Organización y Arquitectura de Computadores II

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico Final

Programación de sistemas heterogéneos utilizando OpenCL

Integrante	LU	Correo electrónico
Vassiliev Saveli	780/09	vassiliev.sav@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Objetivos

El objetivo principal de este trabajo es aprender a programar sistemas heterogéneos utilizando el estándar OpenCL 1.1. Por sistemas heterogéneos en este trabajo se entiende hacer uso del poder de procesamiento de placas gráficas (GPU) junto a los procesadores convencionales (CPU). Para ello presentamos una descripción del modelo de ejecución / memoria que ofrece este estándar implementado por AMD. Para facilitar la explicación de dichos temas presentamos varios programas de ejemplo con explicaciones detalladas. Finalmente presentamos varios ejemplos de mayor interés práctico.

2. Introducción a OpenCL 1.1

El estándar OpenCL es un proyecto relativamente nuevo, que intenta proveer una API para programar sistemas heterogéneos, abstrayendo los detalles específicos del hardware subyacente. El estándar permite interactuar con procesadores convencionales, con placas de video de diversos fabricantes y con Cell B.E. En este trabajo solo utilizaremos las GPU.

Los CPU suelen estar pensados para ejecutar instrucciones en forma serial (SISD - Single Instruction Single Data), o con alguna extensión agregada (por ejemplo SSE) permitir instrucciones con múltples datos (SIMD - Single Instruction Multiple Data). En cambio, las GPU suelen tener muchos núcleos, que comparten el ciclo "fetch-decode-execute", y por lo tanto ejecutan una misma instrucción en muchos hilos diferentes, es decir, son paralelos a nivel de hilos (threads). Este enfoque se denomina Single Instruction Multiple Threads (SIMT).

Una consecuencia importante es que los condicionales en un programa que se ejecuta en GPU, pueden generar una degradaciónes significativas de performance. Supongamos el siguiente ejemplo:

```
1 if(C)
2   f()
3 else
4   g()
```

Si existen por lo menos dos threads en los cuales C tenga valor de verdad diferente, primero se ejecuta $f()$ en los threads donde C sea verdadero, suspendiendo cualquier tipo de operación en los threads donde C es falso; y una vez terminada la ejecución de $f()$ se ejecutará $g()$ en los threads que corresponda, dejando inactivos los threads que acaban de ejecutar $f()$. Es decir, el tiempo aproximado de la ejecución será $T_f + T_g$. En cambio en un modelo paralelo a nivel tareas, este tiempo puede aproximarse con máx$\{T_f, T_g\}$, donde T_f (ó T_g) denota el tiempo de ejecutar $f()$ (ó $g()$) en el procesador mas lento de los involucrados.

2.1. Modelo de ejecución

El modelo representa al sistema como un conjunto de plataformas (Platforms). En nuestro caso la plataforma es única, y representa la computadora. Cada plataforma tiene un conjunto de dispositivos (Devices), estos dispositivos son las unidades de procesamiento, en nuestro caso el conjunto de dispositivos está formado por la CPU y la placa de video. A su vez, cada dispositivo tiene unidades de cómputo (Compute Units), que son un conjunto de elementos de procesamiento (Processing Units), en el caso de un CPU serían los cores del mismo. En el caso de las GPU, los Processing Units son agrupados en Compute Units.

Cada Processing Unit será encargado de ejecutar una instancia de un Kernel. El Kernel es código que será paralelizado en un Device, mediante copias del programa, pero con inputs diferentes.

A la hora de programar en OpenCL debemos implementar aquel programa que inicializa las estructuras necesarias para manejar todos los dispositivos deseados, inicializa los Kernels, los parámetros de los threads utilizando buffers de memoria, etc.. Este programa es ejecutado en el Host Device, usualmente tendremos un Host que controla el resto de los dispositivos, y hace de interfáz con el usuario. El código del mismo puede estar escrito en C, C++ o Python. En este trabajo práctico utilizaremos la versión de

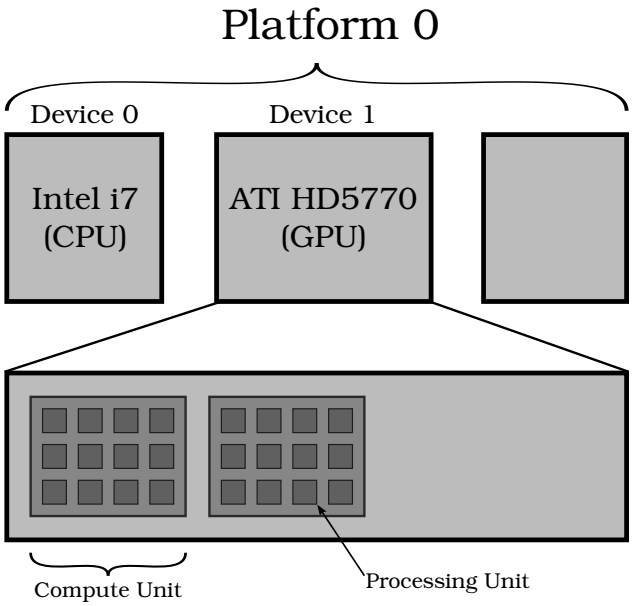

Figura 1: Modelo de ejecución

C++. Por otro lado debemos implementar los programas que serán ejecutados por los Compute Devices, es decir, el código de los Kernels. Los Kernels se programan en una modificación del lenguaje C.

La compilación del Host se hace del modo habitual, utilizando las bibliotecas de OpenCL. La compilación de los Kernel usualmente es online, es decir, en cada ejecución del binario Host se recompila el Kernel. Si bien existe una forma de generar un binario del Kernel, no se suele hacer.

