Capítulo 3

Aplicación Vivid

3.1. Implementacion paralela basada en Flow Graph

Una vez ya planteado el algoritmo serie ViVid, el siguiente paso es realizar su paralelización mediante Intel TBB y más concretamente el módulo OpenCL para la parte de GPU. El objetivo que se pretende con esta paralelización es conseguir la mayor mejora posible de rendimiento en la ejecución del algoritmo cuando se vayan a procesar muchos frames. Para ello se pretende, no sólo paralelizar el algoritmo para implementarlo en CPU, sino sacarle todo el partido a la capacidad que ofrece la arquitectura heterogénea sobre la que se trabaja, implementando también el algoritmo en la GPU integrada. De esta forma se pueden ejecutar paralelamente frames en la CPU (con su paralelización para los distintos cores) y la GPU, consiguiendo así el mejor rendimiento posible en esta arquitectura.

La anterior implementación paralela de ViVid, hacía uso de la estructura Pipeline, la cual permitía implementar de forma efectiva las 3 etapas del filtro ViVid, como si de una cadena de montaje se tratase, donde van pasando por ella los item (o frames) y se van procesando en cada etapa. Sin embargo, la implementación de la parte GPU en esta estructura era muy complicada. El nuevo módulo OpenCL creado para Flow Graph simplifica muchisimo la implementación de la GPU en algoritmos paralelos que se creen mediante Flow Graph.

Surge así el motivo por el que se realiza este proyecto: partir de la implementación serie de ViVid y paralelizarlo pero empleando esta vez la estructura Flow graph y el nuevo módulo OpenCL, con el fin de analizar su efectividad, facilidad de implementación y rendimiento.

3.1.1. Estructura del grafo

A continuación se va a explicar el funcionamiento del Grafo a través del cual se ha llevado a cabo la implementación de ViVid paralelizado. En la Figura 3.1 se puede apreciar la estructura general del grafo el cual se explicará de forma general su funcionamiento, para luego entrar en detalle en la programación de cada uno de los nodos que lo componen junto con su código correspondiente.

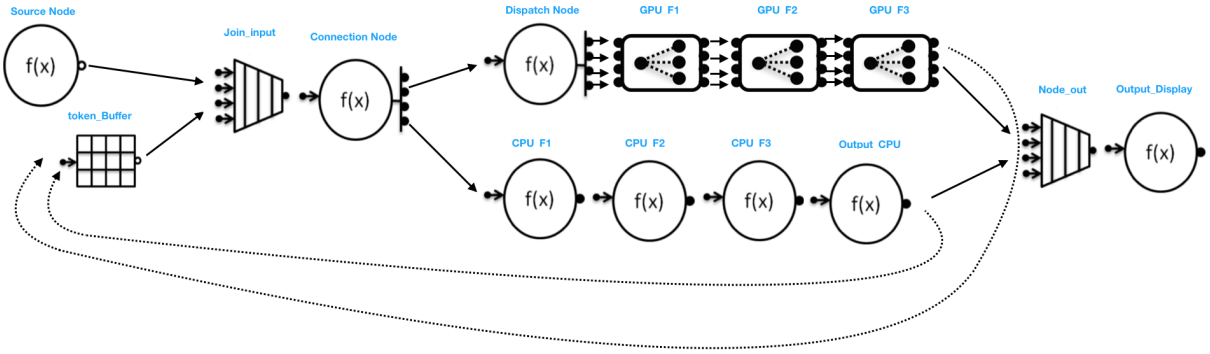


Figura 3.1: Flow Graph de Vivid paralelizado.

El grafo está diseñado como una estructura de funcionamiento basada en "tokens". Un "token" representa un recurso libre del sistema. De esta forma se va a poder controlar el paralelismo a nivel de CPU (con los distintos cores disponibles), así como la GPU. los "tokens" son las unidades de recursos disponibles que permitirán al sistema poder emplear diversos cores de CPU, asi como tener disponible o no la GPU para aumentar el rendimiento. Por lo tanto, el programa comienza generando los "tokens" que se quiere tener disponibles (CPU y/o GPU), comenzando el grafo a través de su nodo fuente, el source_node.

Éste, si tiene un "token" disponible a través del token_buffer, genera un identificador de imagen (esto servirá para poder tener ordenadas las imágenes que se generen, independientemente de si se ejecuten en CPU o GPU, al acabar el programa) y pasa junto con el token correspondiente al siguiente nodo. Éste nodo ()connection_node) se encarga de, en función del token que acompañe al identificador de la imagen, trasladar ambos a una de las dos bifurcaciones del grafo: CPU o GPU.

