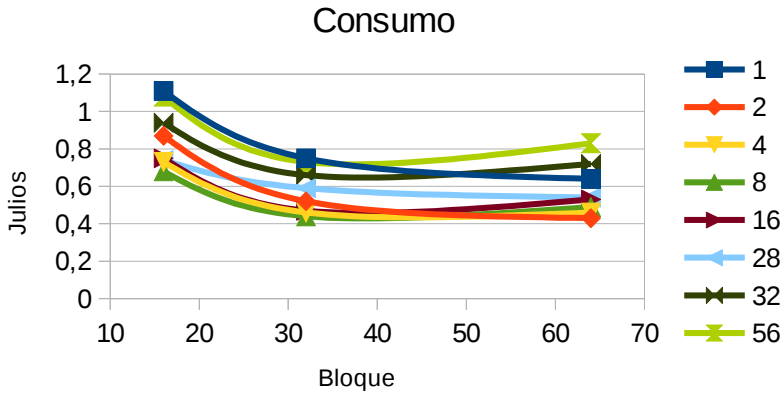
Resultados experimentales y análisis

Utilización exclusiva de CPU (Bujaruelo)

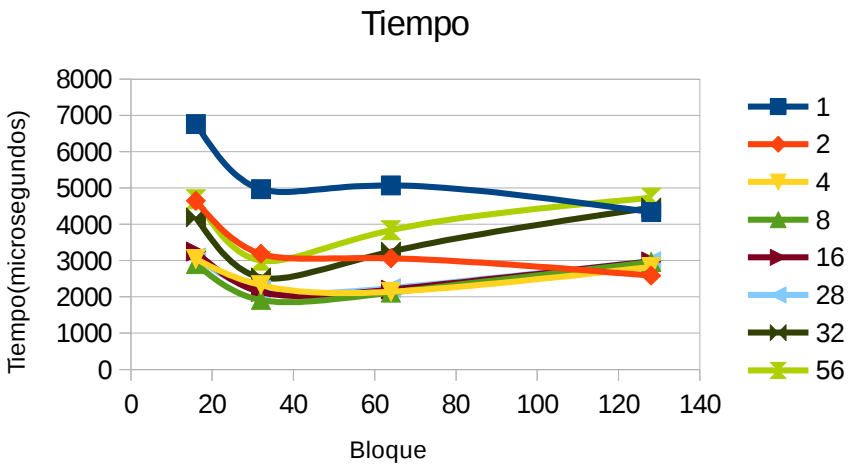
- Tamaño de imagen 128x128, utilizando entre 1 y 56 hilos de ejecución.

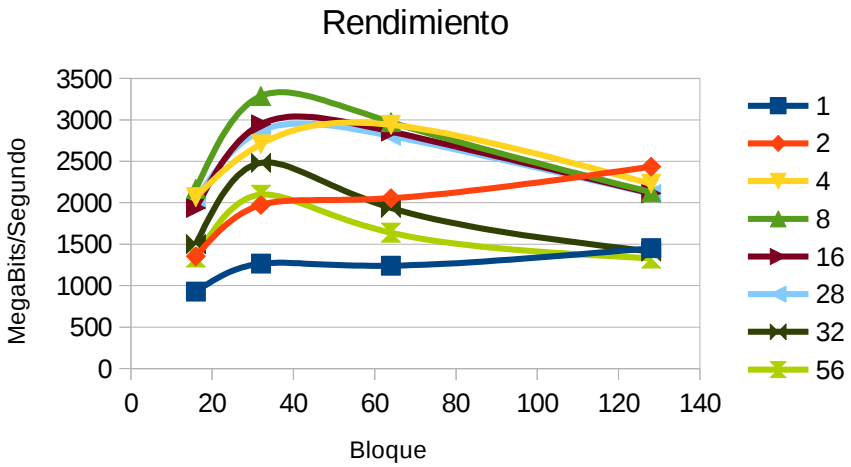


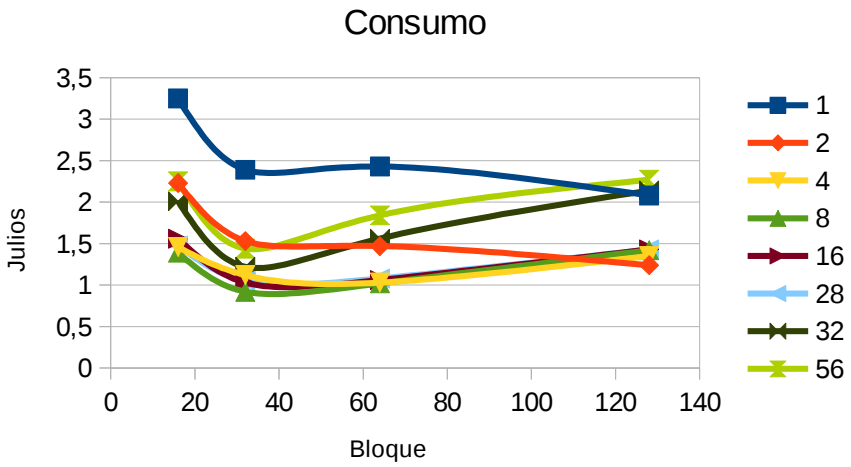




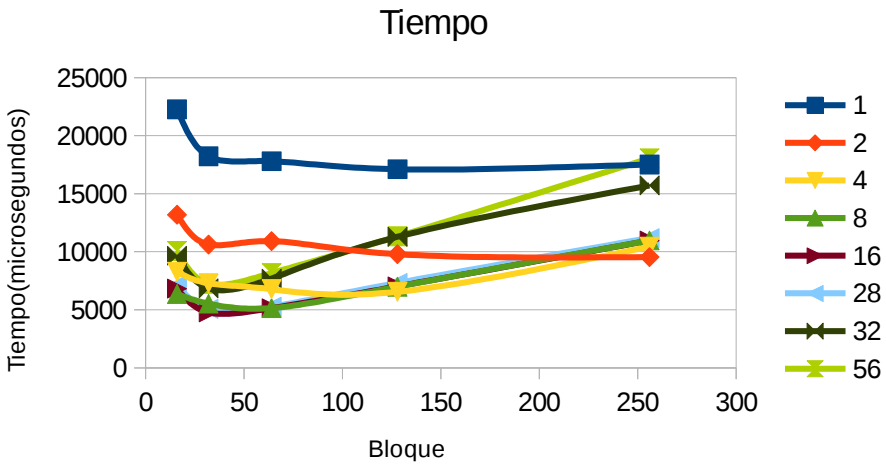
- Tamaño de imagen 256x256, utilizando entre 1 y 56 hilos de ejecución.

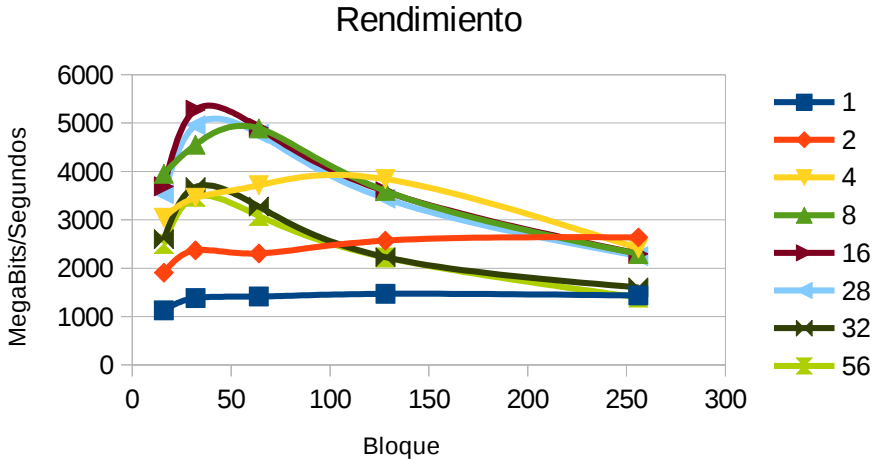


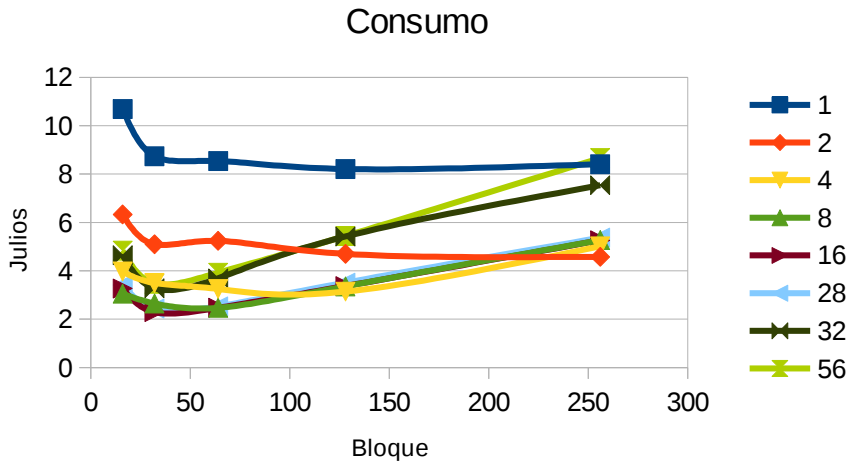




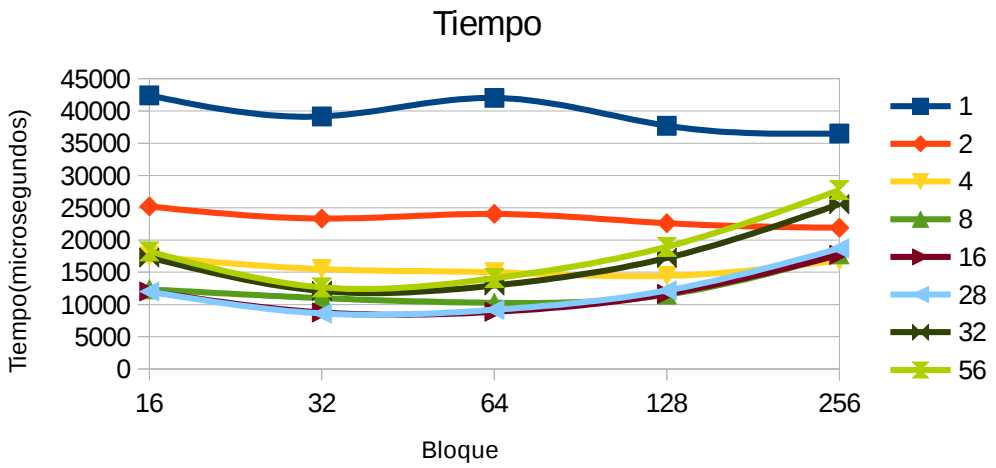
- Tamaño de imagen 512x512, utilizando entre 1 y 56 hilos de ejecución.

