Acceleració d'algorismes d'anàlisis d'imatge mèdica

Pol Sabaté Rulduà

Resum— Aquest treball tracta sobre l'optimització del processament morfològic d'imatges, específicament aplicant l'operació de closing sobre un conjunt d'imatges, utilitzant GPU i FPGA per comparar resultats i temps d'execució. Es fan servir CUDA i OpenCL per a les GPU, amb dues versions del programa: una que utilitza memòria compartida i una altra que no. Per a les FPGA, s'utilitza OmpSs@FPGA. L'estudi analitza les diferències de rendiment, eficiència i velocitat de processament entre aquestes tecnologies, destacant les avantatges i limitacions segons les necessitats específiques de l'aplicació.

Paraules clau—GPU, memòria compartida, element estructurador, OpenCL, CUDA, FPGA, OmpSs, closing, kernel, fil d'execució, blocs, graella, work-item, work-group

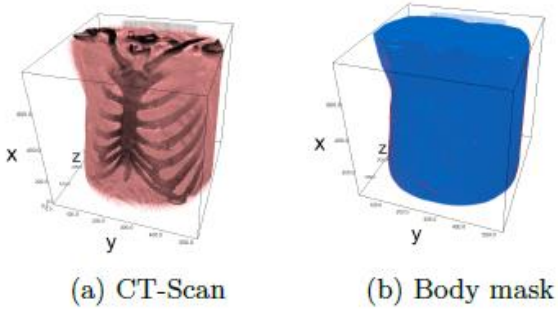
Abstract— This work focuses on optimizing morphological image processing, specifically applying the closing operation to a set of images, using GPU and FPGA to compare results and execution times. CUDA and OpenCL are used for the GPUs, with two versions of the program: one utilizing shared memory and one that does not. For the FPGA, OmpSs@FPGA is used. The study analyzes the performance, efficiency, and processing speed differences between these technologies, highlighting their advantages and limitations based on the specific application needs.

Index Terms—GPU, shared memory, structuring element, OpenCL, CUDA, FPGA, OmpSs, closing, kernel, thread, blocks, grid, work-item, work-group

1 Introducció - Context del treball

ACTUALMENT, el camp de la medicina i el camp de la informàtica no es poden separar, ja sigui per màquines manipulades pel personal sanitari a l'hora de realitzar cirurgies o bé per al processament d'imatges mèdiques. Pel que fa a aquest últim cas, en la part de preprocessament de les imatges s'utilitza el processament morfològic. Aquest tipus de processament d'imatges es basa en la morfologia dels objectes per a efectuar operacions que analitzen i modifiquen la mateixa forma d'aquests. Molts algorismes que treballen amb imatges de TC (tomografia computada) tenen en comú una fase inicial on s'obté una màscara tridimensional de l'àrea del cos a estudiar. La qual posteriorment s'utilitzarà per a descartar informació irrellevant i reduir la dimensionalitat de les dades.

Les TC proporcionen un valor de densitat per a cada punt del volum escanejat. Com es veurà en la figura 1, la màscara del cos es defineix com una matriu booleana 3D que identifica cada punt del TC original com a part o no del cos. Podem utilitzar un valor llindar inferior per obtenir els punts amb una alta probabilitat de pertànyer al cos [1].



(a) CT-Scan (b) Body mask

Fig. 1: Exemple de màscara del cos [1]

Abans d'entrar en detall sobre les operacions morfològiques és important esmentar la convolució binària, que és una tècnica fonamental en el processament d'imatges binàries. Aquest tipus de convolució implica l'aplicació d'un filtre, o també anomenat kernel, sobre una imatge binària per a realitzar diverses transformacions. El filtre es desplaça per tota la imatge i, a cada posició, combina els valors de la imatge amb els del kernel segons una regla específica, operacions AND i/o OR. I, per tant, al tractar en imatges binàries, on només existeixen dos valors possibles, s'afegiran o s'eliminaran píxels d'aquest objecte modificant així la seva forma [2].

Un cop s'ha preparat la imatge amb tècniques com l'explicada en els paràgrafs anteriors, es poden aplicar operacions morfològiques per refinar les màscares tridimensionals.