En ambas ramificaciones del grafo se sigue el mismo funcionamiento, cambiando el hardware sobre el que se lleva a cabo. En primer lugar se toma una imagen y se generan las variables necesarias para la primera etapa del filtro ViVid, pasando los argumentos al mismo. Se sigue el mismo procedimiento con las siguientes etapas del algoritmo ViVid (ya explicado) hasta completar las 3 etapas.

Una vez analizada la imagen tanto en CPU como en GPU, se libera el "token" correspondiente, devolviéndolo al token_buffer. De esta forma queda libre un recurso del sistema, listo para que pueda cogerse de nuevo y analizar otra imagen. Las imagenes analizadas pasan a un nodo de salida, el cual permite, gracias al identificador que las ha acompañado desde el comienzo, realizar cualquier tipo de procesamiento que se quiera con ellas: ordenarlas si fueran frames de vídeo, analizar resultados, etc.

Como se puede ver, esta implementación permite un flujo circular de los recursos del sistema, de forma que mientras queden imágenes que analizar, todos los recursos que se hayan querido emplear están en constante funcionamiento con la máxima eficiencia posible: cada thread (o recurso del sistema) analizando una imagen simultáneamente. Con esta implementación es donde se consigue el alto grado de paralelismo. La implementación serie sólo permitía analizar una imagen al mismo tiempo, desde el comienzo hasta el final del algoritmo. Con la implementación basada en tokens y gracias a la potencia de Flow Graph se pueden estar ejecutando tantas imagenes simultáneamente como recursos disponibles tenga el sistema. Además, con el nuevo nodo de OpenCL, se permite añadir la enorme potencia que aporta la GPU al rendimiento de la aplicación, como se verá en el siguiente capítulo de resultados, lo cual aumenta considerablemente el paralelismo del sistema.

3.1.2. Nodos empleados

En el capítulo anterior se explicó que la estructura de Flow Graph se basa en la implementación de un grafo con nodos conectados entre sí, cada uno con una funcionalidad determinada. Por lo tanto, se van a explicar cuales han sido los nodos empleados, con la funcionalidad implementada en cada uno de ellos.

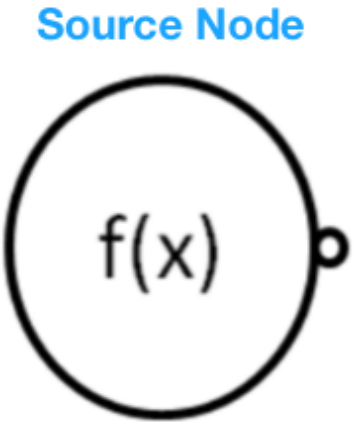


Figura 3.2: Source Node.

Nodo con el que comienza el programa. No se encarga de leer la imagen, pues sino leería la imagen tantas veces como imagenes se quieran ejecutar en el programa, con la pérdida de rendimiento que eso conlleva. La imagen se lee antes una sola vez, este nodo se encarga de ir generando tantos identificadores de imagen como imágenes se quieran procesar y pasarlas al siguiente nodo. Una vez que se hayan generado todos los identificadores de imágenes, el nodo dejará de funcionar, con lo cual en cuanto los siguientes nodos acaben de procesar las últimas imágenes el grafo se parará y acabará el programa.

```
/**************************
SOURCE NODE
**************************/
    source_node<int> input_node(g,[&](int &a)->bool
    {
        if(ID_imagen<NUM_IMAGES)
        {
            a =ID_imagen;
            ID_imagen++;
            return true;
        } else
          return false;

    },false);
```

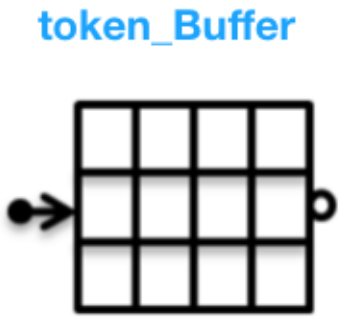


Figura 3.3: Token Buffer.

El token_buffer se encarga de almacenar y trasladar los tokens desde su generación al comienzo del programa hasta el siguiente nodo, join_node. Conecta con este nodo por un lado y por otro con los finales de las dos ramificaciones del grafo: los caminos CPU y GPU. De esta forma se genera el grafo circular, cuando un token acaba una de las dos ejecuciones, vuelve al buffer, para unirse a un nuevo identificador de imagen y comenzar de nuevo el procesamiento de una nueva imagen, mientras haya disponibles.

```
    buffer_node<int> token_buffer(g);
```

Figura 3.4: Join Input.