2.2. Modelo de memoria

Hemos visto que el estándar admite diferentes dispositivos para llevar a cabo la ejecución del cómputo, cada uno de estos dispositivos tiene su propia memoria, que debe ser utilizada correctamente. La memoria de un Device se divide en 4 categorías:

- *Memoria Global.* Esta memoria cumple el rol de memoria principal en las placas de video, acá el programa Host escribe los datos a ser procesados mediante el uso de buffers. La memoria global contiene a la memoria constante. A pesar de que la memoria global es muy rápida, es deseable reducir la cantidad de accesos a la misma, ya que su latencia es elevada. Esta memoria no suele tener una cache asociada por hardware en las GPU. Sólo el programa Host puede reservar memoria global. Tanto el Kernel como el programa Host pueden leer y escribir en esta memoria.

- *Memoria Constante.* Esta memoria está contenida en la memoria global, y su tamaño es pequeño ($\approx$ 64kb). Se la utiliza para el pasaje de argumentos al Kernel y los valores de las variables inicializadas en tiempo de compilación. Esta memoria suele ser mas velóz que la memoria Global en caso de tener un hit en la cache (es común que se produzcan hits ya que la memoria constante es pequeña). El Host puede reservar, leer y escribir esta memoria. El Kernel puede solamente leer esta memoria.

- *Memoria Local.* Cada Compute Unit tiene su propia memoria Local, esta es compartida por todos los Processing Unit de un Compute Unit, por lo cual los accesos a la misma deben ser sincronizados correctamente en el código del Kernel. El uso incorrecto de la misma puede generar conflictos de bancos de memoria, para mas detalles de esto se debe ver la especificación de bancos de memoria de cada fabricante. Esta memoria puede ser reservada por el Host y por el compilador de OpenCL. El programa Host no tiene acceso de lectura/escritura a la misma.

- *Memoria Privada.* Cada Processing Unit tiene su propia memoria privada, esta también es llamada memoria automática, ya que acá son almacenadas las variables locales del Kernel. Esta memoria es muy rápida, y usualmente pequeña. Al ser local a cada Processing Unit no genera conflictos con

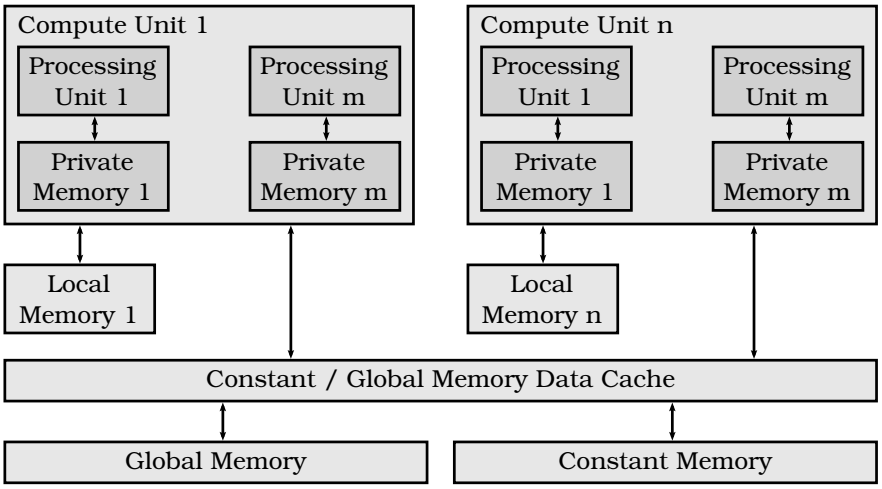

Figura 2: Modelo de memoria

la Cache. El programa Host no tiene ningún tipo de acceso a la memoria privada. El compilador de OpenCL puede reservar memoria privada.

2.3. Espacio de índices

Para resolver un problema en forma paralela, debemos tener un control de que parte del problema es resuelta por cada parte del hardware. Para esto OpenCL nos provee índices que permiten identificar la parte del hardware en la que se está ejecutando. Esto nos permite, desde el código Kernel, identificar que parte del problema se debe resolver. En el caso de las GPU, donde la paralelización del problema suele ser a nivel de datos, los índices suelen utilizarse para decidir a que fragmento del input acceder. Además de esto, a nivel código se define el tamaño del problema, el tamaño de grupo de trabajo (Work-Group) y la dimensión del problema (ND-Range), la versión OpenCL 1.1 permite que esta dimensión sea 1, 2 o 3.

Los índices pueden ser locales o globales.

- *Globales.* Este es el índice del Work-Item en el resto de todos los Work-Items disponibles. Se lo suele para identificar que parte del cómputo realizar. Notemos que el espacio global puede tener varias dimensiones, obteniendo el índice para todas las dimensiones, obtenemos las coordenadas de dicho Work-Item. Para obtener los índices se invoca `get_global_id(dimension)`. A su vez podemos obtener el tamaño del problema en alguna dimensión y la cantidad de dimensiones utilizando `get_global_size(dimension)` y `get_work_dim()` respectivamente.

- *Locales.* Este índice es local respecto del Work-Group en el que se encuentra el Work-Item. La cantidad total de Work-Items es dividida en Work-Groups (es necesario que en cada una de las dimensiones, el tamaño del problema sea divisible por el tamaño de los Work-Groups). Las ejecuciones pueden ser sincronizadas entre threads del mismo Work-Group, pero no es posible sincronizar entre diferentes Work-Groups entre sí. El tamaño de los Work-Groups impacta mucho en la performance, depende de este tamaño la cantidad de Work-Groups que pueda ejecutar un Compute-Unit, ya que este está acotado en memoria y cantidad de threads. Una buena regla práctica es definir el tamaño del Work-Group con múltiplos de 64. Podemos obtener el índice en una dimensión de un Work-Group con `get_local_id(dimension)`.