Una de les operacions morfològiques que es du a terme

- *E-mail de contacte: 1314polsabate@gmail.com*
- *Menció realitzada: Enginyeria de Computadors*
- *Treball tutoritzat per: David Castells Rufas (Departament de Microelectrònica i Sistemes Electrònics)*
- *Curs 2023/24*

és la de closing, que és en la que ens centrarem en aquest treball. Aquesta operació està formada per dues operacions bàsiques, la dilació i l'erosió. La primera d'elles és útil per tancar o completar objectes que puguin tenir forats o esquerdes. El problema que hi ha és que aquesta operació fa que la mida en general de l'objecte sigui més gran, cosa que no volem que passi. Per tant, per a evitar aquest problema podem aplicar una operació d'erosió posteriorment que evita l'ampliació de l'objecte. L'operació quedaria de la següent manera:

$$G \cdot M = (G \oplus M) \ominus M$$

On G respresenta la imatge binària original, M és l'element estructurador o kernel, $\oplus$ representa l'operació de dilació i $\ominus$ l'erosió. En conseqüència, els objectes s'expandeixen segons l'element M, omplint els petits forats i connectant components propers. I tot seguit s'aplica l'operació d'erosió, perquè l'objecte torni a tenir la mida original.[2]

Un cop vist a quin problema ens enfrontem podem veure les diferents eines que utilitzarem per a l'optimització d'aquesta operació de closing. Les eines que s'utilitzaran són CUDA, OpenCL amb C i python i OmpSs per a FPGA.[2]

CUDA (Compute Unified Device Architecture) és una plataforma de computació paral·lela, i un model de programació creat per NVIDIA per a permetre l'acceleració de l'execució de diferents programes fent servir la GPU. La plataforma de CUDA és una capa de software que dona accés al conjunt virtual d'instruccions de la GPU i els seus elements de còmput paral·lel per executar nuclis de còmput.. CUDA només es pot fer servir en GPUs desenvolupades per la mateixa NVIDIA, la qual és la creadora d'aquesta plataforma.[3]

OpenCL (Open Computing Language) és un framework per a programes que s'executen en CPUs, GPUs, DSPs, FPGAs i d'altres processadors heterogenis. És un estàndard obert i lliure per a la programació paral·lela de GPUs entre altres plataformes. OpenCL és una API per a coordinar la computació paral·lela entre diferents processadors i un llenguatge multiplataforma i una especificació de l'entorn de computació [4].

I per acabar OmpSs és un model de programació paral·lela desenvolupat pel Barcelona Supercomputing Center (BSC). El nom prové de la combinació de OpenMP i StarSs, dos altres models de programació paral·lela. OmpSs té com a objectiu proporcionar un entorn de programació flexible per a la programació amb sistemes multiprocessadors heterogenis, incloent-hi tant CPUs, GPUs i FPGAs. Les dues plataformes anteriors són una extensió dels llenguatges de programació, ja sigui C; C++; python... En canvi, OmpSs utilitza directives de compilador per indicar les regions de codi que s'han de paral·lelitzar. Aquestes directives, a través de #pragma, permeten de manera senzilla indicar al compilador com interpretar aquestes regions [5].

2 Estat de l'art

Les imatges mèdiques de la ressonància magnètica (RM) o escàners CT o PET tenen algunes similituds. Les imatges resultants s'organitzen com un conjunt d'imatges 2D que s'agrupen en una matriu 3D. Els valors dels píxels, o vòxels, descriuen propietats del teixit trobat a la ubicació. En els escàners CT, els valors es mesuren en l'escala Hounsfield, que informa sobre l'atenuació del teixit als senyals electromagnètics. En RM, els valors d'intensitat obtinguts tenen una interpretació més complexa [1].

Zheng et al. (2019) [6] utilitzen xarxes neuronals convolucionals (CNN) basades en projecció d'intensitat màxima per a la detecció automàtica de nòduls pulmonars en escàners CT. Aquesta tècnica millora la precisió i eficiència en la detecció de nòduls.