El join_input se encarga de unir los items procedentes del source_node y del token_buffer. Es decir, agrupa en una tupla de datos el identificador de imagen procedente del source_node y el token procedente del token_buffer. El camino que el identificador tomará dependerá de si el token es de GPU o de CPU. Este nodo pasa dicha tupla al siguiente nodo, connection_node. Este tipo de nodo se configura como "reserving", lo cual es diferente a los otros join_node usados en el grafo. Esta particularidad significa que, el nodo no dará una salida hasta que haya una entra de cada tipo, sólo entonces agrupará todos los input en una tupla y generará una salida. La diferencia con al política predefinida de "queuing" es que ésta almacena los input en una cola, a la espera de que haya uno de cada tipo para generar una tupla, el modo "reserving" sólo los marca como disponibles, pero sin almacenarlos hasta que no haya para generar una tupla de salida. De esta manera sólo se consume un ítem de imagen cuando haya un token disponible para ella.

```
typedef join_node<tuple<int,int>, reserving > join_t;
join_t join_input(g);
```

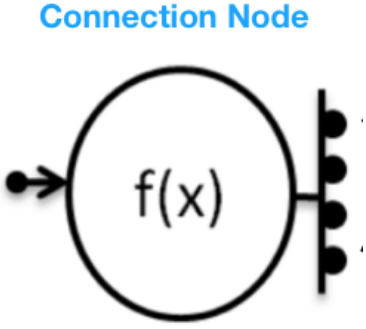


Figura 3.5: Connection Node.

El connection_node se encarga de recoger la tupla procedente del join_input y analizarla. Si el token que ha recibido es un token GPU, envía el identificador a la ramificación GPU. Si por el contrario es un token CPU, envía el identificador a la ramificación CPU. De esta forma se activan cada uno de dichos caminos y comienza el procesamiento de la imagen, bien sea en la CPU o en la GPU, en los siguientes nodos.

```
/*****************************
Connection_node. This node creates a path for the CPU and other for the GPU
    .
*****************************/
    typedef multifunction_node<join_t::output_type, tuple<int,int>> mfn_t1
        ;
    mfn_t1 connection_node(g, unlimited, [&](const join_t::output_type &in,
        mfn_t1::output_ports_type &ports ) {

      if (get<1>(in) == 1)
```

```
    {
    //Go to the CPU, to CPU_F1.
        get<0>(ports).try_put(get<0>(in));
    } else
      {
    //Go to the GPU, to dispatch_node.
        get<1>(ports).try_put(get<0>(in));
      }
  });
```

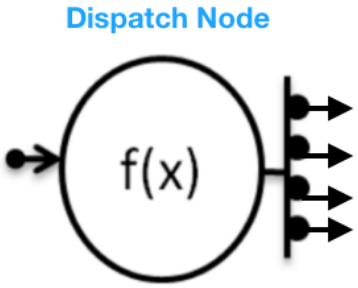


Figura 3.6: Dispatch Node.

Este nodo se activa si ha recibido un token correspondiente desde el connection_node. Una vez activado, se encarga de comenzar el camino para analizar una imagen en la GPU. Para poder usar variables tipo array en los nodos de OpenCL, se debe usar un tipo de dato especial: "opencl_buffer". Este tipo de dato, creado para OpenCL y su implementación en los flowgraph, se encarga de abstraer al programador de toda la árdua tarea de crear el entorno necesario para poder transferir variables de la CPU a la GPU y ejecutarlas en un programa OpenCL. Esta era una de las complicaciones existentes en la anterior implementación de ViVid, la cual con este nuevo módulo se facilita enormemente la labor de programación.
Las variables se crean en el main (para que sean accesibles por el "opencl_node".)Una vez creadas las variables necesarias para realizar el procesamiento en las distintas etapas GPU, se lanzan hacia el primer filtro y así comienza a ejecutarse el análisis de la imagen por parte de la GPU.

```
/*****************************
DISPATCH_NODE. This node starts the stage 1 for GPU and begins the GPU path
    .
*****************************/
    typedef multifunction_node<int, tuple<buffer_float,int,int,int,
        buffer_float,buffer_float,int,buffer_float,int,int>> mfn_t;
    mfn_t dispatch_node(g, unlimited, [&](int ID_imagen, mfn_t::
        output_ports_type &ports ) {

       TGPU = tbb::tick_count::now();
       get<0>(ports).try_put(opencl_imagen);
       get<1>(ports).try_put(width);
       get<2>(ports).try_put(height);
       get<3>(ports).try_put(pitch);
       get<4>(ports).try_put(opencl_ind);
```

```
        get<5>(ports).try_put(opencl_val);
        get<6>(ports).try_put(pitch);
        get<7>(ports).try_put(opencl_FilterBank);
        get<8>(ports).try_put(num_filters);
        get<9>(ports).try_put(ID_imagen);

    });
```

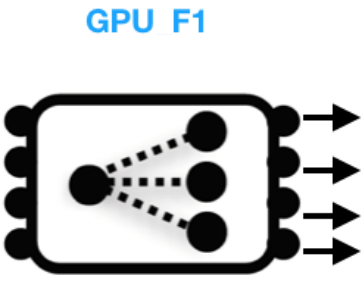


Figura 3.7: GPU_F1.