3. Desarrollo

En esta sección presentamos la descripción de todos los programas desarrollados para este trabajo, junto con comparaciones de pérformance entre la implementación en C++ utilizando un thread en el

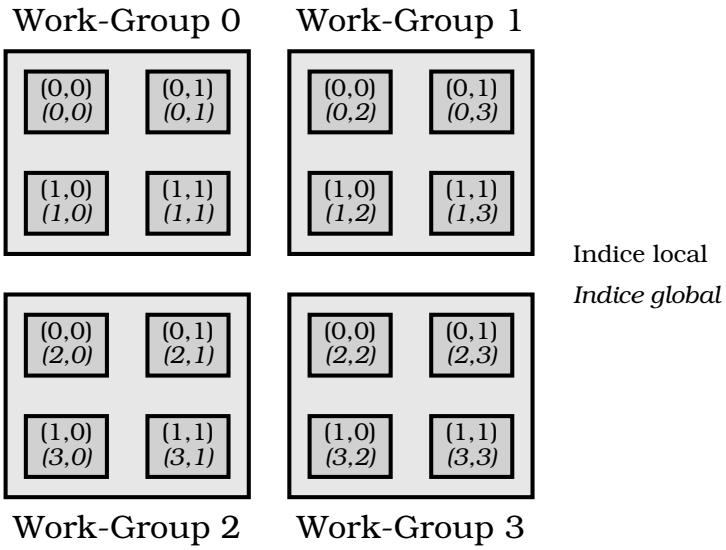

Figura 3: Espacio de índices de 16 elementos con ND = 2 y Work-Group size = 4

CPU, y la implementación hecha para GPU. El hardware utilizado es una CPU Intel Core i7 920, y una GPU ATI HD5770 Vapor-X. Las comparaciones fueron realizadas con algoritmos del mismo orden de complejidad para ambas plataformas. El tiempo de compilación on-line del Kernel y la inicialización de los dispositivos no es medido. Se mide el tiempo necesario en copiar la memoria a la placa de video, su ejecución y obtener el resultado.

3.1. Suma de Vectores (VectorAdd)

Este programa suma dos vectores de números en punto flotante haciendo uso de la GPU. Podemos decir que este programa es el equivalente al "hola mundo" en otros paradigmas. El kernel es el siguiente

```
1  __kernel void vectorAdd(
2    const __global float *a,
3    const __global float *b,
4    __global float *c,
5    const unsigned int n)
6  {
7    /* Es necesario el condicional, ya que el tamanio del
8      vector puede no ser multiplo de Work-Group-Size. Y el
9      tamanio del Problema debe ser divisible por Work-Group-Size. */
10
11   int gid = get_global_id(0);
12   if(gid < n)
13     c[gid] = a[gid] + b[gid];
14 }
```

El modificador `__kernel` indica que es una función que es el nombre de un kernel invocable desde el Host. El modificador `__global` indica que la variable se almacenará en memoria global del dispositivo, es buena práctica utilizar `__const` en los parámetros de entrada, en este caso `*a` y `*b`. Este Kernel se invoca en una configuración de problema unidimensional ($ND = 1$), del menor tamaño posible tal que sea divisible por `WORK_GROUP_SIZE` y sea mayor o igual que n. Este es el motivo del condicional del código.

La inicialización de la plataforma en el Host no se muestra en el informe.

3.2. Reduce

Este programa suma todos los elementos de un vector. El procedimiento consiste en crear un vector en memoria local de tamaño igual a la cantidad de Work-Items por Work-Group. Cada Work-Item acumulará sumas en la Local-Index-ésima entrada de dicho vector. En cada iteración se desplazarán los Work-Items hacia la derecha del vector global. Una vez que se recorrió todo el vector de entrada, se computa la suma de todo el vector y se almacena en el output. Esto da como resultado un vector de tamaño igual a la cantidad de Work-Groups, cuya suma es igual a la suma de todo el vector de entrada. La suma del vector local se realiza optimizando los accesos locales a memoria, aunque la ventaja frente a una implementación donde un Work-Item lo suma secuencialmente no es significativa (este resultado no se muestra en este trabajo). El código del Kernel es el siguiente

```
1  __kernel void reduce(
2    const __global float *in,
3    __global float *out,
4    const unsigned int n,
5    __local float *buffer)
6  {
7    int idl = get_local_id(0);
8    int stride = get_global_size(0);
9    buffer[idl] = 0;
10
11   for(int pos = get_global_id(0); pos < n; pos += stride){
12     buffer[idl] += in[pos];
13   }
14
15   barrier(CLK_LOCAL_MEM_FENCE); //sincronizo para leer memoria local
16
17   /* de esta forma solo el primer work-item del grupo trabaja
18       => se serializan los accesos a memoria
19   if(!idl){ //el resultado de esta implementacion no se muestra
20     for(size_t i = 0; i < get_local_size(0); i++)
21       sum += buffer[i];
22     out[get_group_id(0)] = sum;
23   }
24   */
25
26
27   /* Idea del esquema de bancos de memoria:
28
29   cada numero a continuacion representa una tira de 4 bytes:
30   16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31
31   0 1 2 3 4 5 6 7 8 9 10  11  12  13  14  15
32
33   b0  b1  b2  b3  b4  b5  b6  b7  b8  b9  b10 b11 b12 b13 b14 b15
34
35   La i-esima columna corresponde al i-esimo banco bi, la gestion
36   de memoria es posible gracias al uso de estos bancos. En caso
37   de todos los work-items acceder al mismo banco se gasta un
38   ciclo de clock (este es un caso especial); en caso contrario,
39   si dos work-items acceden al mismo banco, estos dos (o mas, pero
40   no 16) accesos se serializan, gastando asi mucho tiempo. La idea
41   es realizar la suma del buffer local en paralelo, sin serializar
42   ningun acceso a memoria innecesario. Si notamos el buffer
43   v1 v2 v3 .... vk donde k = get_local_size(0),
```

```
44    es decir el tamanio del work-group, podemos computar:
45
46    buffer <- v_1+v_k/2+1 v2_k/2+2 ..
47
48    es decir, si tenemos k elementos, en una iteracion sumamos
49    los ultimos k/2 elementos con los primeros k/2; y asi siguiendo.
50    De esta forma, como k es multiplo de 16, podemos computar la
51    suma de todos los elementos del buffer siempre accediendo a
52    bancos separados, aprovechando al maximo la velocidad de la
53    memoria local - que es igual de rapida que la privada.
54    */
55
56    // de esta forma se paraleliza la suma del buffer local,
57    // entre los work-item de un mismo work-group
58    for(int s = 128; s > 0; s = s >> 1){  // hay 16 bancos de memoria
59      if(idl < s) buffer[idl] += buffer[idl + s];
60      barrier(CLK_LOCAL_MEM_FENCE); //necesario por los branches de arriba
61    }
62    if(!idl) out[get_group_id(0)] = buffer[0];
63  }
```