Maier et al. (2020) [7] han desenvolupat PyQMRI, una caixa d'eines en Python per a la ressonància magnètica quantitativa accelerada. Aquesta eina utilitza PyOpenCL per a la generació de codi en temps d'execució a la GPU, millorant així la velocitat de processament.

Klöckner et al. (2012) [8] presenten PyCUDA i PyOpenCL, que permeten la generació de codi per a la GPU mitjançant scripts, facilitant el desenvolupament de solucions accelerades per a operacions morfològiques i altres aplicacions computacionals intensives.

Garcia-Uceda et al. (2021) [9] utilitzen CNNs robustes i eficients per a la segmentació automàtica de les vies aèries a partir d'imatges de tomografia computada. Aquesta eina és eficient en la identificació i segmentació precisa de les vies aèries, essencial per a diagnòstics mèdics.

3 Objectius

Un cop vista la descripció del problema podem definir els objectius del treball.

El primer d'ells serà la implementació de dues versions utilitzant CUDA per a l'execució en GPU, per a posteriorment realitzar un estudi del rendiment. Aquest estudi serà un perfilatge d'ambdues versions i una posterior comparació.

El segon objectiu serà el mateix que el primer, però en més d'utilitzar CUDA es farà servir el framework OpenCL amb C i pyhton. I es farà una comparació de les dues eines per veure quina és més adequada en aquest cas.

I l'últim objectiu de tots serà el d'aconseguir una versió i un posterior perfilatge que s'executi en una FPGA per veure les diferències amb les dues versions anteriors que s'executen en una GPU.

4 Metodologia

Per aconseguir el compliment d'aquests objectius la metodologia que s'utilitzarà serà la iterativa. Aquesta metodologia es basa en la repetició de cicles de treball curts anomenats iteracions, dividint el treball en parts més petites i més fàcils de manejar.

Els cicles són els següents:

Planificació: es realitza una planificació inicial del projecte, identificant els requisits i els objectius.

Disseny: es crea un disseny basat en els requisits identificats inicialment. Aquest pot ser revisat i modificat posteriorment.

Proves: un cop tenim una versió inicial del producte es duen a terme proves per a detectar errors i problemes.

Avaluació: un cop s'acaba una iteració es fa una recopilació de la informació que es té fins aquest punt. Amb aquesta informació es fa una revisió del producte i es determina si hi ha d'haver canvis en les pròximes iteracions.

Iteracions successives: el procés es va repetint fins que s'arriba al final, és a dir s'aconsegueixen completar satisfactòriament tots els objectius definits al principi.

Finalització del projecte: quan es completa el desenvolupament es fa una última fase de proves i s'entrega el producte final.

Aquesta metodologia s'adapta bé al projecte a desenvolupar, ja que partint d'una versió inicial a través d'aquestes iteracions es poden arribar a completar els objectius definits. I no sol això, per a cada un dels objectius si es van fent iteracions modificant paràmetres de la GPU i utilitzant memòria compartida es pot arribar a un nivell de detall en les versions del programa més elevat.

5 DESENVOLUPAMENT

Per a realitzar els perfilatges de les diferents versions treballarem sobre un codi proporcionat pel mateix tutor d'aquest treball, David Castells Rufas. Aquest codi en un principi està desenvolupat en python i utilitza una sèrie de llibreries per a tractar els elements d'entrada i sortida que són imatges.

5.1 Entorn de treball

Les diferents versions que s'executen en la GPU ho fan a la NVIDIA GeForce GTX 1650 i la NVIDIA 2080 Ti.

En la primera d'elles hi ha un total de 1024 NVIDIA CUDA Cores, amb una freqüència base de 1020 a 1395 MHz i una freqüència impulsada de 1245 a 1560 MHz. El nombre de threads (fils d'execució) per warp és de 32, el màxim de warps per multiprocessador (SM) és de 32 i el màxim de blocs de threads per SM és de 16. La mida màxima de threads per SM és de 1024 i el màxim de threads en un bloc també és de 1024. Té un total de 14 SM, on els SM són els blocs de construcció fonamentals que contenen els nuclis CUDA, unitats de memòria cau, unitats de textura, i altres components necessaris per al processament paral·lel massiu [10].