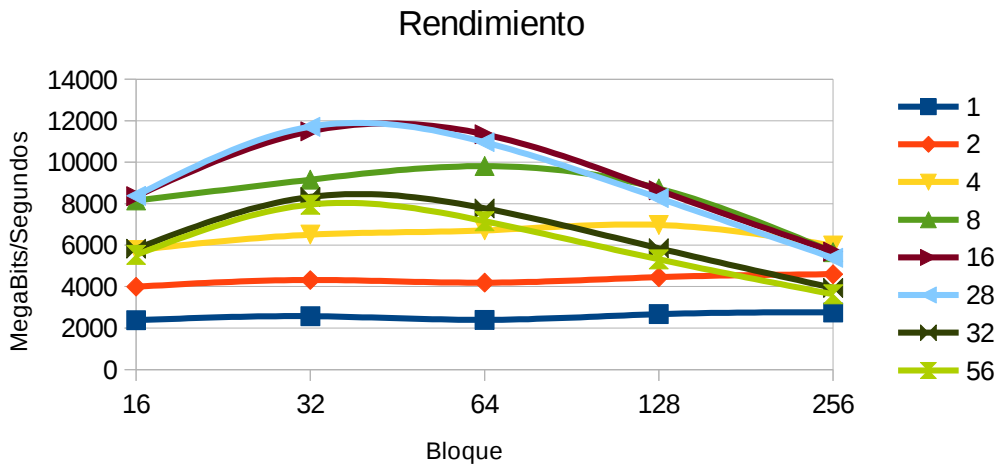


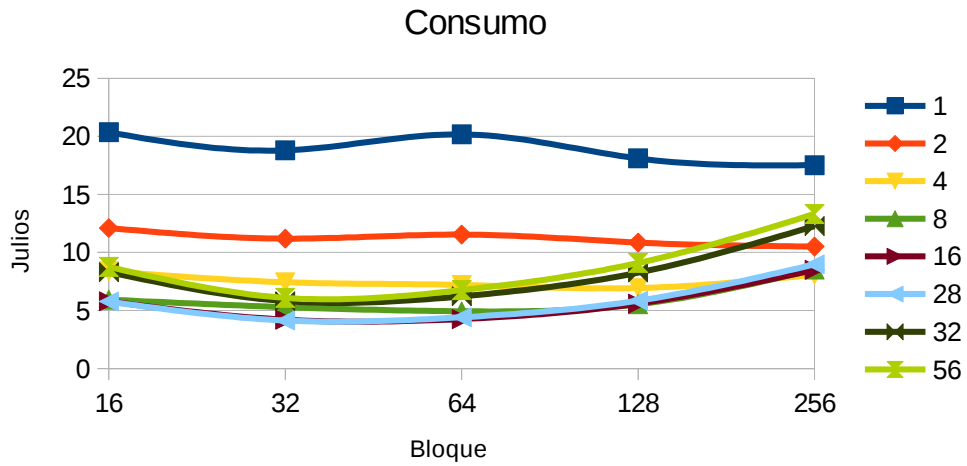




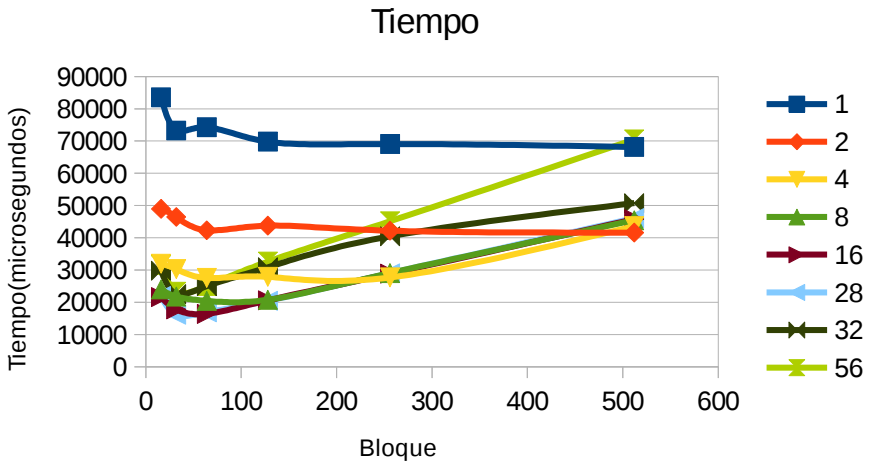
- Tamaño de imagen 768x768, utilizando entre 1 y 56 hilos de ejecución.

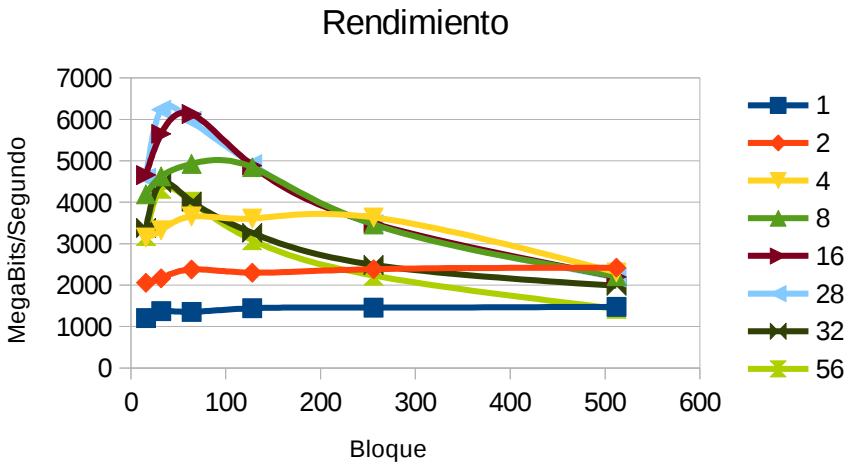


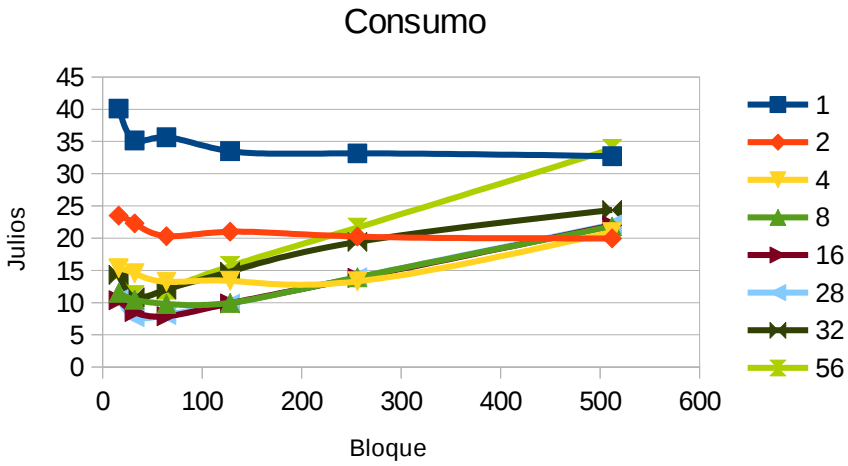




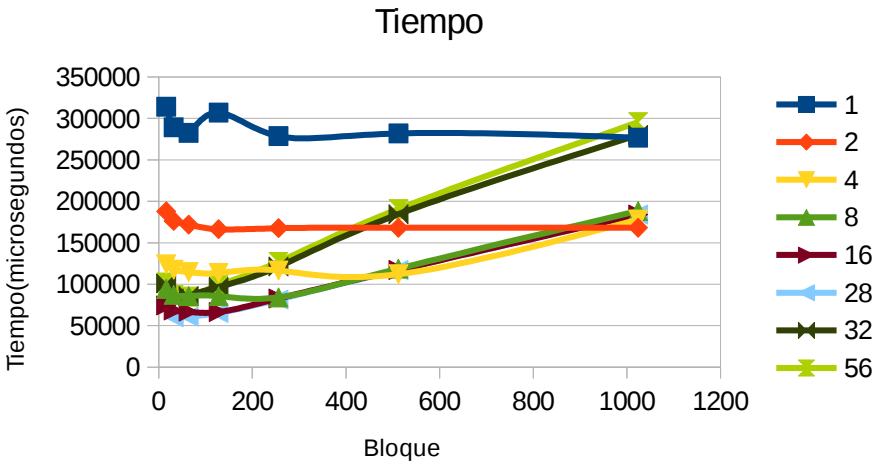
- Tamaño de imagen 1024x1024, utilizando entre 1 y 56 hilos de ejecución.

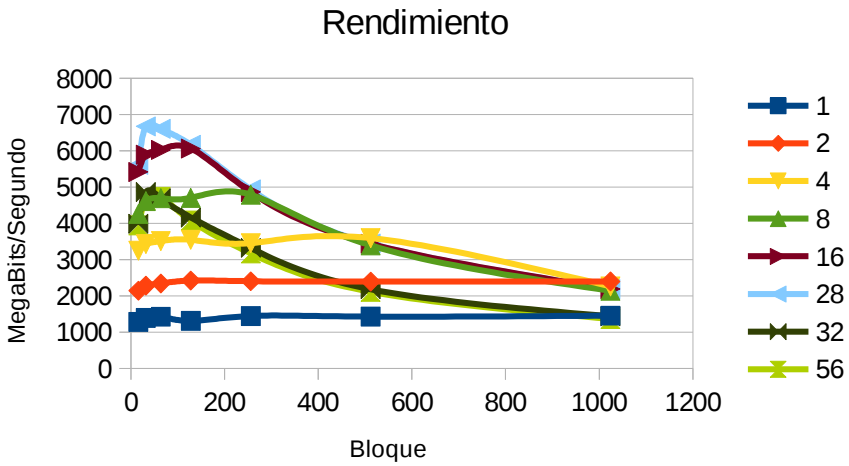


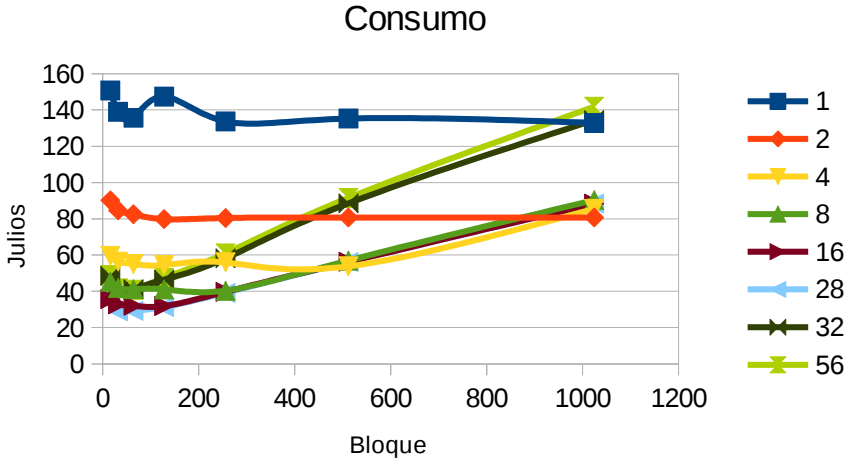




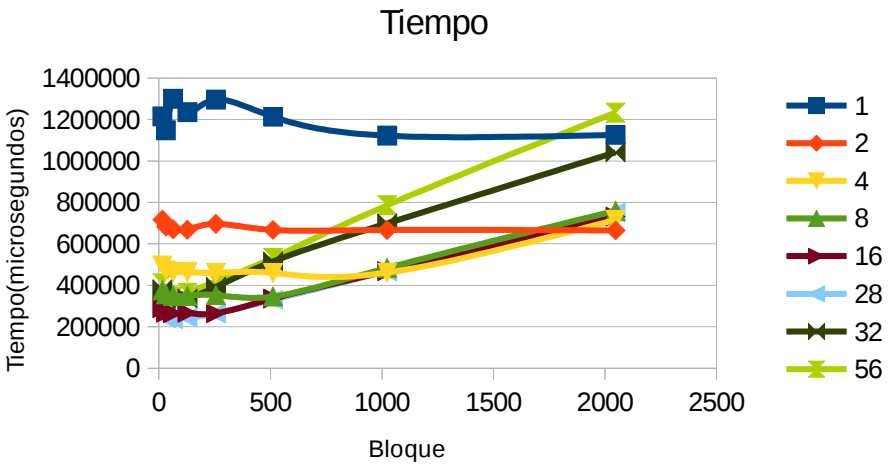
- Tamaño de imagen 2048x2048, utilizando entre 1 y 56 hilos de ejecución.