En el código está brevemente explicado el funcionamiento de los bancos de memoria. La función `barrier` es una directiva de sincronización provista por OpenCL. Al invocar `barrier(CLK_LOCAL_MEM_FENCE)` se fuerza a que todos los hilos de ejecución lleguen a ese punto del programa antes de que alguno de ellos pueda seguir adelante. Esto es necesario para evitar leer memoria no inicializada o no actualizada. Sin esto no podemos garantizar la correcta ejecución del algoritmo.

Observemos que este código representa a la función $fold$ del paradigma funcional, de aquí el interés e importancia en tener una implementación de la misma.

3.3. Producto Interno Canónico (Dot Product)

La función $\langle ; \rangle : \mathbb{R}^n \to \mathbb{R}$ dada por $\langle (x_1, \ldots, x_n); (y_1, \ldots, y_n) \rangle = \sum_{i=1}^{n} x_i y_i$ es el producto interno canónico, para computarla hacemos uso de la función Reduce modificando levemente el código:

```
1   __kernel void dotProduct(
2     const __global float *a,
3     const __global float *b,
4     __global float *out,
5     const unsigned int n,
6     __local float *buffer)
7   {
8     int idl = get_local_id(0);
9     int stride = get_global_size(0);
10    buffer[idl] = 0;
11
12    for(int pos = get_global_id(0); pos < n; pos += stride){
13      buffer[idl] += a[pos]*b[pos];
14    }
15
16    barrier(CLK_LOCAL_MEM_FENCE);
17
18    for(int s = 128; s > 0; s = s >> 1){
19      if(idl < s) buffer[idl] += buffer[idl + s];
20      barrier(CLK_LOCAL_MEM_FENCE);
21    }
```

```
22    if(!idl) out[get_group_id(0)] = buffer[0];
23  }
```

3.4. Potencia de Matrices con dimensión potencia de 2 (MatrixMul)

Este programa realiza el producto $C := AB$ con $A, B \in \mathbb{R}^{2^k \times 2^k}$, $k \geq 4$. El producto se realiza por bloques de 16×16. Cada Work-Group computa un bloque de la matriz de salida, que será la suma de productos de bloques de A y bloques de B. Es decir si la partición en bloques de ambas matrices la escribimos con $A_{ij}, B_{ij} \in \mathbb{R}^{16\times 16}$.

$$A = \begin{pmatrix} A_{11} & \cdots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{n1} & \cdots & A_{nn} \end{pmatrix} \quad , \quad B = \begin{pmatrix} B_{11} & \cdots & B_{1n} \\ \vdots & \ddots & \vdots \\ B_{n1} & \cdots & B_{nn} \end{pmatrix}$$

entonces

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

El código del Kernel es el siguiente

```
1  /* A es de nxm y
2     B es de mxl. Por lo tanto AB es de nxl.
3     Se asumen n,m,l divisibles por BLOCK_SIZE.
4     n viene implicito por la cantidad de WorkGroups
5  */
6
7  #define BLOCK_SIZE 16
8
9  __kernel void matrixMul(
10   const __global float *A,
11   const __global float *B,
12   __global float *AB,
13   const unsigned int m,
14   const unsigned int l )
15 {
16   float ijAcum = 0;       //acumulador del producto
17   int bx = get_group_id(0); //indice x del bloque
18   int by = get_group_id(1); //indice y del bloque
19   int j  = get_local_id(0); //indice x dentro del bloque
20   int i  = get_local_id(1);   //indice y dentro del bloque
21
22   int aBegin = m * BLOCK_SIZE * by; //aca esta el (0,0) del bloque de A actual
23   int aEnd   = aBegin + m - 1;     //a lo sumo llega hasta aca
24                   //(aca hacemos uso de que BLOCK_SIZE divide a m,n,l
25   int aStep  = BLOCK_SIZE;      //salto del bloque A, hacer dibujito para ↩
       entender
26
27   int bBegin = BLOCK_SIZE * bx; //aca esta el (0,0) del bloque de B actual
28   int bStep = BLOCK_SIZE * l;   //salto del bloque B, hacer dibujito para ↩
       entender (es una L)
29
30   for(int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep){
31     //copiamos las submatrices a memoria local, cada W-Item copia su indice
32     __local float As[BLOCK_SIZE][BLOCK_SIZE];
33     __local float Bs[BLOCK_SIZE][BLOCK_SIZE];
```

```
34      As[i][j] = A[a + m*i + j];  //hacer dibujito para ver el indice
35      Bs[i][j] = B[b + l*i + j];  //idem
36
37      barrier(CLK_LOCAL_MEM_FENCE); //esperamos a que las submatrices se carguen
38      for(int k = 0; k < BLOCK_SIZE; k++) ijAcum += As[i][k] * Bs[k][j];
39      barrier(CLK_LOCAL_MEM_FENCE); //esperamos asi nadie pisa las matrices As y↩
            Bs
40    }
41    int c = l * BLOCK_SIZE * by + BLOCK_SIZE * bx;  //el (0,0) del bloque bx by
42    AB[c + l*i+j] = ijAcum;
43  }
```