La velocitat de la memòria és de 8 Gbps i l'ample de banda de la mateixa és de 128 GB/s [10].

Aquesta té una arquitectura Turing. Segons NVIDIA (2018), "Within the core architecture, the key enablers for Turing's significant boost in graphics performance are a new GPU processor (streaming multiprocessor—SM) architecture with improved shader execution efficiency, and a new memory system architecture that includes support for the latest GDDR6 memory technology.

Turing GPUs also inherit all the enhancements to the NVIDIA CUDA™ platform introduced in the Volta architecture that improve the capability, flexibility, productivity, and portability of compute applications. Features such as independent thread scheduling, hardware-accelerated Multi Process Service (MPS) with address space isolation for multiple applications, and Cooperative Groups are all part of the Turing GPU architecture." (NVIDIA, 2018) [11].

La segona GPU, la NVIDIA GeForce RTX 2080 Ti té un total de 4352 CUDA cores. 11 GB de memòria GDDR6 connectats mitjançant una interfície de memòria de 352 bits. La GPU funciona a una freqüència de 1350 MHz, que es pot augmentar fins a 1545 MHz, i la memòria funciona a 1750 MHz (14 Gbps efectius). L'ample de banda és de 616 GB/s i té un total de 68 SMs [12].

Pel que fa a la FPGA aquesta és una ALVEO U200. Aquesta pot arribra a proporcionar fins a 90 vegades més rendiment que les CPUs per a càrregues de treballar. Construïdes amb l'arquitectura AMD UltraScale de 16nm.

Aquestes FPGAs tenen una capacitat de memòria de 64 GB, l'ample total de memòria és de 77 GB/s, la capaitat de les SRAM és de 35 MB i l'ample de banda total d'aquestes memòries és de 31 TB/s. El nombre de taules de cerca (Look-Up-Tables) és de 892000. El número de Tera operacions màxim INT8 per segons és de 18,6. Aquesta mesura indica la capacitat màxima de processament en Tera operacions per segon utilitzant operacions aritmètiques en nombres enters de 8 bits [13].

5.2 Codi

Un cop vist l'entorn on s'executarà el nostre programa/kernel podem veure el codi en si.

Si comencem amb la versió seqüencial d'aquest, al principi com veurem a la figura 2, podem observar com tenim l'element estructurador de cinc posicions (central, superior, dreta, esquerra i inferior) en forma de creu que serveix per a fer la lectura dels píxels veïns a l'actual. Tot seguit hi trobem tres bucles anidats a través dels quals recorrerem tots els píxels de totes les imatges. On el més exterior ens posiciona sobre la imatge corresponent i els dos següents recorren la imatge en qüestió. Aquí trobem la inicialització de l'element vo a u, que posteriorment com veurem en la figura 2 ens servirà per a fer l'operació binària AND. S'inicialitza a u, ja que per a aquesta operació és el valor neutre.

```
void close(const char* inImg, char* outImg, const int XS, const int YS, const int ZS,
           const int oxs, const int oys, const int ozs)
{
    int sex[] = { -1, 1, 0, 0, 0 };
    int sey[] = { 0, 0, 0, 1, -1 };

    int see_x[] = { -1, 1, 0, 0, 0 };
    int see_y[] = { 0, 0, 0, 1, -1 };

    for (int z = 0; z < ZS; z++)
    {
        for (int y = 0; y < YS; y++)
        {
            for (int x = 0; x < XS; x++)
            {
                int vo = 1;
```

Fig. 2: Inicialització funció close seqüencial

Un cop arribat a aquest punt on ja se sap a quin píxel es vol accedir necessitem saber qui son els elements veïns. Per això utilitzem l'element estructurador, esmentat anteriorment, a través de dos bucles més anidats. Aquests dos bucles ens donen la informació de cap a on fer el desplaçament per trobar el veí. Un cop es sap la posició, s'accedeix al valor i s'efectua un altra operació binària, però aquest cop una OR sobre una variable, vi, prèviament inicialitzada a 0. Com en el cas de l'operació AND, s'inicialitza al valor neutre de l'operació a realitzar. Ho podem observar en la figura 3.