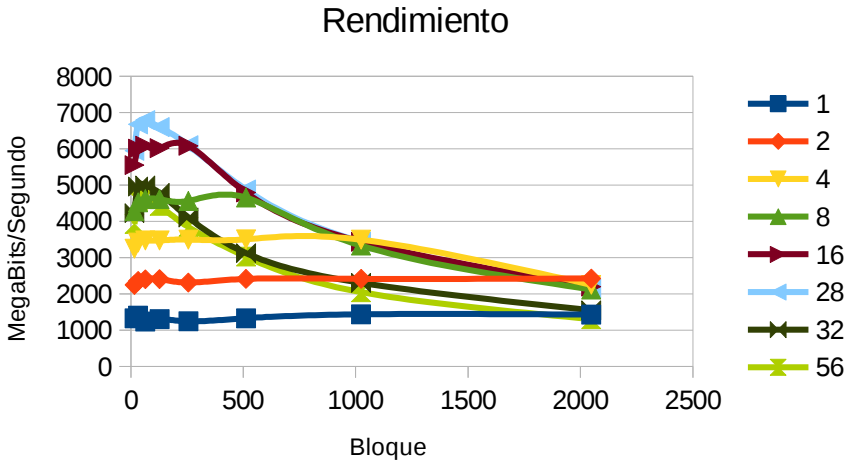


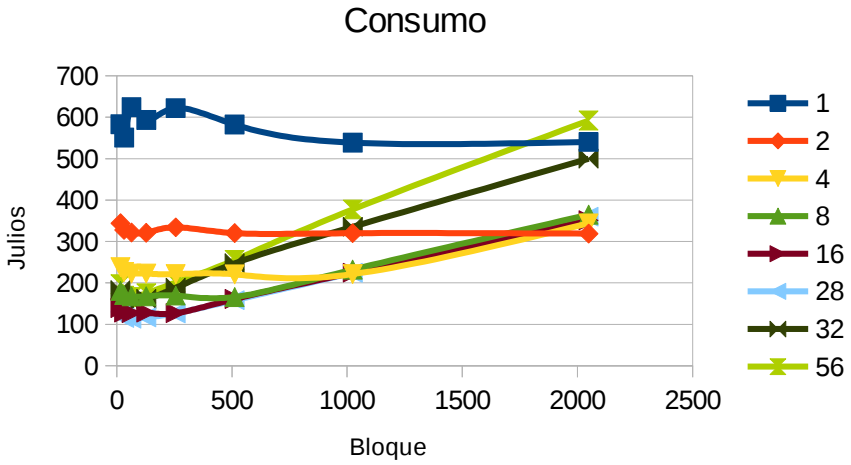




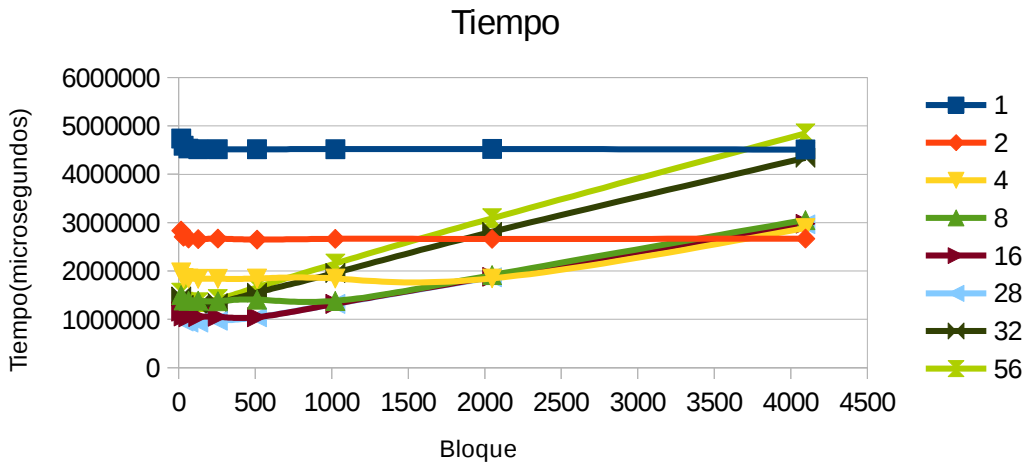
- Tamaño de imagen 4096x4096, utilizando entre 1 y 56 hilos de ejecución.

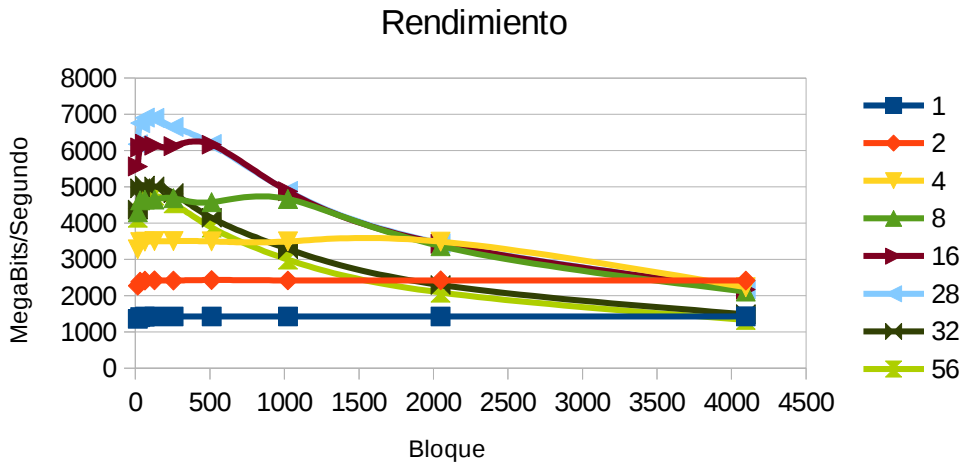


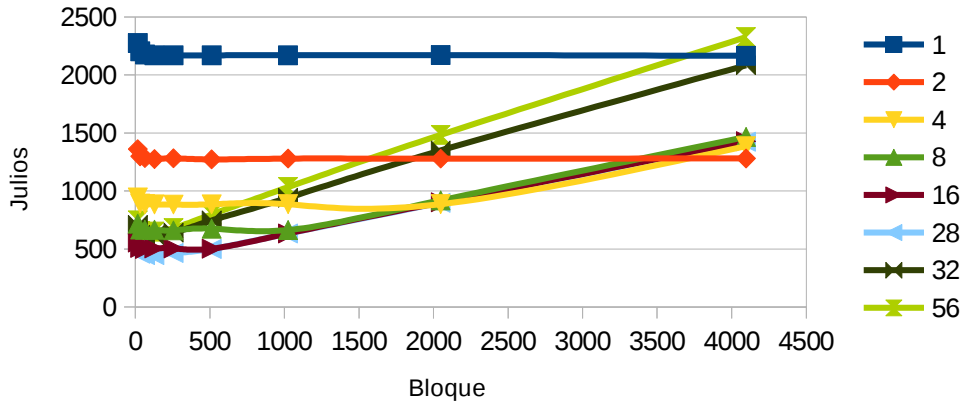




- Tamaño de imagen 8192x8192, utilizando entre 1 y 56 hilos de ejecución.







Discusión de resultados

Al analizar una series de distintos tamaños para una misma imagen sobre esta arquitectura, podemos apreciar una serie de tendencias y observaciones generales:

- Tiempo de ejecución

    1. El uso de la tecnología Hyperthreading implica que sólo ante el uso de tantos threads como cores físicos hay disponibles en el sistema se alcance un rendimiento óptimo. Aumentar el número de hilos de ejecución por encima de dicho límite degrada el rendimiento para todos los tamaños de problema.
    2. En general, el rendimiento aumenta sustancialmente a medida que el tamaño de imagen aumenta. Típicamente, el procesamiento de imágenes de mayor tamaño implica la posibilidad de utilizar tamaños de bloque mayores, con mejor aprovechamiento de la jerarquía de memoria.

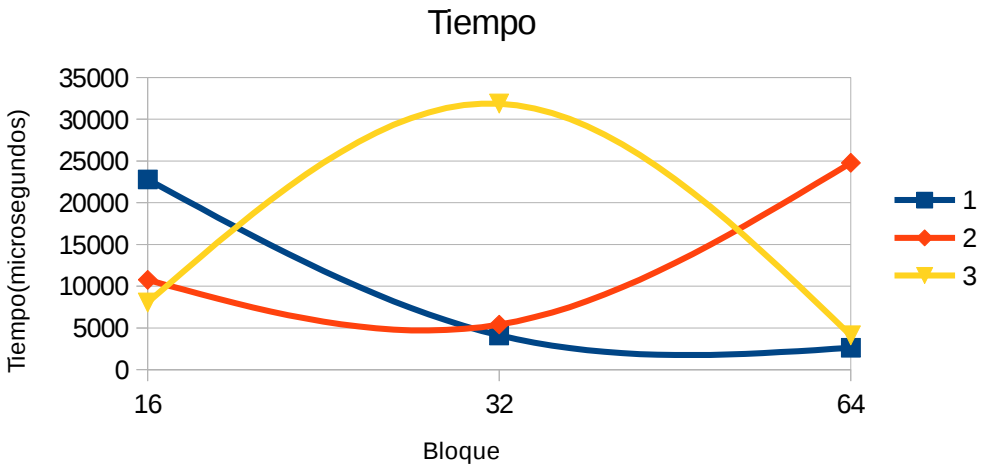
- Tamaños de bloque

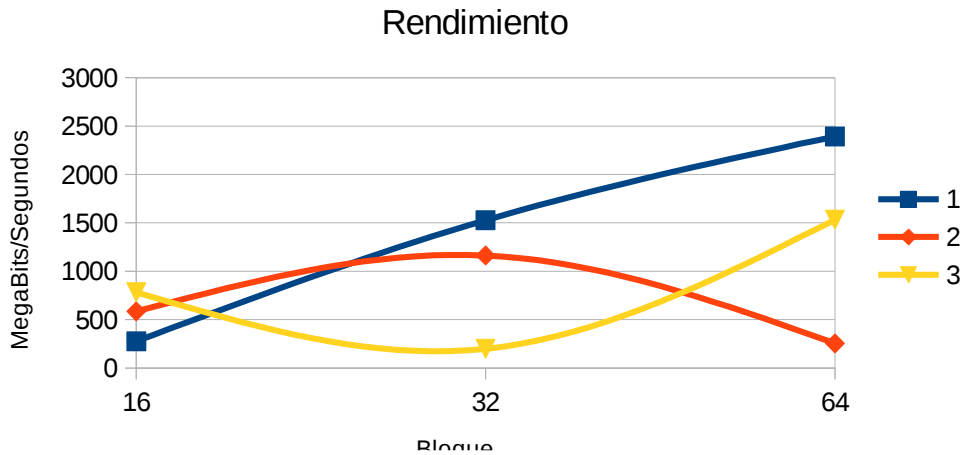
    1. Al igual que en el apartado anterior, sólo ante un número reducido de bloques tiene sentido utilizar un número reducido de hebras. En general, como es lógico, sólo cuando el número de bloques es considerable resulta beneficioso aumentar el número de hebras de ejecución.

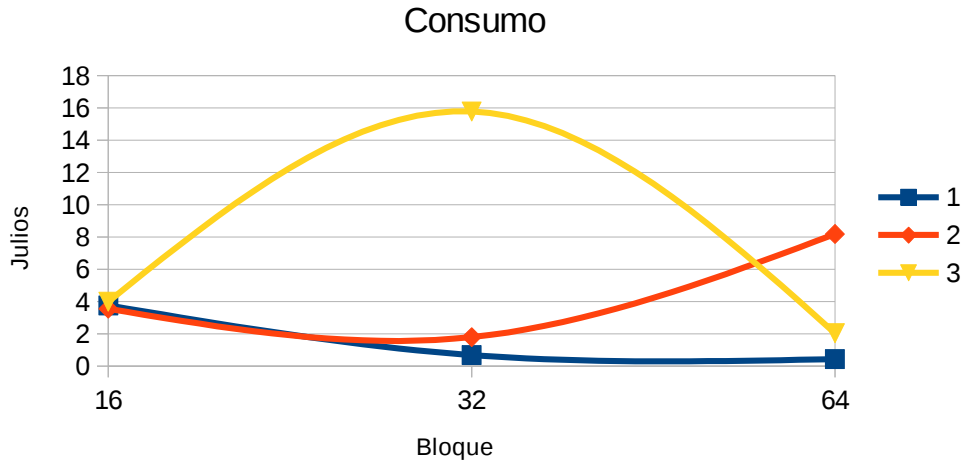
Utilización exclusiva de GPUs (Bujaruelo)

Por limitaciones en la cantidad de memoria disponible en GPU, sólo ha sido posible testear imágenes de hasta un tamaño de 4096x4096.