Esta etapa se encarga de realizar el primer filtro de las etapas GPU. Para ello lanza las variables que componen los argumentos de entrada de la primera etapa GPU al nodo OpenCL asociado, al cual se le ha cargado el código correspondiente a la primera etapa. Así se hace con las otras 2 etapas también. Su código se puede encontrar en los anexos, en la parte de código GPU. Realiza el mismo cálculo que la primera etapa serie de ViVid, pero adaptado para la GPU bajo código OpenCL. Además, se añade el código de su etapa CPU de salida, la cual se añade para poder recoger los argumentos de salida de la primera etapa GPU. Se encarga de lanzar los argumentos de entrada a la segunda etapa GPU del filtro.

Aparte de estos dos nodos, se va a mostrar el código que corresponde a la creacion de los 3 nodos OpenCL, la carga de los códigos correspondientes a cada etapa, y la definición de los rangos de cálculo, para optimizar el paralelismo en GPU.

```
//Graph and OpenCL nodes definition
    opencl_graph g;
    opencl_program<> program(g, "Filtros_GPU_copy.cl");
    opencl_node<buffer_datosGPU1> GPU_F1(g, program.get_kernel("
        blockwise_distance_kernel"));
    opencl_node<buffer_datosGPU2> GPU_F2(g, program.get_kernel("
        cellHistogramKernel3"));
    opencl_node<buffer_datosGPU3> GPU_F3(g, program.get_kernel("
        pairwiseDistanceKernel"));
    std::array<unsigned int, 2> range1{412-2,600-2};
    std::array<unsigned int, 2> range2{(412-2)/8,(600-2)/8};
    std::array<unsigned int, 2> range3{(412-2)/8,(600-2)/8};
    GPU_F1.set_range(range1);
    GPU_F2.set_range(range2);
    GPU_F3.set_range(range3);
```

```
/*******************************
*OUTPUT_GPU1
******************************* */
    typedef multifunction_node<join1_t::output_type, buffer_datosGPU2,
        reserving > mfn1_t;
    mfn1_t output_gpu1(g, unlimited, [&](const join1_t::output_type &m,
        mfn1_t::output_ports_type &ports ) {
       TGPU1 = tbb::tick_count::now();
       //std::cout <<"GPU 1= "<< (TGPU1 - TGPU).seconds() <<" segundos"<<
           std::endl;
       get<0>(ports).try_put(opencl_his);
       get<1>(ports).try_put(pitch);
       get<2>(ports).try_put(opencl_ind);
       get<3>(ports).try_put(pitch);
       get<4>(ports).try_put(opencl_val);
       get<5>(ports).try_put(pitch);
       get<6>(ports).try_put(num_filters);
       get<7>(ports).try_put(cell_size);
       get<8>(ports).try_put(n_parts_x_gpu);
    });
```

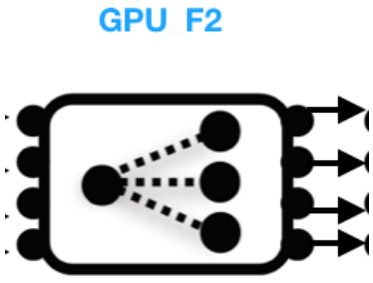


Figura 3.8: GPU_F2.

Esta etapa se encarga de realizar el segundo filtro de las etapas GPU. Su código se puede encontrar en los anexos, en la parte de código GPU. Realiza el mismo cálculo que la segunda etapa serie de ViVid, pero adaptado para la GPU bajo código OpenCL. Se añade la etapa CPU de salida, la cual recoge los argumentos de salida de la etapa 2 y lanza los argumentos de entrada para la etapa 3 de ViVid.

```
/*******************************
*OUTPUT_GPU2
******************************* */
    typedef multifunction_node<join2_t::output_type, buffer_datosGPU3 >
        mfn2_t;
    mfn2_t output_gpu2(g, unlimited, [&](const join2_t::output_type &m,
        mfn2_t::output_ports_type &ports ) {
       TGPU2 = tbb::tick_count::now();
       //std::cout <<"GPU 2= "<< (TGPU2 - TGPU1).seconds() <<" segundos"<<
           std::endl;
       get<0>(ports).try_put(opencl_coefficients);
       get<1>(ports).try_put(width);
       get<2>(ports).try_put(width);
       get<3>(ports).try_put(opencl_his);
       get<4>(ports).try_put(width);
       get<5>(ports).try_put(opencl_out);
       get<6>(ports).try_put(width);
```

```
    });
```

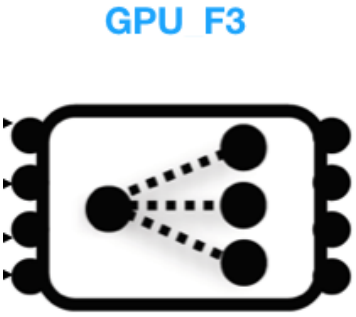


Figura 3.9: GPU_F3.