3.5. Convolución en imágenes (VideoConvolution)

Este programa permite cómputos de convolución, estos son utilizados en filtros de imágenes, por ejemplo en detección de bordes o filtro gaussiano. Esta implementación utiliza una matriz Kernel de 3×3 que está fijada en el código Kernel, esto se podría parametrizar, pero optamos por esta implementación. La idea del algoritmo consiste en copiar primero partes de la imagen entera a memoria local del Work-Group, realizar los cómputos con el Kernel y una vez que se tienen los resultados para todos los pixels que corresponda, guardar la imagen en el buffer de salida. El algoritmo está hecho para imágenes de 3 canales con valores codificados con `unsigned char`. Del lado del Host se utilizó la librería OpenCV 2.3.1, de ahí el formato utilizado del lado de la GPU. El borde de un pixel de la imagen de salida no está definido por comodidad. Notemos que al copiar parte de la imagen original a memoria local, necesitamos un poco mas de información que la que corresponde a un mapeo directo del Work-Group a la imagen entera, esto es, hace falta cargar los bordes "externos" de 1px de ancho, sino no se tiene información válida para computar los bordes de cada fracción de imagen asociada a cada Work-Group. Esto se logra con un pequeño shift en los índices de la memoria local: lo que corresponde al i-ésimo elemento se escribe con $i + 1$, lo que corresponde al $i - 1$-ésimo elemento se escribe con i, etc. (esto vale tanto para columnas como para filas). Además debemos tener cuidado de no acceder memoria fuera del buffer que obtenemos como parámetro. La motivación de esta implementación es permitir realizar filtros basados en convolución en videos HD y FullHD sin sacrificar muchos fps. El código del Kernel es el siguiente:

```
1  #define BLOCK_SIZE 16 //debe coincidir con el BLOCK_SIZE del cpp
2  #define CHANNELS 3
3  #define M00 -1
4  #define M01 -1
5  #define M02 -1
6  #define M10 -1
7  #define M11 8
8  #define M12 -1
9  #define M20 -1
10 #define M21 -1
11 #define M22 -1
12
13
14 __kernel void videoConvolution(
15   const __global unsigned char *img,
16   __global unsigned char *out,
17   const unsigned int width,
18   const unsigned int height,
19   const unsigned int step)
20 {
21
```

```
22    int ijAcum;
23    int j  = get_local_id(0); //indice x dentro del bloque
24    int i  = get_local_id(1); //indice y dentro del bloque
25
26    int gidj  = get_global_id(0); //indice x global
27    int gidi  = get_global_id(1); //indice y global
28
29    if( gidj < width &&
30      gidi < height ) //procesamos dentro de la imagen
31    {
32      //el tamanio es del bloque y los bordes
33      __local unsigned char M[(BLOCK_SIZE*BLOCK_SIZE+4*BLOCK_SIZE+4)*CHANNELS];
34      int stepM = (BLOCK_SIZE+2)*CHANNELS;
35
36      //el i+1 hace rol del central, el i del elem de arriba e i+2 del de abajo
37      M[(i+1)*stepM+(j+1)*CHANNELS+0] = img[gidi*step+gidj*CHANNELS+0];
38      M[(i+1)*stepM+(j+1)*CHANNELS+1] = img[gidi*step+gidj*CHANNELS+1];
39      M[(i+1)*stepM+(j+1)*CHANNELS+2] = img[gidi*step+gidj*CHANNELS+2];
40
41      //copia los bordes
42      barrier(CLK_LOCAL_MEM_FENCE);
43      if(i == 0 && gidi != 0) {
44        M[(i)*stepM+(j+1)*CHANNELS+0] = img[(gidi-1)*step+gidj*CHANNELS+0];
45        M[(i)*stepM+(j+1)*CHANNELS+1] = img[(gidi-1)*step+gidj*CHANNELS+1];
46        M[(i)*stepM+(j+1)*CHANNELS+2] = img[(gidi-1)*step+gidj*CHANNELS+2];
47      }
48      if(i == BLOCK_SIZE-1 && gidi != height-1) {
49        M[(i+2)*stepM+(j+1)*CHANNELS+0] = img[(gidi+1)*step+gidj*CHANNELS+0];
50        M[(i+2)*stepM+(j+1)*CHANNELS+1] = img[(gidi+1)*step+gidj*CHANNELS+1];
51        M[(i+2)*stepM+(j+1)*CHANNELS+2] = img[(gidi+1)*step+gidj*CHANNELS+2];
52      }
53      if(j == 0 && gidj != 0) {
54        M[(i+1)*stepM+(j)*CHANNELS+0] = img[(gidi)*step+(gidj-1)*CHANNELS+0];
55        M[(i+1)*stepM+(j)*CHANNELS+1] = img[(gidi)*step+(gidj-1)*CHANNELS+1];
56        M[(i+1)*stepM+(j)*CHANNELS+2] = img[(gidi)*step+(gidj-1)*CHANNELS+2];
57      }
58      if(j == BLOCK_SIZE-1 && gidj != width-1) {
59        M[(i+1)*stepM+(j+2)*CHANNELS+0] = img[(gidi)*step+(gidj+1)*CHANNELS+0];
60        M[(i+1)*stepM+(j+2)*CHANNELS+1] = img[(gidi)*step+(gidj+1)*CHANNELS+1];
61        M[(i+1)*stepM+(j+2)*CHANNELS+2] = img[(gidi)*step+(gidj+1)*CHANNELS+2];
62      }
63      if(i == 0 && j == 0 && gidj != 0 && gidi != 0) {
64        M[(i)*stepM+(j)*CHANNELS+0] = img[(gidi-1)*step+(gidj-1)*CHANNELS+0];
65        M[(i)*stepM+(j)*CHANNELS+1] = img[(gidi-1)*step+(gidj-1)*CHANNELS+1];
66        M[(i)*stepM+(j)*CHANNELS+2] = img[(gidi-1)*step+(gidj-1)*CHANNELS+2];
67      }
68      if(i == 0 && j == BLOCK_SIZE-1 && gidj != width-1 && gidi != 0) {
69        M[(i)*stepM+(j+2)*CHANNELS+0] = img[(gidi-1)*step+(gidj+1)*CHANNELS+0];
70        M[(i)*stepM+(j+2)*CHANNELS+1] = img[(gidi-1)*step+(gidj+1)*CHANNELS+1];
71        M[(i)*stepM+(j+2)*CHANNELS+2] = img[(gidi-1)*step+(gidj+1)*CHANNELS+2];
72      }
73      if(i == BLOCK_SIZE-1 && j == BLOCK_SIZE-1 && gidj != width-1 && gidi !=
          height-1) {
74        M[(i+2)*stepM+(j+2)*CHANNELS+0] = img[(gidi+1)*step+(gidj+1)*CHANNELS
            +0];
```

```
75        M[(i+2)*stepM+(j+2)*CHANNELS+1] = img[(gidi+1)*step+(gidj+1)*CHANNELS↩
              +1];
76        M[(i+2)*stepM+(j+2)*CHANNELS+2] = img[(gidi+1)*step+(gidj+1)*CHANNELS↩
              +2];
77      }
78      if(i == BLOCK_SIZE-1 && j == 0 && gidj != 0 && gidi != height-1) {
79        M[(i+2)*stepM+(j)*CHANNELS+0] = img[(gidi+1)*step+(gidj-1)*CHANNELS+0];
80        M[(i+2)*stepM+(j)*CHANNELS+1] = img[(gidi+1)*step+(gidj-1)*CHANNELS+1];
81        M[(i+2)*stepM+(j)*CHANNELS+2] = img[(gidi+1)*step+(gidj-1)*CHANNELS+2];
82      }
83
84      barrier(CLK_LOCAL_MEM_FENCE); // aseguramos que estan todos los datos
85      for(int k = 0; k < CHANNELS; k++){
86        ijAcum = M[(i)*stepM  + (j)*CHANNELS+k] * M00;
87        ijAcum += M[(i)*stepM + (j+1)*CHANNELS+k] * M01;
88        ijAcum += M[(i)*stepM + (j+2)*CHANNELS+k] * M02;
89        ijAcum += M[(i+1)*stepM + (j)*CHANNELS+k] * M10;
90        ijAcum += M[(i+1)*stepM + (j+1)*CHANNELS+k] * M11;
91        ijAcum += M[(i+1)*stepM + (j+2)*CHANNELS+k] * M12;
92        ijAcum += M[(i+2)*stepM + (j)*CHANNELS+k] * M20;
93        ijAcum += M[(i+2)*stepM + (j+1)*CHANNELS+k] * M21;
94        ijAcum += M[(i+2)*stepM + (j+2)*CHANNELS+k] * M22;
95        if(ijAcum < 0) ijAcum = 0;     //saturamos
96        else if(ijAcum >= 255) ijAcum = 255;
97        out[gidi*step+gidj*CHANNELS+k] = ijAcum;
98      }
99    }
100 }
```