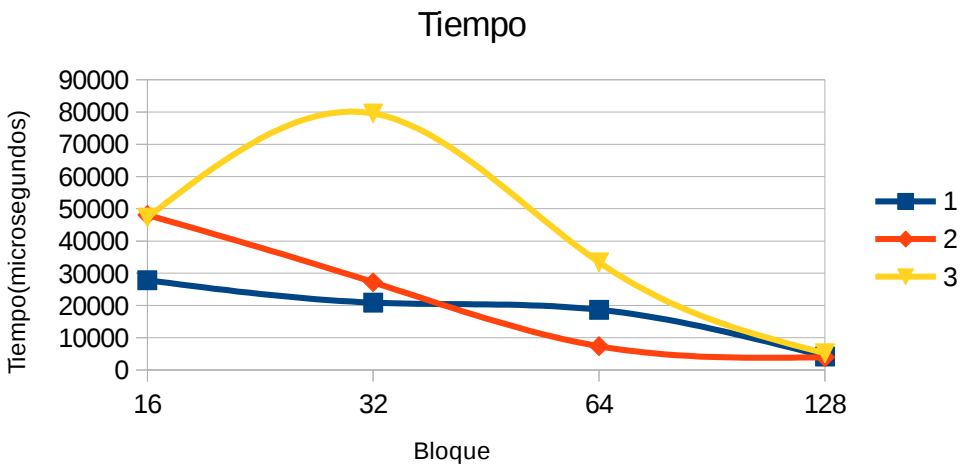
- Tamaño de imagen 128x128, utilizando 1, 2 y 3 GPUs.

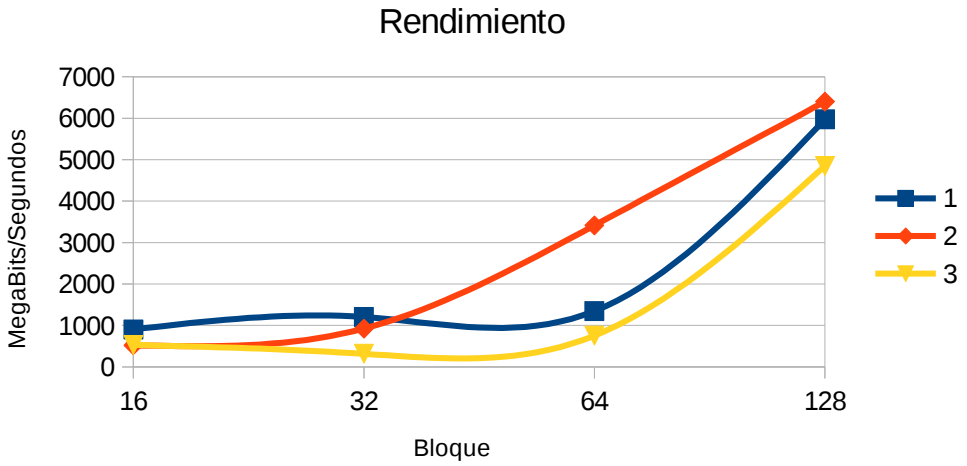


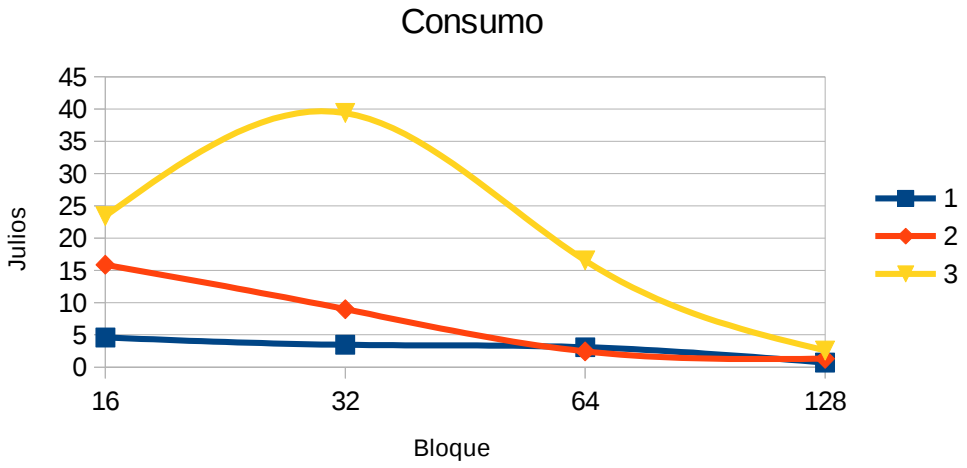




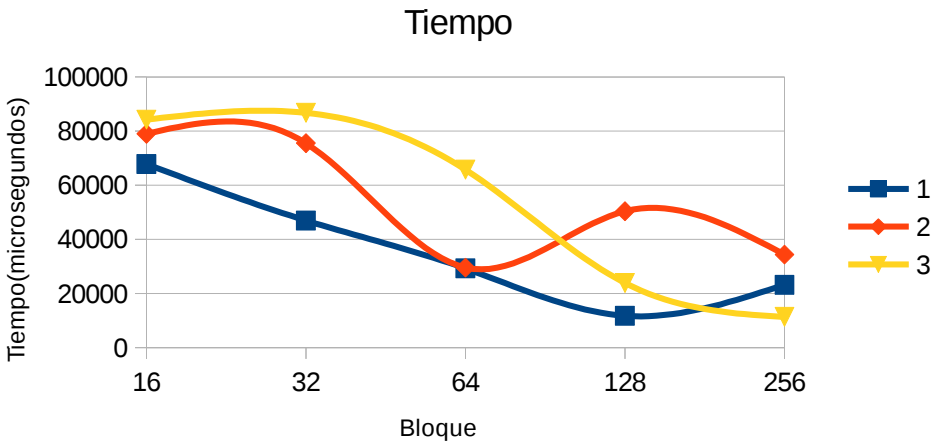
- Tamaño de imagen 256x256, utilizando 1, 2 y 3 GPUs.

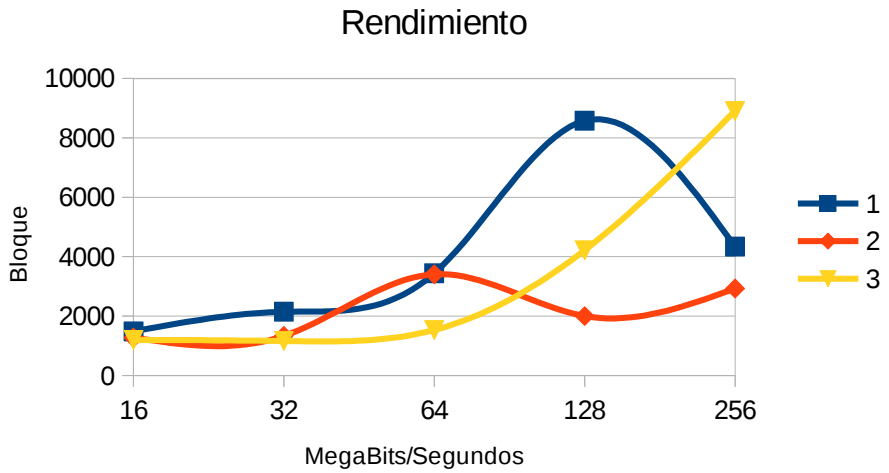


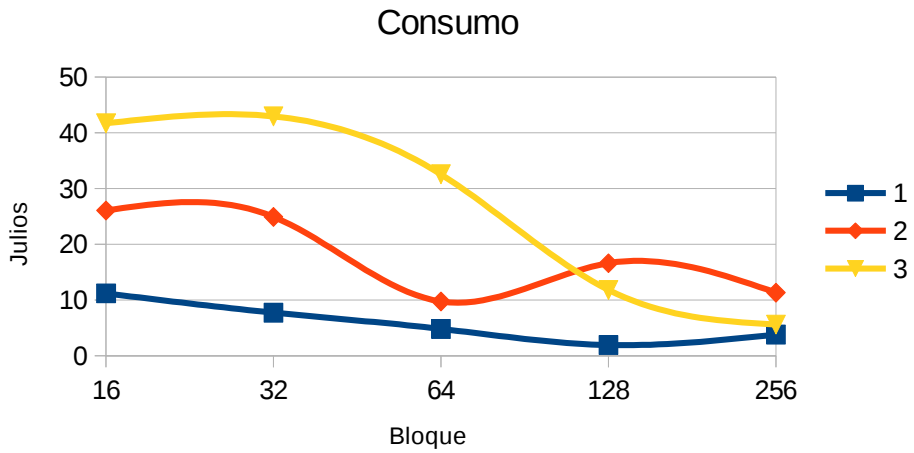




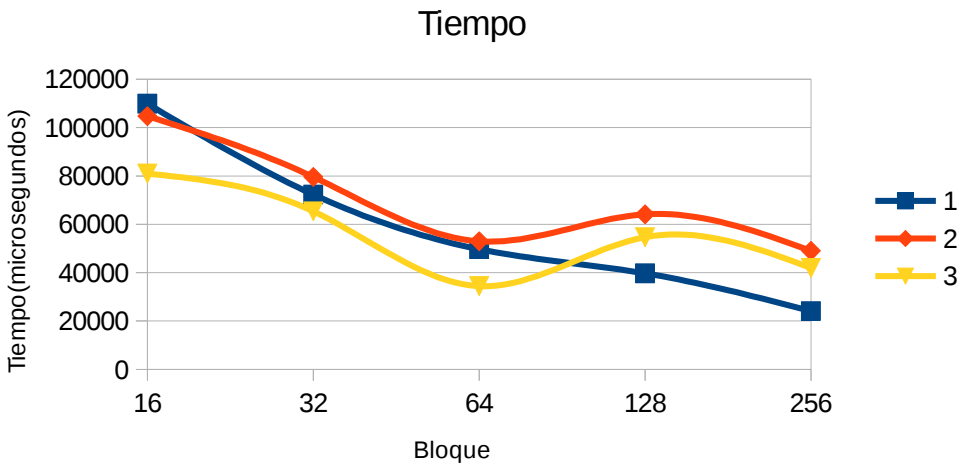
- Tamaño de imagen 512x512, utilizando 1, 2 y 3 GPUs.