Esta etapa se encarga de realizar el tercer filtro de las etapas GPU. Su código se puede encontrar en los anexos, en la parte de código GPU. Realiza el mismo cálculo que la tercera etapa serie de ViVid, pero adaptado para la GPU bajo código OpenCL. Se añade el código de la etapa CPU de salida, la cual recoge los argumentos de salida de la tercera y última etapa de ViVid, además de la ID de la imagen, la cual viene procedente del dispatch_node directamente (al ejecutarse sólo una imagen simultaneamente en la GPU). En esta etapa se crea una tupla en la cual se añade los datos de salida y la ID de esa imagen. La salida de este nodo es esa tupla, en dirección hacia el nodo de salida final, además del token gpu que se libera hacia el token buffer del comienzo del programa. De esta forma la GPU puede comenzar a analizar una nueva imagen y ésta, que ya ha sido procesada, pasa al nodo de salida. Además lleva la cuenta de las imágenes analizadas por la GPU.

```
/*******************************
 *OUTPUT_GPU3
*******************************/
    typedef multifunction_node<join3_t::output_type, tuple<out_gpu,int>>
        mfn3_t;
    mfn3_t output_gpu3(g, unlimited, [&](const join3_t::output_type &m,
        mfn3_t::output_ports_type &ports ) {
        TGPU3 = tbb::tick_count::now();
        int token_gpu=0;
        //std::cout <<"GPU 3= "<< (TGPU3 - TGPU2).seconds() <<" segundos"<<
            std::endl;
        //std::cout << "Total GPU="<<(TGPU3-TGPU).seconds() <<" segundos"<<
            '\n';
        cont++;
        out_gpu out_gpu1;
        get<0>(out_gpu1)=get<5>(m);//Image
        get<1>(out_gpu1)=get<7>(m);//ID

        get<0>(ports).try_put(out_gpu1);//Image+ID
        get<1>(ports).try_put(token_gpu);//Token
    });
```

Se va a mostrar a continuación la parte CPU de la ejecución de ViVid. La otra bifur-

cación que se producía en el dispatch_node, la cual generaba las 3 etapas de ViVid en la CPU.

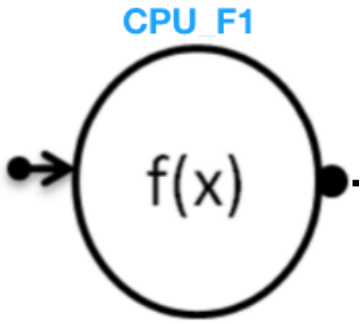


Figura 3.10: CPU_F1.

Primera etapa CPU, este nodo se encarga de invocar la primera etapa del filtro de ViVid, ya explicado en la versión serie, además de crear las variables que necesita dicho filtro, y pasar al siguiente nodo las variables que se usarán en el histograma (los dos arrays ya explicados en la versión serie de ViVid), la segunda etapa. Recoge desde el "connection_node" la ID de la imagen y crea una copia nueva de la misma para analizarla. Mediante una tupla de datos guarda todas estas variables. Esta tupla pasa a través de las 3 etapas CPU.

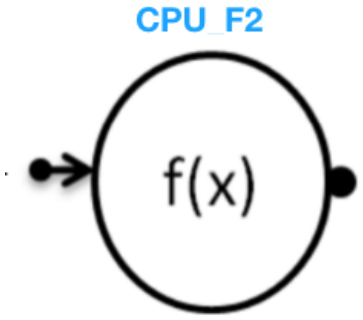
```
/*******************************
 *CPU_STAGES
******************************* */

  function_node<int,datos_imagen> cpu_f1(g,unlimited,[&](int ID_imagen) {

    datos_imagen datos2;
    get<6>(datos2)=ID_imagen;

    float* image2=(float *)malloc(sizeof(float)* height*width);
    memcpy(image2,f_imData,height*width);
    get<0>(datos2)=image2;
    get<5>(datos2)=1;

    get<1>(datos2)=(float *)malloc(sizeof(float)* height*width);
    get<2>(datos2)=(float *)malloc(sizeof(float)* height*width);
    float* filter_bank = (float*) malloc(num_filters * filter_dim *
        filter_dim * sizeof(float));
    for (int i = 0; i < num_filters * filter_dim * filter_dim; i++)
      {
        filter_bank[i] = float( std::rand() ) / RAND_MAX;
      }
    cosine_filter_transpose(get<0>(datos2),filter_bank,height,width,
        filter_dim,filter_dim,num_filters,get<1>(datos2),get<2>(datos2));
    return datos2;
  });
```