3.6. Filtro de Prewitt

Este programa aplica el filtro de detección de bordes de Prewitt. El código es una leve modificación del algoritmo de Convolución, por lo que no lo mostramos en el informe (ver adjunto en `./src/Prewitt/Prewitt.cl` por detalles de la modificación).

3.7. Filtro de la Mediana (MedianFilter)

Este programa aplica el filtro de la mediana a un video. Este filtro consiste en asignar, por cada canal, a cada pixel la mediana de el y su vecindario inmediato, es decir, la mediana entre él y los ocho pixels que lo rodean. Esto permite mejorar la calidad de imágenes dañadas con algunos tipos particulares de ruido. Para poder ver esto en acción hemos adjuntado un programa que agrega ruido a un video y lo guarda en formato avi, en `./src/Noiser/`. Para obtener la mediana se utilizó un selection sort en caso del GPU, y en caso de la CPU se utilizó la función `Vector::sort` de la STL de C++. El código de este programa es muy similar los dos anteriores, por lo que tampoco se muestra en el informe. Ver `./src/MedianFilter/MedianFilter.cl`.

3.8. Filtro Emboss

Este programa aplica el filtro Emboss a un video. El método no convierte la imagen a blanco y negro previamente, y se basa en un algoritmo de convolución.

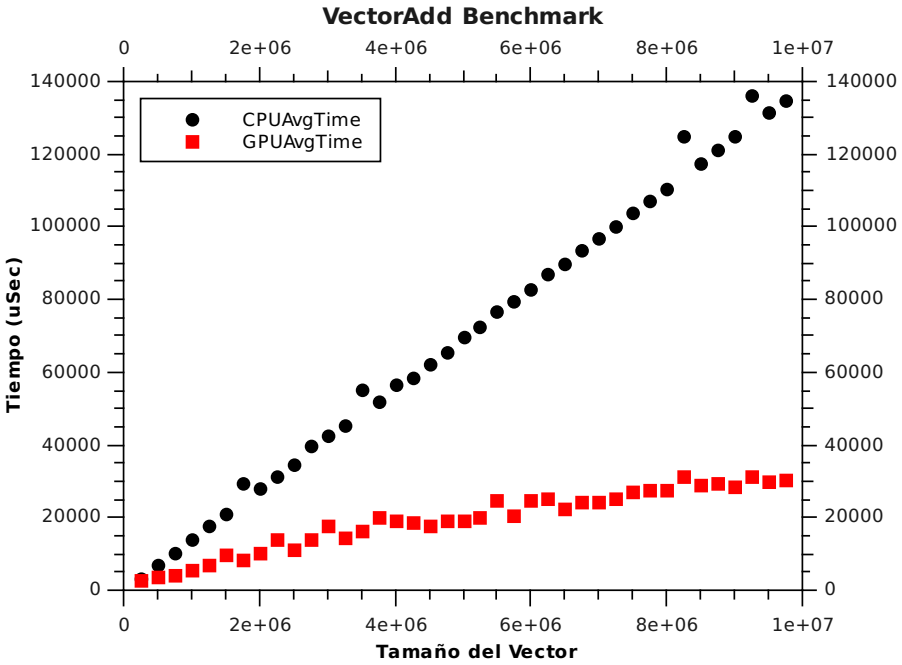

Figura 4: Comparación de tiempo medio de corrida para VectorAdd

3.9. Equalización de Histograma (EqualizeHist)

Este programa equaliza un histograma de un video cuadro a cuadro. La utilidad de este algoritmo es lograr que la distribución de las intensidades de los pixeles sea mas uniforme. Presentamos solamente la implementación en CPU, por lo cual no mostraremos resultados de performance. El método previamente convierte a blanco y negro la imagen de entrada.