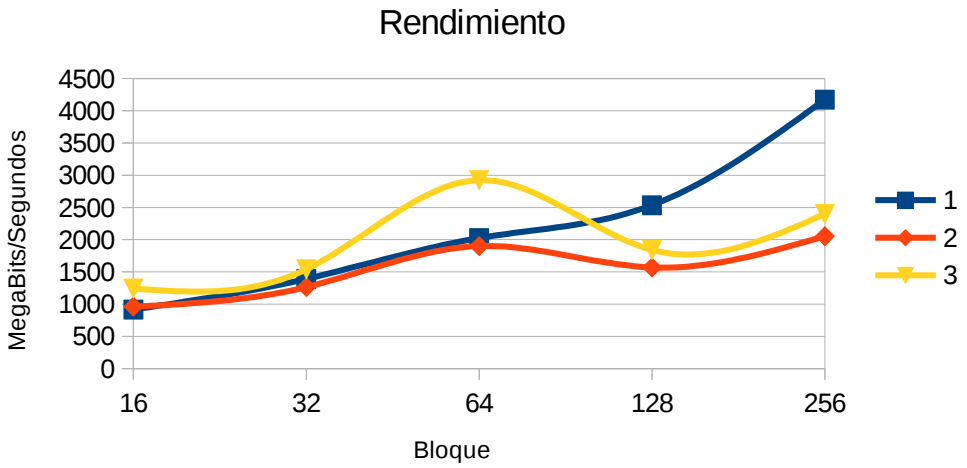


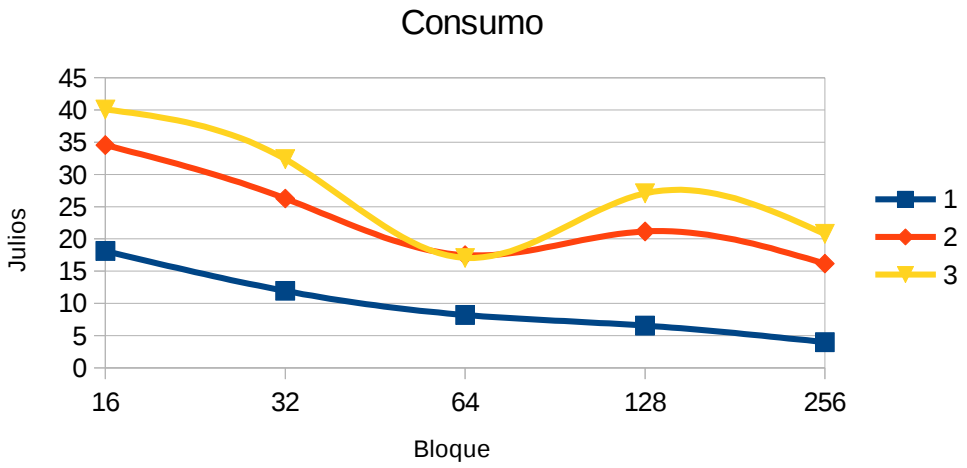




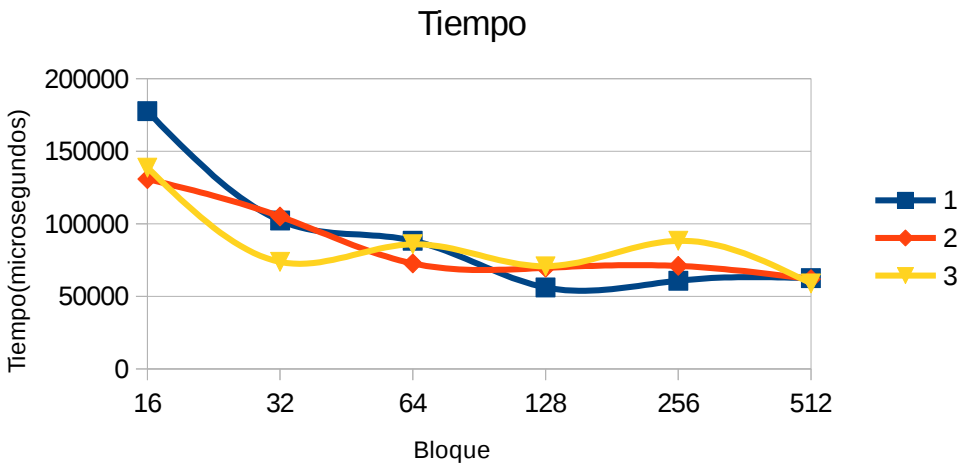
- Tamaño de imagen 768x768, utilizando 1, 2 y 3 GPUs.

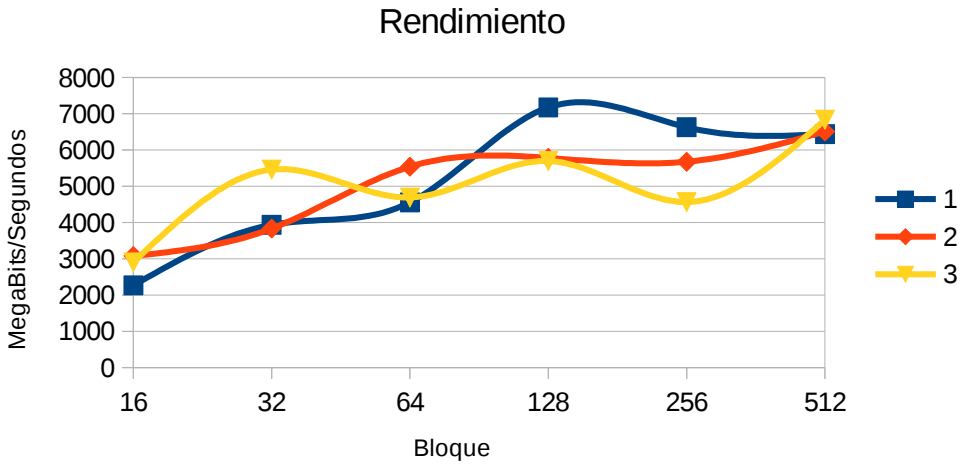


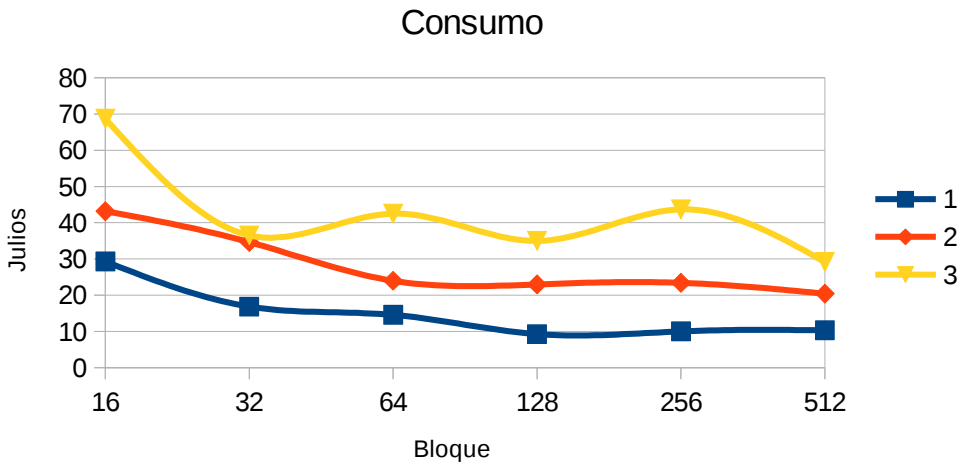




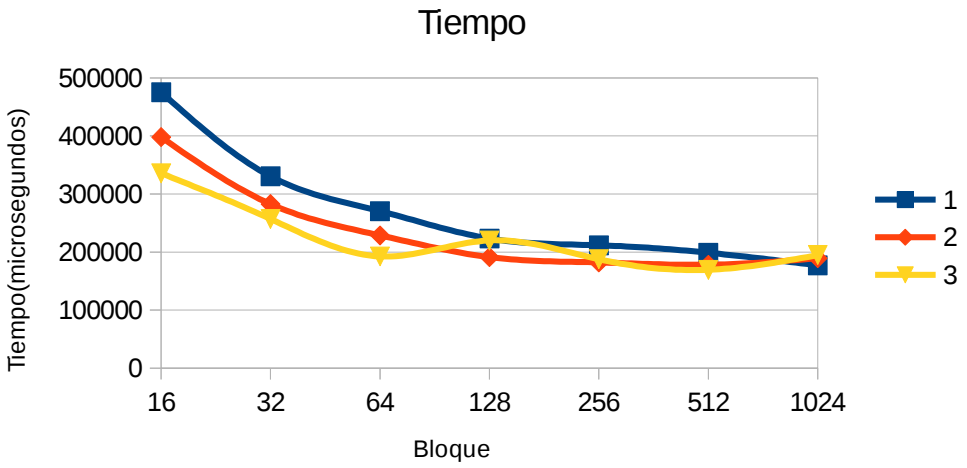
- Tamaño de imagen 1024x1024, utilizando 1, 2 y 3 GPUs.

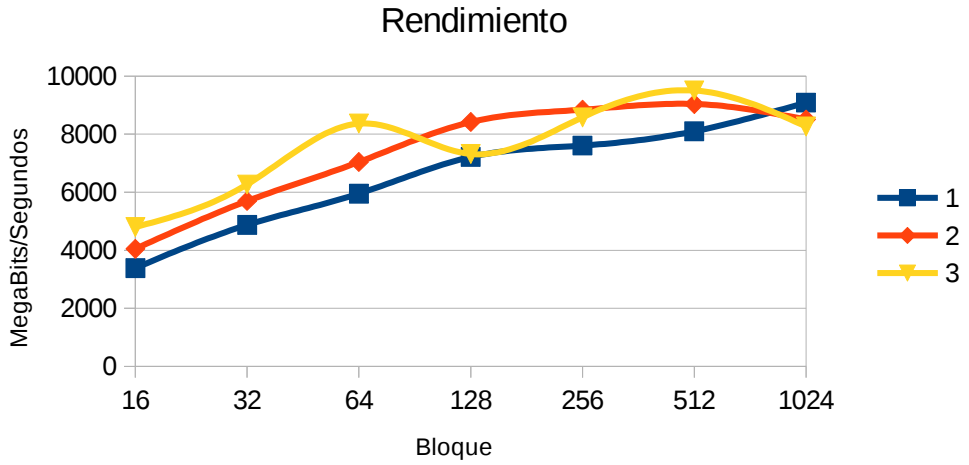


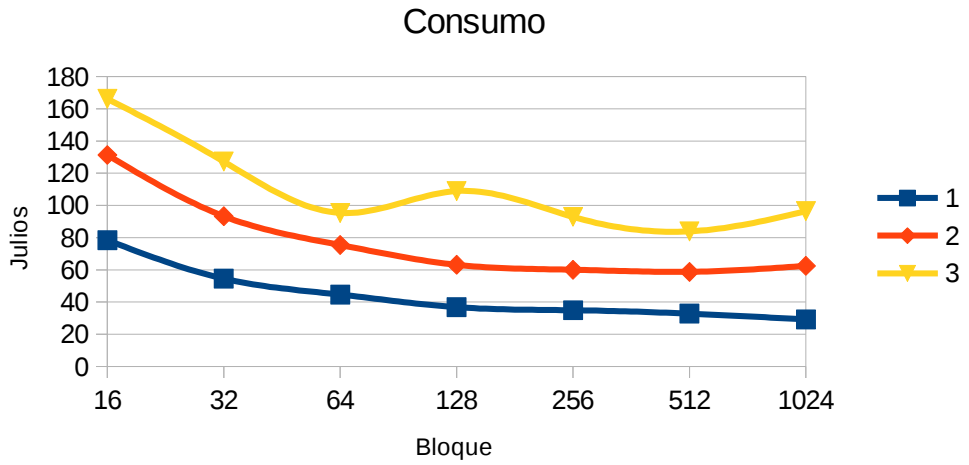




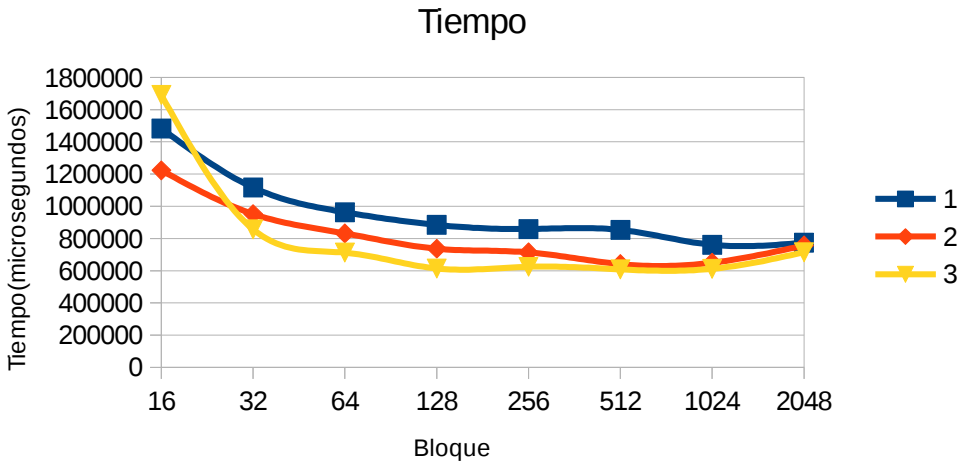
- Tamaño de imagen 2048x2048, utilizando 1, 2 y 3 GPUs.