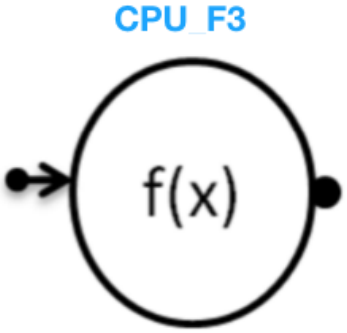


Segunda etapa CPU. Se encarga de realizar el histograma a la imagen, a partir de los resultados de la primera etapa. Devuelve el histograma y lo pasa a la tercera etapa CPU. La función que invoca es la que se analizó en la segunda etapa del filtro ViVid serie.

Figura 3.11: CPU_F2.

```
function_node<datos_imagen,datos_imagen> cpu_f2(g,unlimited,[&](
    datos_imagen datos) {

    float* his = block_histogram(get<1>(datos),get<2>(datos), num_filters
        , 8, 0, 0, height, width);
    get<3>(datos)=his;
    free(get<1>(datos));
    free(get<2>(datos));
    return datos;
  });
```



Tercera y última etapa CPU. Se encarga de comprobar si se ha detectado el objeto buscado o no. La función que invoca el nodo es la que se explicó en la tercera etapa del filtro ViVid, en la implementación serie. Pasa la tupla de datos al nodo de salida de las etapas CPU, con los resultados obtenidos.

Figura 3.12: CPU_F3.

```
function_node<datos_imagen,datos_imagen> cpu_f3(g,unlimited,[&](
    datos_imagen datos) {

    float * coefficients = (float *) malloc(sizeof(float) * n_total_coeff
        );
    for(int i = 0; i < n_total_coeff; i++)
      {
        coefficients[i] = float(std::rand())/RAND_MAX;
      }
    float *results = pwdist_c(coefficients, n_total_coeff/dict_size,
        dict_size, get<3>(datos), n_parts_x * n_parts_y , num_filters );
    get<4>(datos)=results;
    free(get<3>(datos));
    free(get<4>(datos));
    return datos;
```

```
});
```

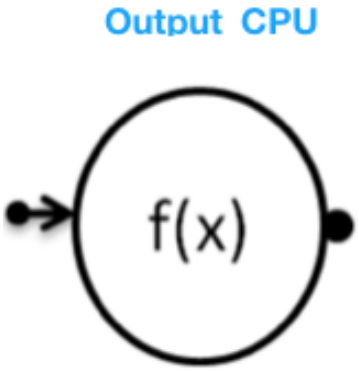


Figura 3.13: Output CPU.

El "output_cpu" se encarga de recoger la salida proporcionada por la CPU. Además aumenta el contador de las imagenes analizadas por la CPU. Se encarga de crear una tupla de datos de salida CPU, con los resultados del filtro y la ID de la imagen asociada, para pasarlas al nodo de salida del grafo. Además libera el token CPU para que vuelva al token buffer y que otro thread de la CPU pueda cogerlo y analizar una nueva imagen.

```
typedef multifunction_node<datos_imagen, tuple<out_cpu,int>> mfn5_t;
mfn5_t output_cpu(g, unlimited, [&](datos_imagen datos, mfn5_t::
    output_ports_type &ports ) {
     cont_cpu++;
     out_cpu out_cpu1;
     get<0>(out_cpu1)=get<4>(datos);//Image
     get<1>(out_cpu1)=get<6>(datos);//ID

     get<0>(ports).try_put(out_cpu1);//Image+ID
     get<1>(ports).try_put(get<5>(datos));//Token
 });
```

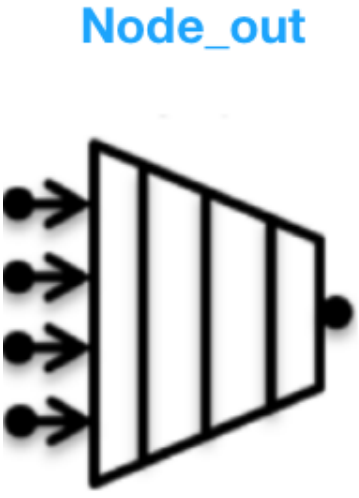


Figura 3.14: Node Out.