Un cop acabat el bucle més intern es realitza l'AND esmentada prèviament i es continua iterant fins que s'acaba el bucle més extern dels dos, moment en el qual és guarda el valor al vector de sortida.

```
for (int k = 0; k < 5; k++)
{
    int vi = 0;

    for (int i = 0; i < 5; i++)
    {
        int lx = x + sex[i] - see_x[k];
        int ly = y + sey[i] - see_y[k];

        if (lx >= 0 && lx < XS && ly >= 0 && ly < YS)
        {
            int possrc = inImg[z * XS * YS + ly * XS + lx];
            vi |= possrc;
        }
    }
    vo &= vi;
}

int posdst = x * oxs + y * oys + z * ozs;
outImg[posdst] = vo;
```

Fig 3: Operació de closing

6 Execució GPU

Un cop vista la versió seqüencial del programa podem passar a veure les versions que s'executaran a la GPU.

El motiu pel qual provem l'execució en GPU és perque al treballar amb píxels que no depenen un dels altres no existeix una dependència que provoqui que els fils d'execució s'hagin d'esperar a certs resultats. Aleshores podem aprofitar el parel·lelisme que ens ofereix la GPU.

6.1 CUDA

En aquesta secció s'explicarà la versió del programa implementada utilitzant CUDA. Com hem dit en la introducció CUDA (Compute Unified Device Architecture) és una plataforma de computació paral·lela de programació creada per NVIDA. A través de les funcions i els paràmetres de CUDA intentarem optimitzar el rendiment del programa, perquè aquest vagi més ràpid que la versió seqüencial.

Uns paràmetres molt importants i que més avant, a l'hora d'analitzar els resultats veurem l'impacte que tenen en el rendiment són la mida dels blocs de threads i la mida del grid.

Pel que fa a la mida dels blocs definim, com veurem en la figura 4, a través d'una variable de tipus dim3 quants fils d'execució hi haurà en cada un dels blocs. I no només definim el número sinó que també definim les dimensions, en el nostre cas seran de dos, o de tres i la tercera dimensió Z només tindrà un únic thread. Aquests fils s'executen simultàniament i poden cooperar i comunicar-se entre ells.

I la mida del grid fa referència al nombre total de blocs que s'executaran en la GPU. La definició de la mida del grid implica en com es distribuirà la feina entre els blocs. I com en la mida dels blocs també podem definir tres dimensions, que en aquest cas sí que farem servir.

I abans d'exectuar el kernel en si hem de fer la còpia de les dades a la GPU a través de les funcions de CUDA cudaMalloc i cudaMemcpy i quan s'acaba el kernel copiar les dades de sortida a l'amfitrió com veurem a la figura 5.

```
//Definició mida del bloc i del grid
dim3 blockDim(THREADS_PER_BLOCK, THREADS_PER_BLOCK, 1);

int gridWidht = (width + blockDim.x - 1) / blockDim.x;
int gridHeight = (height + blockDim.y - 1) / blockDim.y;

gridHeight = ceil(gridHeight);
gridWidht = ceil(gridWidht);

dim3 gridDim(gridWidht, gridHeight, NUM_FOTOS);

close << < gridDim, blockDim >> > (width, height, NUM_FOTOS, 1, width, width * height, 1,
                                   width, width * height, imageArray_input_d, imageArray_output_d);
```

Fig. 4: Definició mida dels bloc i mida del grid

```
char* imageArray_output_d;
char* imageArray_input_d;
cudaMalloc((char**)&imageArray_input_d, sizeof(char) * width * height * NUM_FOTOS);
cudaMalloc((char**)&imageArray_output_d, sizeof(char) * width * height * NUM_FOTOS);

cudaError_t err;
err = cudaMemcpy(imageArray_input_d, imageArray_input, sizeof(char) * width * height * NUM_FOTOS, cudaMemcpyHostToDevice);
if (err != cudaSuccess)
{
    // Hi ha hagut un error
    printf("Error: %s\n", cudaGetErrorString(err));
    return -3;
}

//Definició mida bloc i grid i execució kernel (...)

err = cudaMemcpy(imageArray_output, imageArray_output_d, sizeof(char) * width * height * NUM_FOTOS, cudaMemcpyDeviceToHost);
if (err != cudaSuccess)
{
    printf("Error: %s\n", cudaGetErrorString(err));
    return -3;
}
```

Fig. 5: Reservar memòria a la GPU i còpia de les dades

Un cop vist com fem la definició dels paràmetres necessaris podem passar a veure les modificacions que s'han fet al kernel.