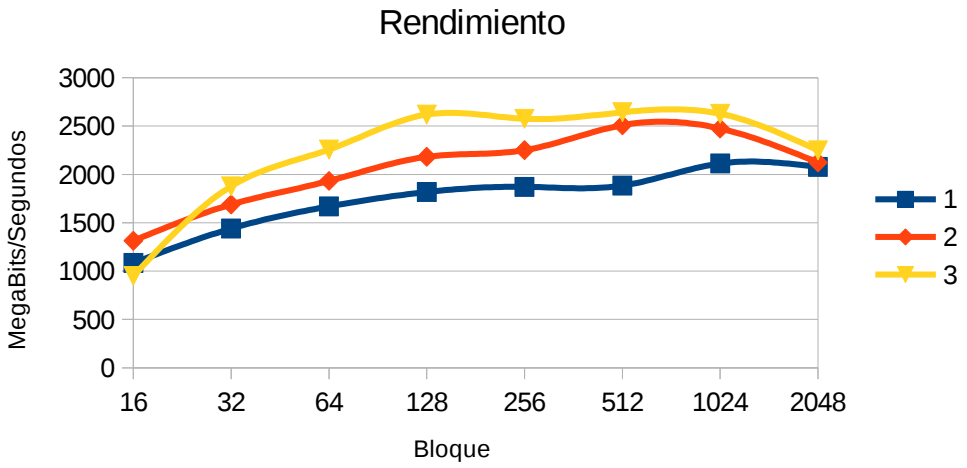


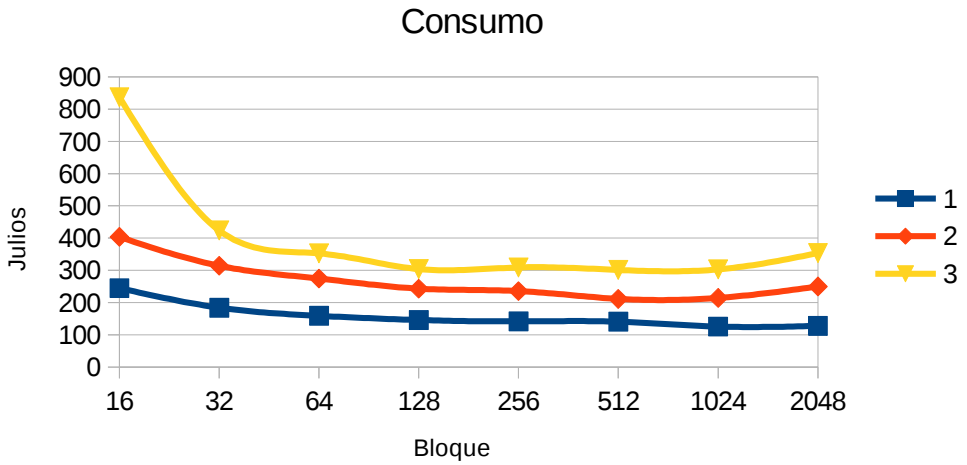




- Tamaño de imagen 4096x4096, utilizando 1, 2 y 3 GPUs.







Apreciaciones generales y observaciones.

Al analizar una series de distintos tamaños para una misma imagen, podemos apreciar una serie de tendencias:

- Tiempo

  En imágenes relativamente pequeñas la diferencia entre lanzar a 1, 2 o 3 GPUs no es significativa, pero si se puede apreciar una tendencia a ser más rápidas al tener 3GPUs cuanto mayor es la imagen. El problema surge por la limitación de OmpSs a la hora de gestionar imágenes mayores a 4096x4096; en este sentido, a la vista de la tendencia, se espera mayor rendimiento al aumentar significativamente el tamaño de la imagen por encima de este límite.

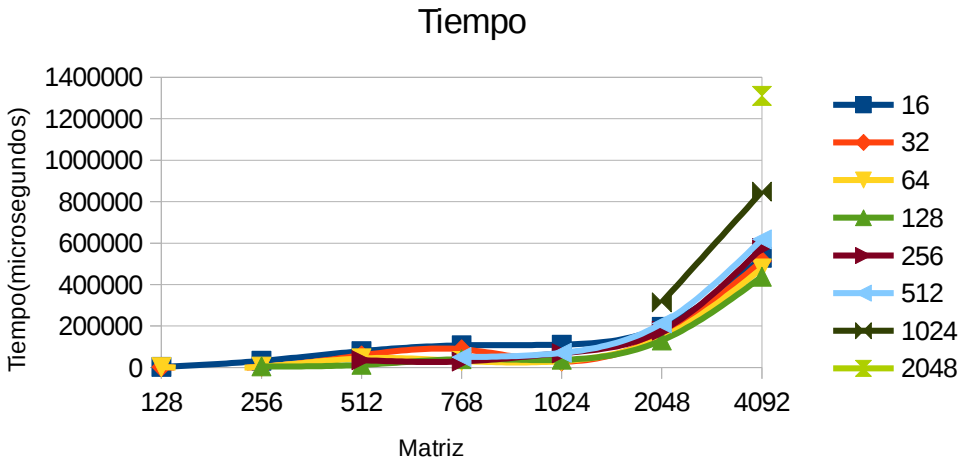
- Rendimiento

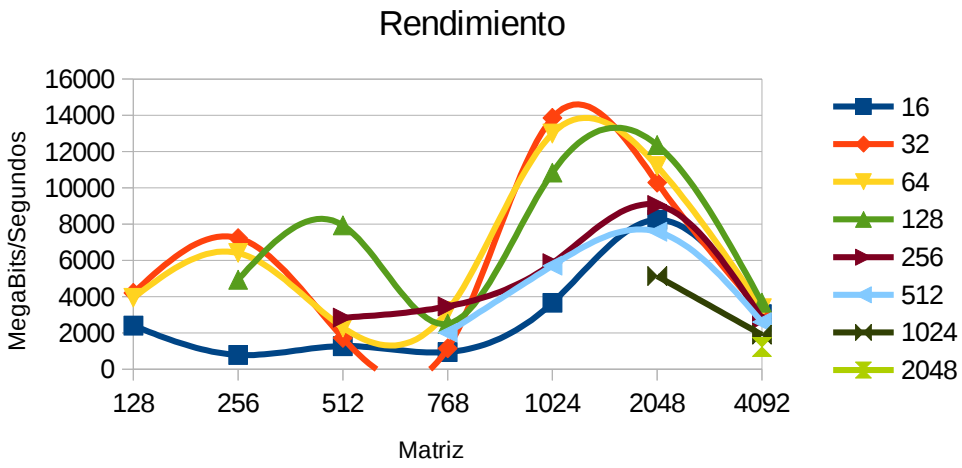
  Al igual que en el apartado anterior, cuanto mayor es la imagen y por tanto los bloques son mayores, obtenemos un rendimiento mayor teniendo al aumentar el número de GPUs.

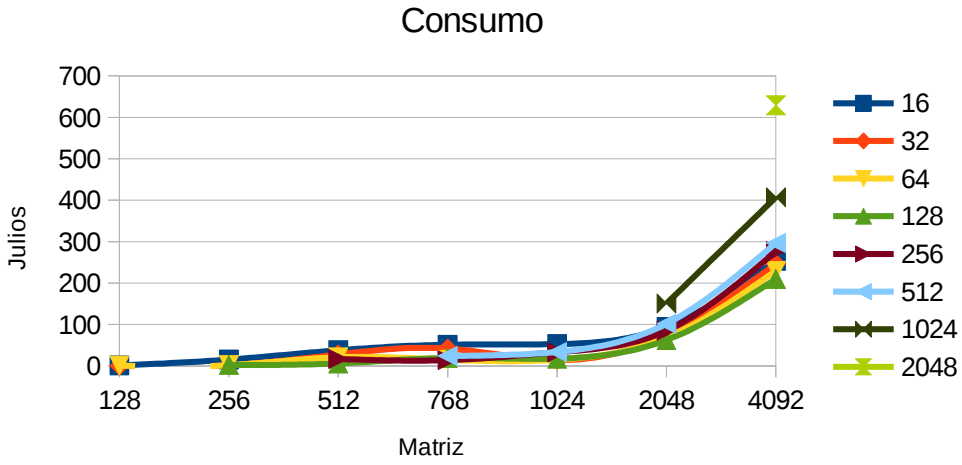
- Consumo

  Esta ligado al tiempo de ejecución, cuanto mayor sea el tiempo mayor consumo, por lo tanto cuanto más GPUs tengamos, siempre que la imagen sea relativamente grande, será de menor consumo. Dada la limitación en rendimiento asociada a los límites en el tamaño de imagen, el consumo energético es mayor que al utilizar exclusivamente núcelos de propósito general. Sin embargo, como se ha comentado, se supone una mayor eficiencia energética al aumentar el tamaño de imagen (y por tanto verse aumentado significativamente el rendimiento).

Utilización conjunta de CPU y GPU (Bujaruelo)







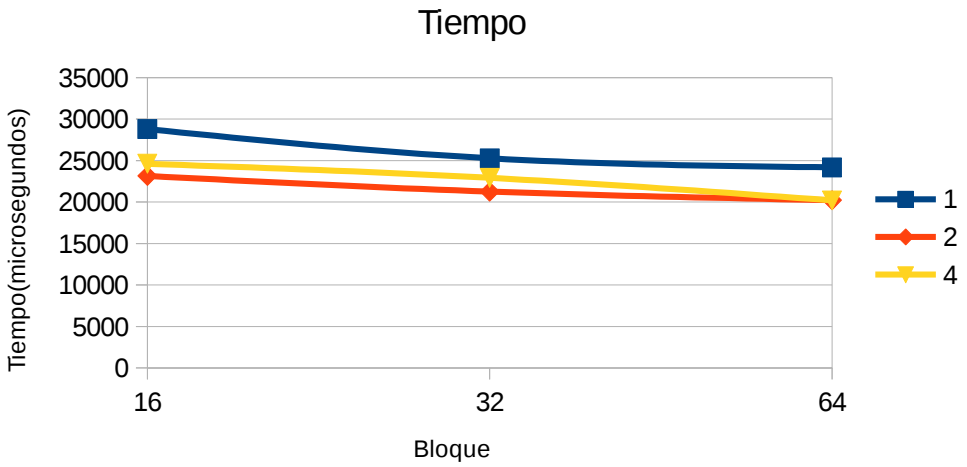
Apreciaciones generales y observaciones

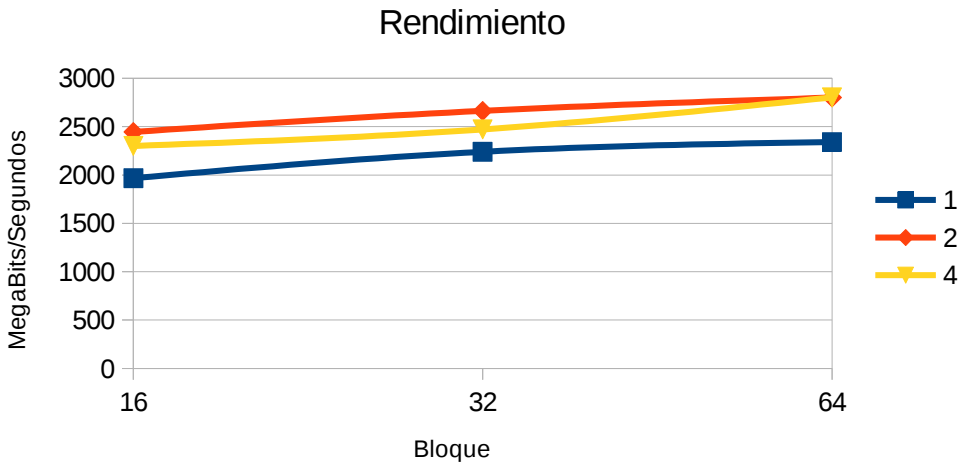
Al combinar CPU y GPU, se puede apreciar un mayor aprovechamiento de los recursos, bajando los tiempos de ejecución, aumentando significativamente el rendimiento y reduciéndose el consumo.