El "node_out" se encarga de unir los dos caminos, CPU y GPU, de forma que se unifiquen las salidas. Conecta el camino CPU y el camino GPU con el nodo de salida final para trasladar las imágenes y sus ID respectivos. Es un nodo diferente a los anteriores. Los "join_node" (todos los anteriores), son nodos que solo proporcionan una salida cuando tienen un item en cada uno de sus input disponibles. Si es un "join_node" que une 2 caminos, hasta que no hay 2 item disponibles en los 2 caminos no genera una salida. Esta configuración no sirve en este caso, pues lo que se quiere conseguir es que las salidas pasen al nodo de salida, independientemente del numero que generen cada uno de los dos caminos (no esperar que la CPU y la GPU calculen exactamente el mismo número de imagenes). Por lo tanto se emplea un "indexer_node". Este nodo permite generar una salida con cada entrada que reciba, independientemente de lo que ocurra en el resto de sus entradas disponibles.

```
    typedef indexer_node<out_gpu,out_cpu> join4_t;
    join4_t node_out(g);
```

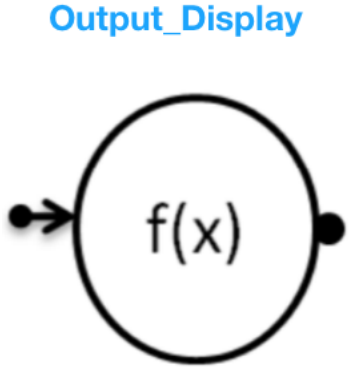


Figura 3.15: Output Display.

El "output_display" se encarga de recoger las salidas proporcionadas tanto por la CPU como la GPU, que previamente han sido unidas por el "node_out". Este nodo, por su particularidad de que no necesita que haya una entrada de cada tipo para generar una salida, al recibir una entrada la genera como salida unido a una etiqueta identificativa. Esta etiqueta es el número de la entrada por la que ha llegado el item. De esta manera permite identificar de qué entrada proceden las salidas que va generando el nodo. De esta forma en el nodo de salida, "output_display", se puede identificar si la tupla que se recibe (datos de salida + ID de la imagen) procede de la GPU o de la CPU. Así, en este nodo, se tiene identificado tanto los datos de salida de cada imagen, junto con su identificador, así como su procedencia: si ha sido ejecutado en la CPU o en la GPU. Con este nodo finaliza el grafo y la ejecución vuelve de nuevo al comienzo del mismo, gracias a los tokens, hasta que no haya más imágenes que analizar. Una vez ocurra ésto y las últimas imágenes lleguen a este nodo, el programa finalizará.

```
/*****************************
 *OUTPUT_DISPLAY
*****************************/
    function_node<join4_t::output_type> output_display(g, unlimited, [&](
        const join4_t::output_type &m){
      if(m.tag()==0){
        //std::cout << "gpu" << '\n';
        out_gpu gpu=cast_to<out_gpu>(m);
        //std::cout << get<1>(gpu) << '\n';

      }else{
        //std::cout << "cpu" << '\n';
        out_cpu cpu=cast_to<out_cpu>(m);
        //std::cout << get<1>(cpu) << '\n';
      }

    });
```

Por último, una vez explicados todos los nodos empleados, queda mostrar cómo se han realizado las conexiones entre ellos. Para unir dos nodos, en Flow Graph se emplea la orden "make_edge(X,Y)" siendo X e Y los dos nodos que se quieren unir. De esta forma los mensajes que salgan del nodo X van al nodo Y directamente, creandose así el grafo de nodos entrelazados. Los nodos que tienen una única entrada y salida se conectan directamente a través de los make_edge, sin embargo, si se tienen múltiples entradas y salidas, se deben unir cada una de éstas identificando si és entrada o salida mediante input_port u output_port respectivamente (y el número de la entrada o salida del nodo que ocupen), para cada uno de los elementos que entren o salgan del nodo.