Els tres for anidats que teníem en la figura 1 passen a despareixer, com veurem en la figura 6, i passen a ser substituïts per l'identificador de cada thread. Aquest identificador, com hem definit blocs de tres dimensions, està format de tres variables, x, y i z que serien les equivalents a les mateixes variables dels bucles esmentats, però sense haver de fer-los. El càlcul ens dona un identificador únic per a cada un dels threads. Tot seguit s'ha de comprovar que aquests tres valors estiguin dins del límit de les imatges a través d'un if. La resta del càlcul amb l'element estructurador continua de la mateixa forma.

```
__global__ void close(const int XS, const int YS, const int ZS,

const int ixs, const int iys, const int izs,

const int oxs, const int oys, const int ozs,

char* inImg,

char* outImg)
{

int x = blockIdx.x * blockDim.x + threadIdx.x;
int y = blockIdx.y * blockDim.y + threadIdx.y;
int z = blockIdx.z * blockDim.z + threadIdx.z;
if ((z < ZS) && (y < YS) && (x < XS))
{
```

Fig. 6: Càlcul identificadors thread CUDA

6.2 OpenCL C

En aquest altre apartat veurem la diferència que hi ha entre les altres implementacions i la implementació en OpenCL utilitzant C, per a després comparar els resultats de cada una i veure les possibles diferències i el perquè.

La primera cosa que cal comentar és el desajust entre els termes de CUDA i OpenCL. Abans hem parlat de blocs de fils i de la mida del grid. Ara hem de parlar de work-item que seria l'equivalent a un thread i que també té un identificador únic. Després, no tenim blocs de fils d'execució sinó

que tenim work-groups, aquest grups es comporten com els blocs de CUDA. I per acabar tenim NDRange que seria el mateix que el grid de CUDA. L'NDRange especifica la mida global de l'espai de treball i es pot veure com una matriu de work-groups. D'aquesta manera es defineixen la totalitat de work-items que es llançaran en un kernel.
Aleshores, podem observar que aquesta diferència canviarà una mica el càlcul d'aquests paràmetres però realment no deixa de ser el mateix. Com veurem en la figura 7.
Tot seguit també hi ha un canvi a l'hora de reservar la memòria a la GPU, com es passen els arguments al kernel i com aquest es crida per a executar-se. Com es mostrarà en la figura 8.

```
//Definir mides del bloc
size_t block_size[3] = { THREADS_PER_BLOCK, THREADS_PER_BLOCK, 1 };

int gridWidht = (width + block_size[0] - 1) / block_size[0];
int gridHeight = (height + block_size[1] - 1) / block_size[1];

gridWidht = ceil(gridWidht);
gridHeight = ceil(gridHeight);

size_t grid_size[3] = { gridWidht, gridHeight, NUM_FOTOS };

size_t global_size[3] = {
    block_size[0] * grid_size[0],
    block_size[1] * grid_size[1],
    block_size[2] * grid_size[2]
};

size_t local_size[3] = {block_size[0], block_size[1], block_size[2]};
```