3.10. Filtro de Erosión (Erode)

Este programa convierte a blanco y negro un video, filtra los pixeles según un umbral que se puede variar con las teclas `up` y `down`, poniendo en blanco los que superen el umbral y en negro los que no. Luego de esto aplica una iteración de erosión a la imagen. Es fácil extender el algoritmo para aplicar tantas iteraciones de erosión como se desee. La erosión implementada se basa en convolución.

3.11. Filtro Sharpen

Este programa aplica el filtro Sharpen a un video, nuevamente la implementación se basa en una convolución.

3.12. Agregar ruido a un video (Noiser)

Este programa genera ruido basándose en un método aleatorio a un video, como es un programa auxiliar no se ha implementado en GPU. Sirve para generar inputs interesantes para el filtro de la Mediana.

4. Resultados

En esta sección presentamos los resultados de rendimiento que hemos obtenido.

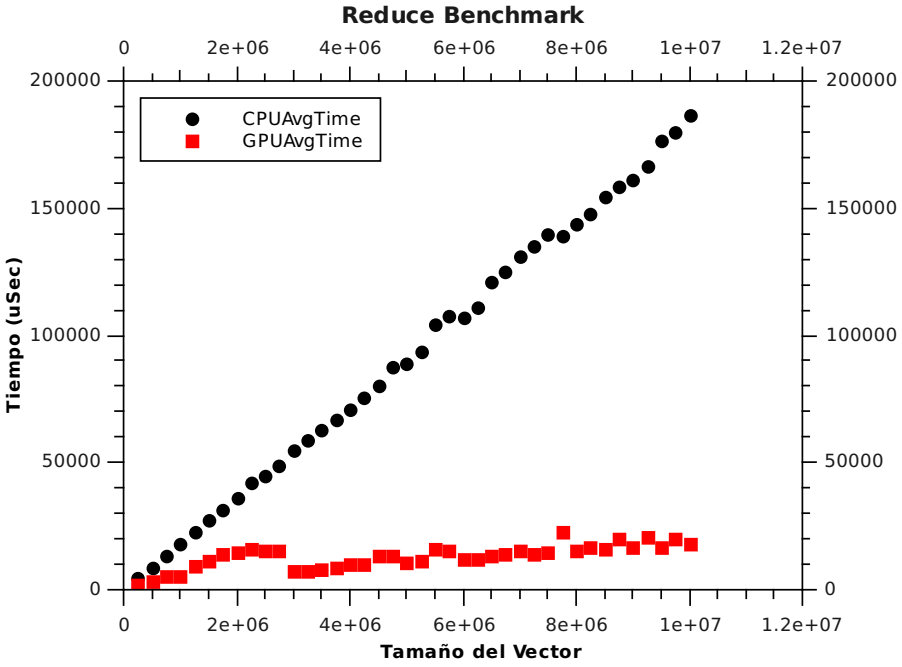

Figura 5: Comparación de tiempo medio de corrida para Reduce

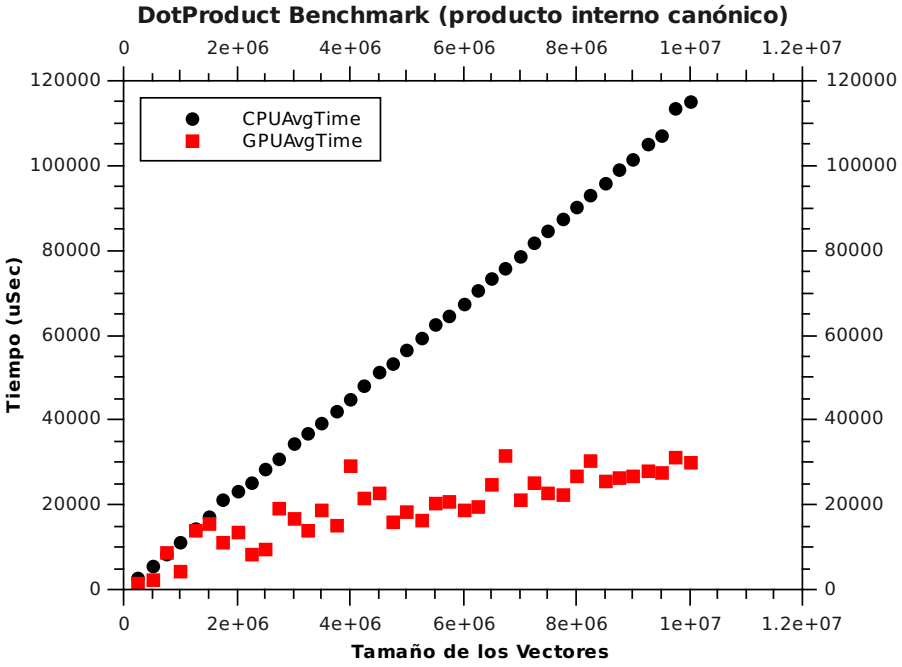

Figura 6: Comparación de tiempo medio de corrida para DotProduct

n	Avg CPU	Avg GPU
16	26	6097.4
32	201	4877.5
64	1692	7343.3
128	16757	11869.9
256	129423	23273.6
512	1183114	44045.3
1024	9227893	89253.9
2048	193651890.5	205957.4

Cuadro 1: Comparación de tiempo medio de corrida para MatrixMul, donde $n \times n$ es la dimensión de la matriz.