Se van a mostrar, como cierre del capítulo, el listado de "make_edge" que han sido necesarios para crear el grafo con todos los nodos que se han ido desarrollando en los párrafos anteriores.

```
/*******************************
Edges from source_node to dispatch_node and CPU stages
*******************************/
    make_edge(input_node,input_port<0>(join_input));
    make_edge(token_buffer,input_port<1>(join_input));
    make_edge(join_input,connection_node);
    make_edge(output_port<0>(connection_node), cpu_f1);

    make_edge(cpu_f1,cpu_f2);
    make_edge(cpu_f2,cpu_f3);
    make_edge(cpu_f3,output_cpu);
    make_edge(output_port<0>(output_cpu),input_port<1>(node_out));
    make_edge(output_port<1>(output_cpu),token_buffer);

/*******************************
Edges from Node 1 GPU
*******************************/
    make_edge(output_port<1>(connection_node), dispatch_node);

    make_edge(output_port<0>(dispatch_node),input_port<0>(GPU_F1));
    make_edge(output_port<1>(dispatch_node),input_port<1>(GPU_F1));
    make_edge(output_port<2>(dispatch_node),input_port<2>(GPU_F1));
    make_edge(output_port<3>(dispatch_node),input_port<3>(GPU_F1));
    make_edge(output_port<4>(dispatch_node),input_port<4>(GPU_F1));
    make_edge(output_port<5>(dispatch_node),input_port<5>(GPU_F1));
    make_edge(output_port<6>(dispatch_node),input_port<6>(GPU_F1));
    make_edge(output_port<7>(dispatch_node),input_port<7>(GPU_F1));
    make_edge(output_port<8>(dispatch_node),input_port<8>(GPU_F1));

    make_edge(output_port<0>(GPU_F1),input_port<0>(node_joinGPU));
    make_edge(output_port<1>(GPU_F1),input_port<1>(node_joinGPU));
    make_edge(output_port<2>(GPU_F1),input_port<2>(node_joinGPU));
```

```
    make_edge(output_port<3>(GPU_F1),input_port<3>(node_joinGPU));
    make_edge(output_port<4>(GPU_F1),input_port<4>(node_joinGPU));
    make_edge(output_port<5>(GPU_F1),input_port<5>(node_joinGPU));
    make_edge(output_port<6>(GPU_F1),input_port<6>(node_joinGPU));
    make_edge(output_port<7>(GPU_F1),input_port<7>(node_joinGPU));
    make_edge(output_port<8>(GPU_F1),input_port<8>(node_joinGPU));

    make_edge(node_joinGPU,output_gpu1);

/*******************************
Edges from Node 2 GPU
*******************************/
    make_edge(output_port<0>(output_gpu1),input_port<0>(GPU_F2));
    make_edge(output_port<1>(output_gpu1),input_port<1>(GPU_F2));
    make_edge(output_port<2>(output_gpu1),input_port<2>(GPU_F2));
    make_edge(output_port<3>(output_gpu1),input_port<3>(GPU_F2));
    make_edge(output_port<4>(output_gpu1),input_port<4>(GPU_F2));
    make_edge(output_port<5>(output_gpu1),input_port<5>(GPU_F2));
    make_edge(output_port<6>(output_gpu1),input_port<6>(GPU_F2));
    make_edge(output_port<7>(output_gpu1),input_port<7>(GPU_F2));
    make_edge(output_port<8>(output_gpu1),input_port<8>(GPU_F2));

    make_edge(output_port<0>(GPU_F2),input_port<0>(node_joinGPU2));
    make_edge(output_port<1>(GPU_F2),input_port<1>(node_joinGPU2));
    make_edge(output_port<2>(GPU_F2),input_port<2>(node_joinGPU2));
    make_edge(output_port<3>(GPU_F2),input_port<3>(node_joinGPU2));
    make_edge(output_port<4>(GPU_F2),input_port<4>(node_joinGPU2));
    make_edge(output_port<5>(GPU_F2),input_port<5>(node_joinGPU2));
    make_edge(output_port<6>(GPU_F2),input_port<6>(node_joinGPU2));
    make_edge(output_port<7>(GPU_F2),input_port<7>(node_joinGPU2));
    make_edge(output_port<8>(GPU_F2),input_port<8>(node_joinGPU2));

    make_edge(node_joinGPU2,output_gpu2);

/*******************************
Edges from Node 3 GPU
*******************************/
    make_edge(output_port<0>(output_gpu2),input_port<0>(GPU_F3));
    make_edge(output_port<1>(output_gpu2),input_port<1>(GPU_F3));
    make_edge(output_port<2>(output_gpu2),input_port<2>(GPU_F3));
    make_edge(output_port<3>(output_gpu2),input_port<3>(GPU_F3));
    make_edge(output_port<4>(output_gpu2),input_port<4>(GPU_F3));
    make_edge(output_port<5>(output_gpu2),input_port<5>(GPU_F3));
    make_edge(output_port<6>(output_gpu2),input_port<6>(GPU_F3));

    make_edge(output_port<0>(GPU_F3),input_port<0>(node_joinGPU3));
    make_edge(output_port<1>(GPU_F3),input_port<1>(node_joinGPU3));
    make_edge(output_port<2>(GPU_F3),input_port<2>(node_joinGPU3));
```

```
    make_edge(output_port<3>(GPU_F3),input_port<3>(node_joinGPU3));
    make_edge(output_port<4>(GPU_F3),input_port<4>(node_joinGPU3));
    make_edge(output_port<5>(GPU_F3),input_port<5>(node_joinGPU3));
    make_edge(output_port<6>(GPU_F3),input_port<6>(node_joinGPU3));

    make_edge(output_port<9>(dispatch_node),input_port<7>(node_joinGPU3));
    make_edge(node_joinGPU3,output_gpu3);

    make_edge(output_port<0>(output_gpu3),input_port<0>(node_out));
    make_edge(node_out,output_display);
    make_edge(output_port<1>(output_gpu3),token_buffer);
```