Fig. 7: Definició d l'NDRange i la mida de cada work-group

```
//Creació buffers GPU + transferència
inputBuffer = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(char) *  TOTAL_PIXELS, imageArray_input, &err);
checkError(err, "clCreateBuffer (input)");

outputBuffer = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(char) * WIDTH * HEIGHT * NUM_FOTOS, NULL, &err);
checkError(err, "clCreateBuffer (output)");

//Pas d'arguments al kernel
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &inputBuffer);
checkError(err, "clSetKernelArg (0)");
err = clSetKernelArg(kernel, 1, sizeof(cl_mem), &outputBuffer);
checkError(err, "clSetKernelArg (1)");
err = clSetKernelArg(kernel, 2, sizeof(int), &width);
checkError(err, "clSetKernelArg (2)");
err = clSetKernelArg(kernel, 3, sizeof(int), &height);
checkError(err, "clSetKernelArg (3)");
err = clSetKernelArg(kernel, 4, sizeof(int), &num_fotos);
checkError(err, "clSetKernelArg (4)");
err = clSetKernelArg(kernel, 5, sizeof(int), &x_step);
checkError(err, "clSetKernelArg (5)");
err = clSetKernelArg(kernel, 6, sizeof(int), &width);
checkError(err, "clSetKernelArg (6)");
err = clSetKernelArg(kernel, 7, sizeof(int), &total_pixels);
checkError(err, "clSetKernelArg (7)");
err = clSetKernelArg(kernel, 7, sizeof(int), &total_pixels);
checkError(err, "clSetKernelArg (7)");
```

Fig. 8: Pas d'argument si reserva de memòria OpenCL

I per acabar en el kernel la diferència estaria amb les funcions que ens retornen l'identificador únic de cada thread, ara work-item, com farem saber en la figura 9.

```
const char* kernel_source =

"__kernel void close(__global const char* inImg, __global char* outImg,"
                    "const int XS, const int YS, const int ZS, const int oxs, "
                    "const int oys, const int ozs) {"

"   int x = get_global_id(0);"
"   int y = get_global_id(1);"
"   int z = get_global_id(2);"
```

Fig 9: Declaració kernel OpenCL identificador únic

Pel que fa a la resta del codi és exactament el mateix que amb CUDA, ja que les altres instruccions són pròpies de C i no específiques d'alguna de les eines.

6.3 OpenCL Python

En aquest apartat s'explicarà com s'implementa la mateixa versió amb python i openCL per posteriorment també comparar els resultats amb les altres versions.

Primer de tot, com utilitzem python ens ha tocat canviar de llibreries per a processar les imatges d'entrada a bits i viceversa. L'únic que com no forma part del kernel hem decidit no entrar en detall, simplement fem servir les llibreries numpy i PIL.

Pel que fa a la resta de funcions que hem vist en l'apartat anterior d'OpenCL en C aquestes estan adaptades a python i canvien petits detalls. Aquests són per exemple la forma en la que passem els arguments i la forma en la qual definim l'NDRange i quants work-items hi ha en cada work-group, ho podem visualitzar en la figura 10.

```
mf = cl.mem_flags

#Crear buffers + transferencia a la GPU
inImg = cl.Buffer(context, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=all_images_array)
outImg = cl.Buffer(context, mf.WRITE_ONLY, all_images_array.nbytes)

#Càlcul paràmetres NDRange i mida local_work_size
block_size = (32, 32, 1)  # Equivalent a local_work_size en OpenCL
grid_size = (16, 16, 771)
global_size = (block_size[0] * grid_size[0], block_size[1] * grid_size[1], block_size[2] * grid_size[2])
local_size = block_size

#Execució kernel
kernel = program.close
kernel.set_args(inImg, outImg, np.int32(pixels_x), np.int32(pixels_y), np.int32(total_f),
                np.int32(1), np.int32(pixels_x), np.int32(total_pixels_per_imatge));

cl.enqueue_nd_range_kernel(queue, kernel, global_size, local_size)

#Copiar les dades de sortida
cl.enqueue_copy(queue, outImg_h, outImg)

#Esperar a que no quedi res a la cua
queue.finish();
```

Fig. 10: Versió pyOpenCL

I com seguim estant en el mateix framework, en aquest cas el kernel és exactament igual que en la versió anterior, és a dir, utilitzant les funcions de get_global_id per a cada una de les dimensions.

7 VERSIÓ MEMÒRIA COMPARTIDA

En aquesta secció es parlarà del perquè l'ús de la memòria compartida i després en les subseccions de la diferència d'implementació entre CUDA i openCL.