CPU fps	GPU fps
5.07143	19.2857
5.09524	19.3571
5.11905	19.4286
5.14286	19.5
5.16667	19.5714
5.06977	19.6429

Cuadro 2: Comparación de cuadros por segundo (fps) en un video 720p para Convolución de Imágenes (VideoConvolution)

CPU fps	GPU fps
4.37037	18.7
4.38889	18.75
4.40741	18.8
4.36364	18.85
4.38182	18.9
4.34545	18.95

Cuadro 3: Comparación de cuadros por segundo (fps) en un video 720p para el filtro Prewitt

CPU fps	GPU fps
0.469565	12.5789
0.470085	12.6316
0.470588	12.6842
0.471074	12.7368
0.471545	12.7895

Cuadro 4: Comparación de cuadros por segundo (fps) en un video 720p para el filtro de la mediana (MedianFilter)

5. Discusión

5.1. Suma de Vectores (VectorAdd)

Observamos que en todos los casos la suma de vectores se realiza mas rápido en la GPU, a pesar de ser una operación muy sencilla y velóz de realizar en CPU. Es importante notar que hemos medido el tiempo de copia del buffer desde el Host hacia el Device, pero esto no se manifiesta de forma significativa en los resultados obtenidos (ver Figura 4). Entendemos que esto se debe al buen manejo de memoria que brinda la implementación de AMD, y la posibilidad de copiar gran cantidad de datos mediante PCI-e. Algo que se puede observar es la irregularidad en los tiempos medios de la GPU, estos pueden atribuirse a la carga del sistema a la hora de realizar mediciones. Notamos también que el algoritmo sigue comportándose de forma lineal, lo cual es un resultado esperado.

5.2. Reduce

Se puede observar en las Figuras (5) y (4) que el Reduce muestra mejor respuesta en términos de tiempo que VectorAdd. Esto suponemos que se debe al manejo mas inteligente de la memoria local, explicado brevemente en el código del Kernel de Reduce. Se observa una seria mejora respecto del tiempo que insume el reduce en CPU.

5.3. Producto de Matrices con dimensión potencia de 2 (MatrixMul)

Este es el primer algoritmo de los implementados que no tiene orden lineal, y esto se refleja fuertemente en los resultados que obtuvimos (ver Cuadro 1). El cuadro presentado muestra la media para GPU con 100 corridas, pero para la CPU se hicieron sólo 2 corridas para hallar un estimador mas débil de la media, el motivo de esto es la muy baja performance que obtuvimos para la implementación en CPU. A pesar de no ser una media precisa, refleja la gran diferencia entre ambos enfoques, lo que cumple nuestros objetivos. Numéricamente hemos obtenido un speedup de aproximadamente $\times 900$ en el caso de $n = 2048$ y aproximadamente $\times 100$ para el caso $n = 1024$. Este resultado sugiere la gran utilidad de los procesadores GPU para cálculo científico.

5.4. Convolución en imágenes (VideoConvolution)

Observamos que al aplicar convolución con nuestra implementación estamos relativamente cerca de tener un framerate de video (25fps) (ver Cuadro 2). Quizás se pueda obtener este framerate procesando los frames de video no en forma secuencial, sino de forma asincrónica. Esto introduce mayor complejidad en en el código del Host, y probablemente deba utilizar mas de un thread en el Host, que no es nuestro objetivo, por lo cual no lo hemos implementado. De todos modos obtenemos una imagen mucho mas fluida.

5.5. Prewitt

Este filtro, por la cantidad de operaciones que ejecuta, es muy parecido a una convolución, pero hace algunas cuentas mas. Es razonable que estas cuentas impacten en aproximadamente un fps en ambas plataformas (ver Cuadros 3 2).

5.6. Filtro de la mediana (MedianFilter)

Este filtro por cada pixel debe ordenar un vector de 9 elementos, lo cual impacta mucho en el rendimiento, esto se observa tanto en el CPU como en GPU. La implementación es similar a los anteriores en cuanto al manejo de los datos internamente. Observamos que con la GPU podemos ver una serie de imágenes razonablemente bien, pero con el CPU vemos solo un frame cada dos segundos (ver Cuadro 4).

6. Conclusiones

Hoy día el poder de cómputo que ofrecen las placas de video es muy atractivo, hay una gran variedad de problemas que tienen naturaleza de ser "paralelos a nivel de datos", que suelen ser bastante demandantes en términos de tiempo para los procesadores convencionales, cuestión que, según vimos, las GPU mejoran bastante. A pesar de que no hemos explorado paralelizar en varios cores de CPU o en varios CPU en este trabajo, parece ser una buena alternativa programar dichos problemas en placas de video en vez de paralelizar como se hacía hasta hace varios años. Además de mejorar la velocidad reducimos la carga de los CPU permitiendo trabajar en tareas menos paralelizables a nivel de datos. No parece haber motivo de no aprender a programar en CUDA u OpenCL hoy dia.

7. Entregable

Por cada uno de los programas descritos en la sección de Desarrollo, hay una carpeta con el nombre correspondiente (o el indicado entre paréntesis a lo largo del informe) que contiene una implementación en C++ del programa, una implementación en C++ del Host y un Kernel asociado al Host. Además hay un Makefile por programa. Para compilar se requiere una instalación de los drivers ATI Catalyst 11.9 que soporte OpenCL 1.1, una implementación de AMD del estándar OpenCL 1.1 (instalados siguiendo la documentación de AMD), y, en el caso de los programas que manipulen imágenes se requiere OpenCV 2.3.1. En cada una de las carpetas de los programas enumerados hay dos archivos que son los que se usaron para generar los resultados (`runcpu.txt` y `rungpu.txt`). Además de esto hay un `Makefile` en `./src/` que compila todos los programas y la biblioteca `./src/lib` que contiene utilidades que se usaron a lo largo del trabajo.

Nota: Probablemente todas las condiciones para compilar no se cumplan en la mayoría de las máquinas, por lo cual sería deseable ver algún modo de mostrar el trabajo andando, una forma sería filmar una captura de las corridas, al menos las que tengan imágenes, y mostrar dicha grabación. Otra forma es utilizar remote desktop.