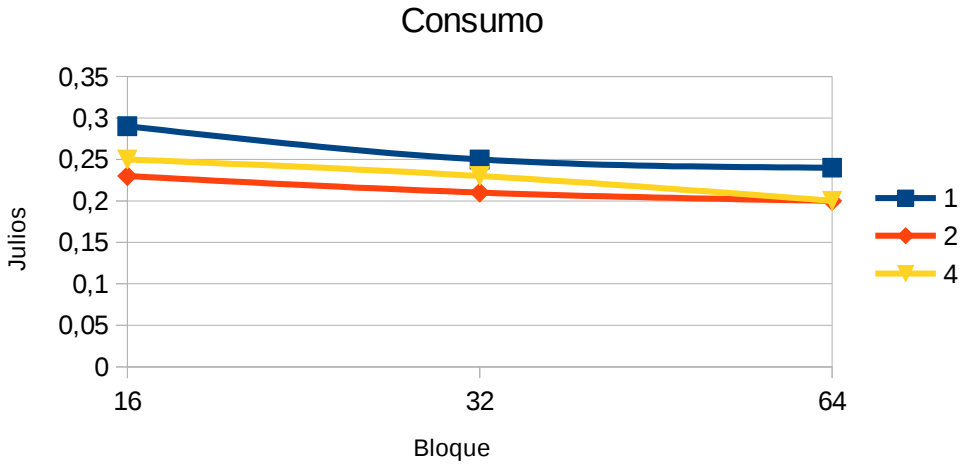
Utilización exclusiva de cores ARM (K2H)

En este caso, la limitación de la máquina no permite testear con imágenes mayores de 768x768.

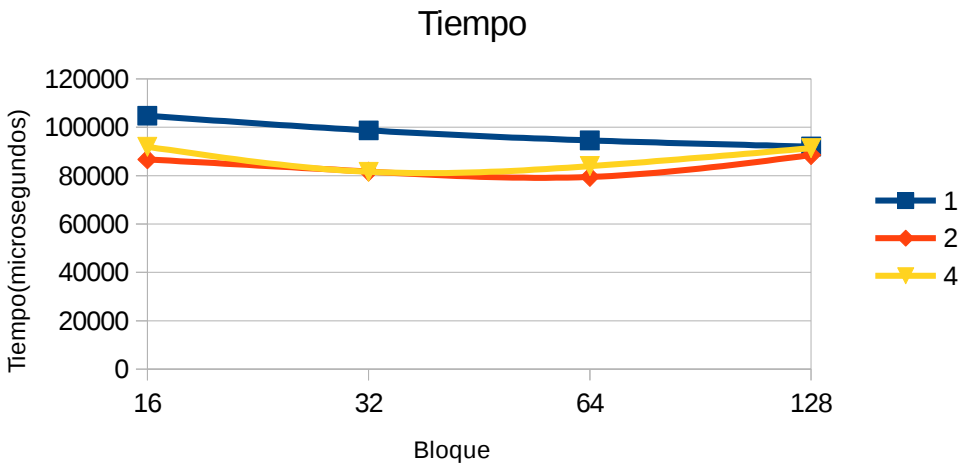
- Tamaño de imagen 128x128, utilizando 1, 2 y 4 cores.

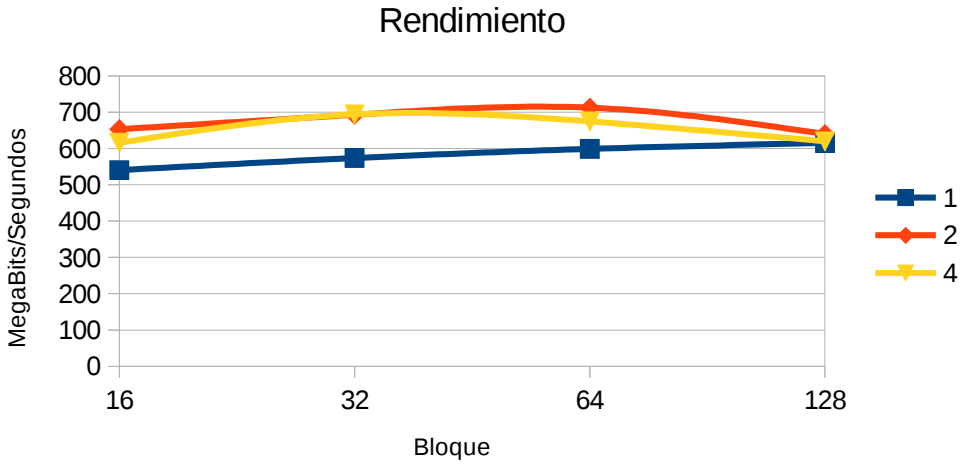


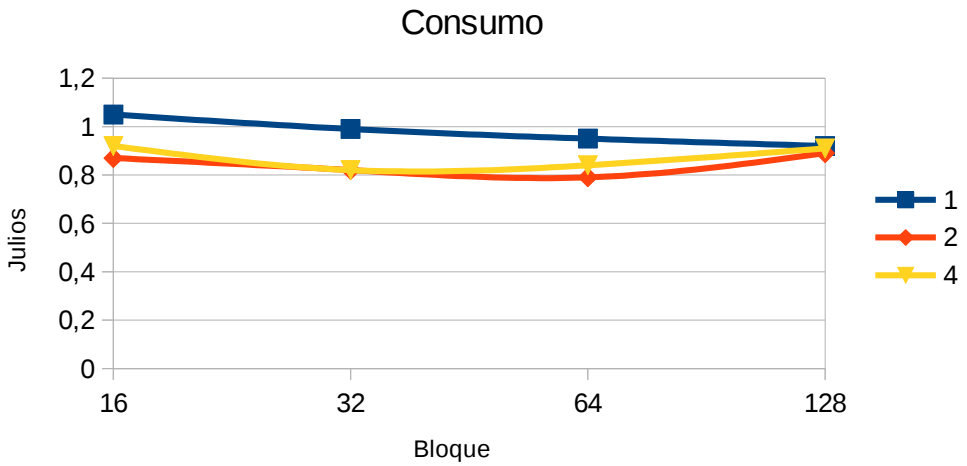




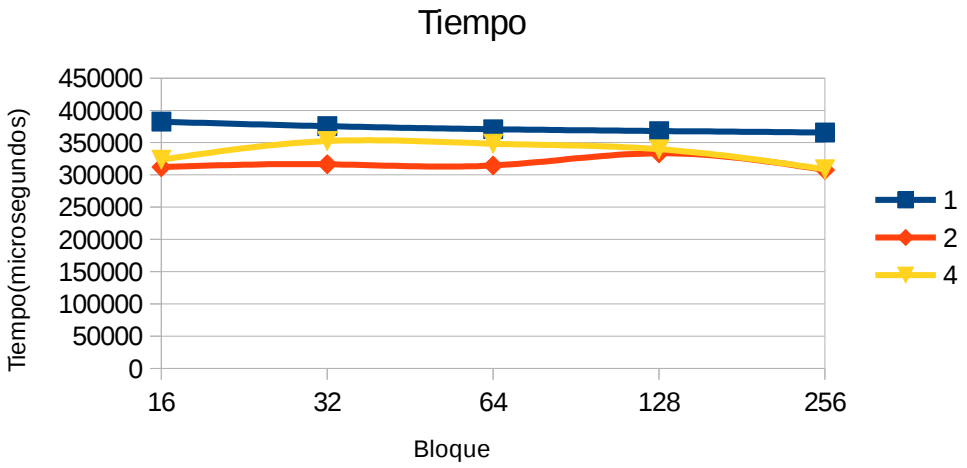
- Tamaño de imagen 256x256, utilizando 1, 2 y 4 cores.

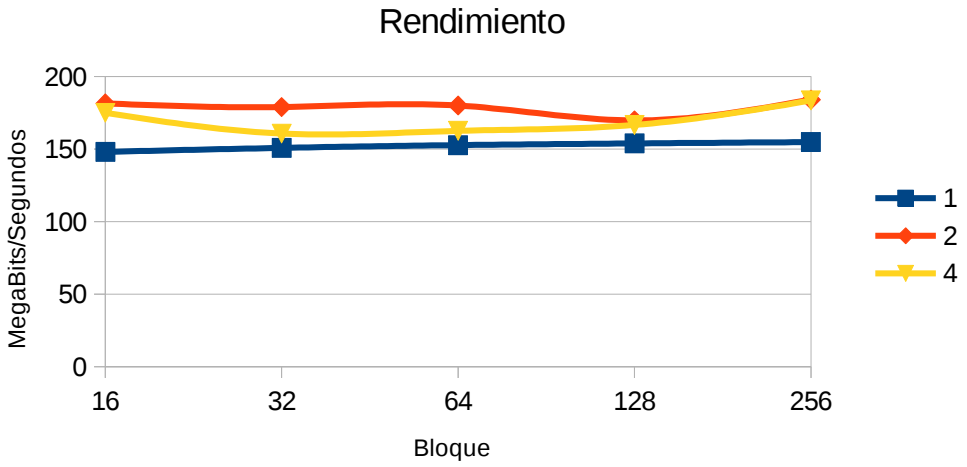


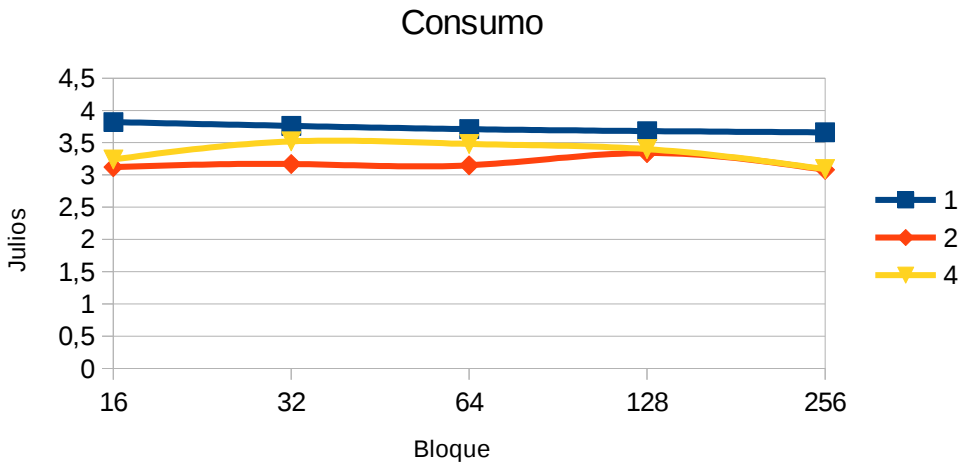




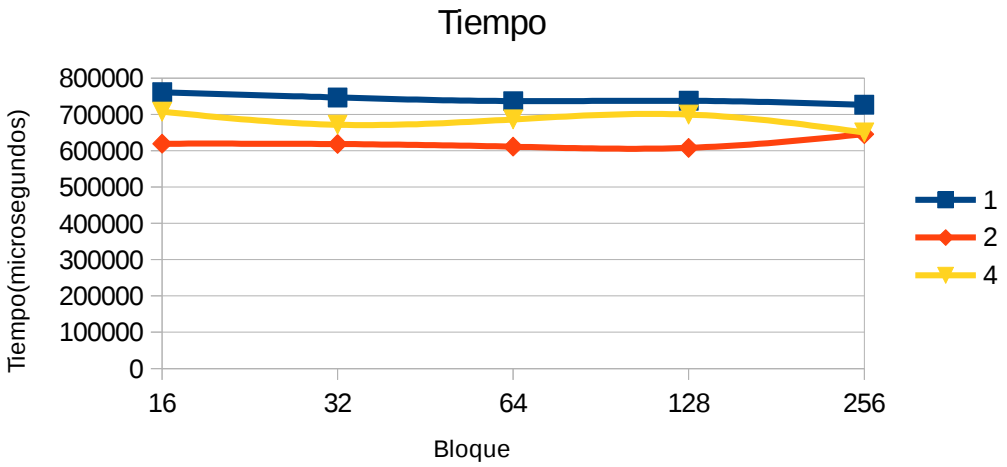
- Tamaño de imagen 512x512, utilizando 1, 2 y 4 cores.

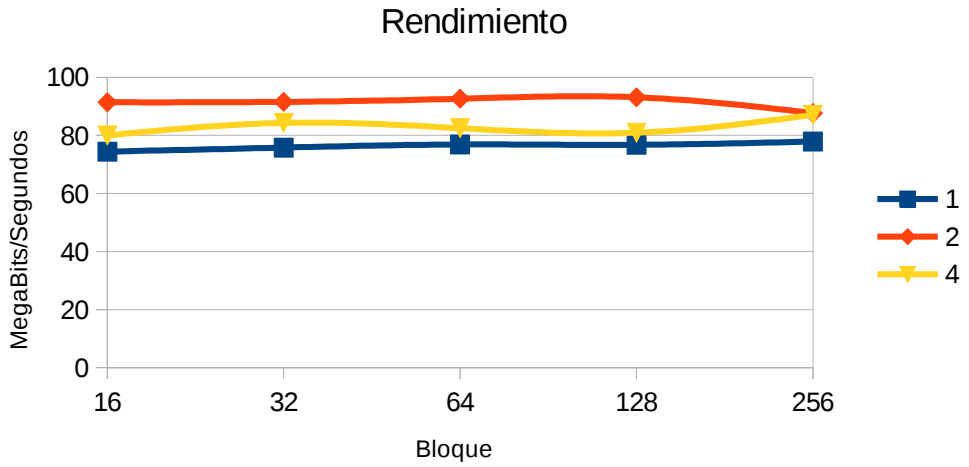


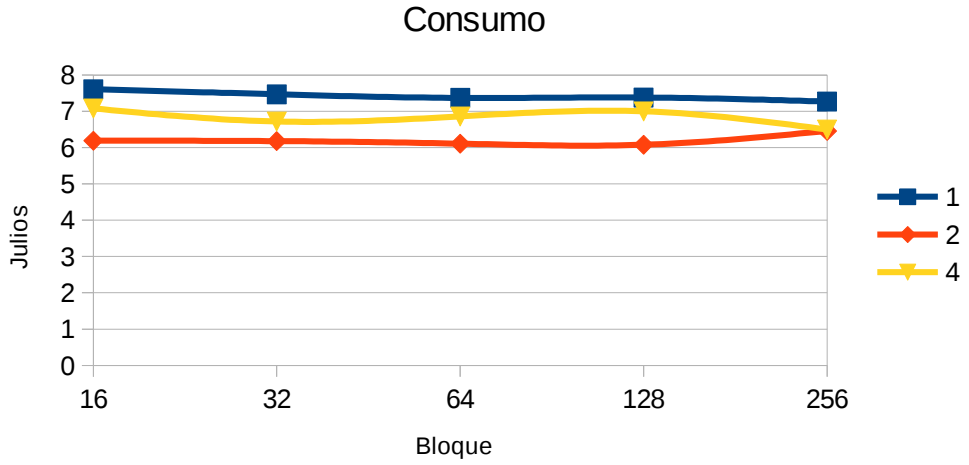




- Tamaño de imagen 768x768, utilizando 1, 2 y 4 cores.







Apreciaciones

Al analizar las series de experimentos para distintos tamaños para una misma imagen, podemos apreciar una serie de tendencias:

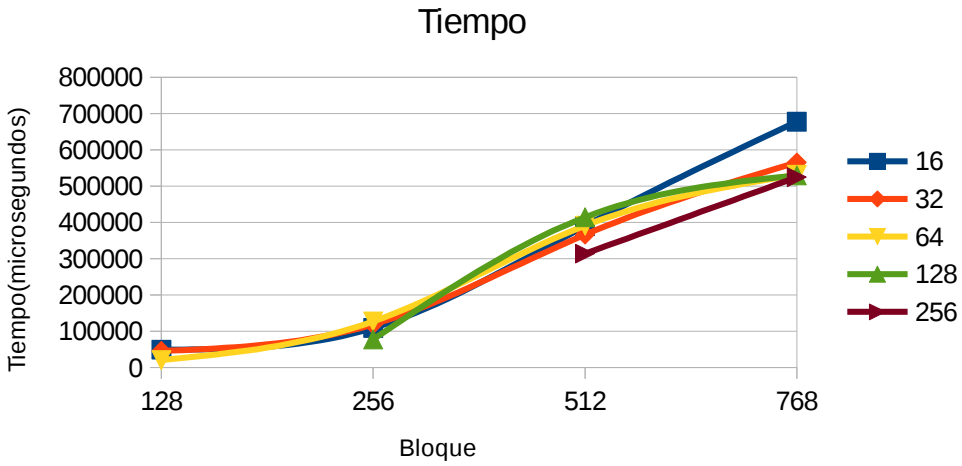
- Tiempo de ejecución.

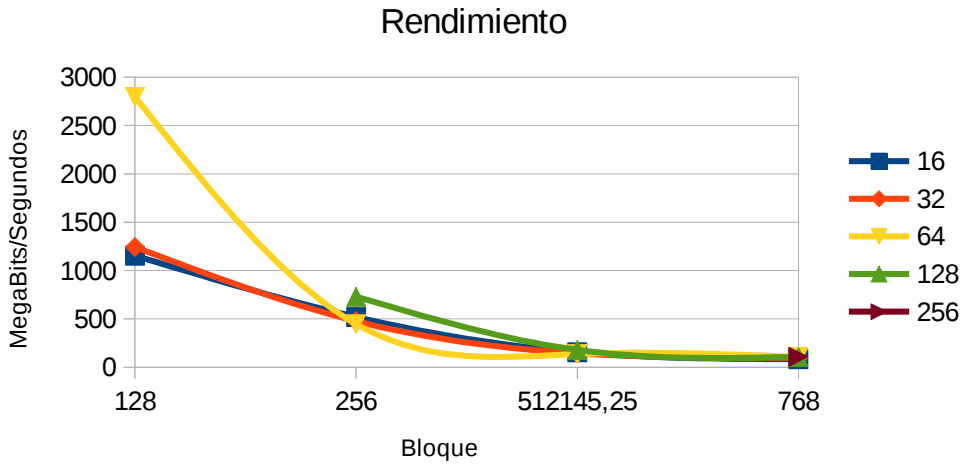
  No existe en este caso una gran diferencia entre la elección de uno, dos o cuatro threads; se puede apreciar una pequeña mejora con dos threads, pero no al aumentar hasta cuatro threads. La razón estriba en lo costoso del proceso de planificación para tamaños de bloque de reducidas dimensiones con respecto al breve tiempo de procesamiento.

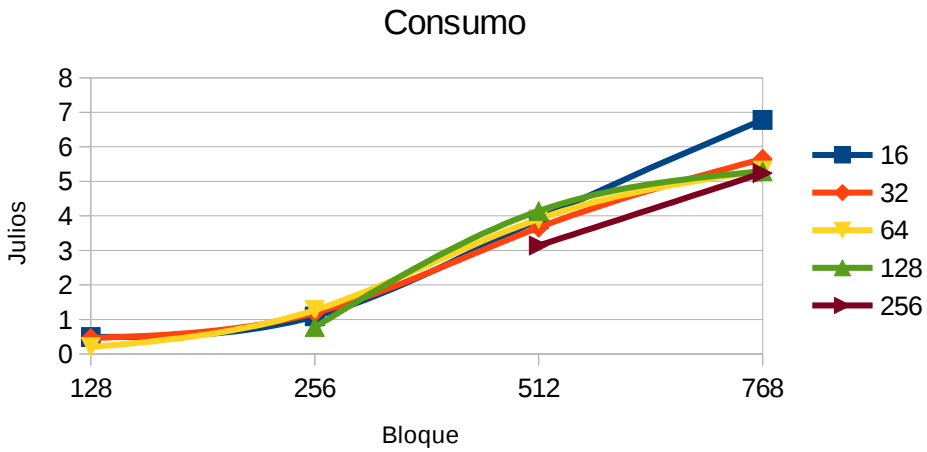
- Rendimiento/consumo.

  Comparativamente, el rendimiento es mucho menor que en una máquina de alto rendimiento. Sin embargo, el consumo energético es mucho menor (y por tanto la eficiencia energética mayor). Este es uno de los principales puntos fuertes de este tipo de arquitecturas.

Utilización conjunta de ARM y DSP (K2H)







Apreciaciones

En general, los tiempos de ejecución no son significativamente mejores al introducir el uso del DSP

como coprocesador. Sin embargo, la limitación en la cantidad de memoria disponible hace que tanto el tamaño de las imágenes como el tamaño de bloque se vea seriamente limitado, y por tanto el rendimiento obtenido no sea significativamente mejor. Se espera, no obstante, que dicho rendimiento mejore para imágenes de mayor tamaño (siempre que el tamaño de memoria deje de ser una limitación en futuras generaciones).

5 Conclusiones

En este trabajo se ha desarrollado una aplicación completa para la detección de bordes en imágenes en color explotando paralelismo a nivel de tareas. Aunque el objetivo general planteado consistió en realizar una implementación portable de dicho código sobre una arquitectura de proposito específico formada por procesadores ARM y acelerada mediante DSP, el desarrollo del proyecto ha permitido comprobar el funcionamiento y portabilidad de la misma, con ningún cambio en el código, sobre otro tipo de arquitecturas. Éstas incluyen procesadores de alto rendimiento y múltiples GPUs en un mismo sistema.

Los principales hitos conseguidos se resumen en:

1. Se ha diseñado, implementado y evaluado el rendimiento de una implementación del algoritmo de Canny explotando paralelismo a nivel de tareas.
2. Internamente, se han desarrollado kernels utilizando los paradigmas CUDA y OpenCL para explotar el paralelismo interno de cada tipo de acelerador.
3. Se ha portado y evaluado, sin cambios en el código, dicha implementación a arquitecturas radicalmente distintas, basadas en GPUs y DSPs.
4. Se ha utilizado el mecanismo proporcionado por OmpSs para la ejecución concurrente de tareas en CPU y acelerador de forma transparente.

Los resultados obtenidos demuestran la posibilidad de explotar los DSPs como plataforma de aceleración de código, y la posibilidad de utilizar un sistema basado en planificador de tareas (OmpSs) sobre este tipo de plataformas. Aunque los resultados obtenidos no son alentadores en términos de rendimiento, muchos de ellos se basan en las limitaciones actuales en cuanto a cantidad de memoria de los DSPs de nueva generación. La evaluación de los mismos códigos sobre plataformas similares futuras resultará trivial, puesto que no será necesario reimplementar los códigos. Cabe destacar que se trata de la primera experimentación sobre este tipo de plataformas utilizando OmpSs encontrada en la literatura.

Como trabajo futuro, se propone la evaluación de otro tipo de implementaciones que exhiban paralelismo a nivel de tareas, la utilización de entornos precisos de medición de consumo energético y la optimización del código interno de las tareas para acelerar el tiempo de ejecución individual de cada una de ellas.

Conclusions

In this work, we have developed a complete application for edge detection implementing the Canny Algorithm on color images exploiting task parallellism. Although the main goal was to develop a portable implementation targeting ARM processors accelerated via DSPs, the development of the project derived into an evaluation of the code also on other types of architectures. This has shown the possibility of porting the developed codes to a number of architectures, including CPUs and multi-GPU architectures.

The main milestones achieved can be summarized as:

1. A complete implementation of the Canny Algorithm exploiting task-level parallelism has been designed, implemented and evaluated relying on a runtime task scheduler.
2. Internally, the necessary kernels (task codes) have been developed using both CUDA and OpenCL, depending on the underlying hardware, in order to extract data parallelism from each accelerator.
3. Without changes in the code, we have ported and evaluated the performance of the implementation on dramatically different architectures, namely GPU and DSP-based.
4. We have leveraged the mechanism offered by OmpSs to allow the concurrent execution of the tasks on the CPU and the accelerator in a transparent fashion.

The obtained results show the possibility of exploiting the DSPs as an accelerating platform, and the feasibility of task schedulers to be used on this type of platforms. Although the attained results are not optimal in terms of performance, many of them are based on current hardware limitations. It is important to remark the ease of programming offered by task-based paradigms relying on underlying runtime task schedulers. This is, in fact, the first experimentation with OmpSs on DSP-based architectures found in the literature.

As future work , we propose the evaluation of other types of implementations that exhibit task parallelism, the usage of precise power consumption environments and the optimization of the implementations of tasks in order to improve performance.

6 Bibliografía

[1] Benedict R.Gaster, Lee Howes, David R.Kaeli, Perhadd Mistry, Dana Schaa(2012): Heterogeneous Computing with OpenCL.

[2] *Moore, Gordon E. (1965-04-19). "Cramming more components onto integrated circuits" (PDF). Electronics. Retrieved 2011-08-22.*

[3] CUDA Toolkit Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide.

[4] The OmpSs Programming Model. https://pm.bsc.es/ompss.