Un cop implementada la versió que hem explicat anteriorment vàrem pensar en possibles millores que podíem implementar. Com podrem observar posteriorment en la part de resultats quan fèiem el perfilatge vàrem veure que la memòria podia arribar a ser un possible coll d'ampolla. Cosa que ens va fer plantejar una possible optimització en aquest aspecte.

Aleshores revisant la documentació de CUDA vam veure la possibilitat d'utilitzar la memòria compartida.

La memòria compartida com es troba integrada directament al xip és molt més ràpida que la memòria local i global. La latència de la memòria compartida és aproximadament 100 vegades més baixa que la de la memòria global, sempre que no hi hagi memòria catxe, i sempre que no hi hagi conflictes de bancs entre els diferents fils. Aquest tipus de memòria s'assigna per a cada bloc de fils d'execució, de manera que tots els fils dins d'un bloc tenen accés a la mateixa memòria. Podent així compartir informació més ràpidament [14].

Aquests fils dins d'un mateix bloc poden accedir a les dades de la memòria compartida que s'han carregat prèviament des de la memòria global pels mateixos fils o altres fils d'execució dins del mateix bloc [14].

Aquesta capacitat combinada amb la sincronització de fils té diversos usos, com ara catxes gestionades pel mateix

usuari i per facilitar la col·lisió de la memòria global, que en altres casos no seria possible [14].

Un fet que cal controlar i hem de tenir en compte és el de la sincronització dels fils. En el cas que el nostre programa necessiti sincronització per a evitar condicions de carrera entre fils, i que així el programa tingui un comportament determinista, ens hem d'assegurar que estigui ben implementada [14].

Per assegurar aquest comportament determinista hem de sincronitzar els fils quan aquests estiguin cooperant. En CUDA existeix una primitiva de sincronització de barrera, __syncthreads(). Aquesta primitiva ens assegura que l'execució de cada un dels fils no pugui continuar després de trobar-se amb aquesta funció, i que s'hagin d'esperar al fet que tots els fils dins del mateix bloc l'hagin executat. Així doncs, evitem la problemàtica de la condició de carrera, impedint que els fils accedeixin a la memòria compartida abans que aquesta estigui inicialitzada correctament. Cal recalcar que fer un ús de __syncthreads() en codi divergent pot conduir a un bloqueig complet. Tots els fils haurien de cridar a la primitiva en el mateix punt [14].

Hi ha dues formes d'implementar la memòria compartida en CUDA. Si sabem la mida de la memòria en temps de compilació es pot definir de forma estàtica. Si la definim d'aquesta forma, s'ha de fer en el propi kernel que la farà servir. En canvi, si no sabem la quantitat de memòria que farem servir en temps de compilació s'ha de fer de forma dinàmica [14].

Per aconseguir un ample de memòria alt per a accessos concurrents, la memòria compartida es divideix en mòduls de la mateixa mida, bancs de memòria, als quals es pot accedir a la vegada. En conseqüència, qualsevol escriptura o lectura que s'estengui per N bancs diferents es pot fer simultàniament [14].

Si les adreces de memòria, de les lectures/escriptures, fetes per diferents fils es mapegen al mateix banc, aquests accessos se serialitzen. El maquinari divideix una petició de memòria conflictiva en tates peticions separades lliures de conflicte. L'inconvenient d'això és que quan se serialitzen els accessos disminueix l'ample de banda efectiu [14].

Els dispositius amb més capacitat de còmput de 3.x poden configurar la mida del banc a quatre o vuit bytes, depenent de quin tipus de dades volen i per a evitar conflictes. En cas de treballar amb vuit bytes es poden evitar conflictes si es treballa amb dades de precisió doble [14].

Per a resumir, podem dir que la memòria compartida és una característica potent per realitzar programes en CUDA i optimitzar-los molt bé. Com hem dit abans a l'estar la memòria integrada dins del xip en si, l'accés és molt més ràpid que no pas a la resta de les memòries que no són cachés [14].

Una manera d'aprofitar la memòria compartida i la cooperació dels fils és permetre la coalescència de la memòria global, si els threads dins d'un mateix warp accedeixen a posicions contigües (contigüitat de 32 bytes) de la memòria global, l'accés es considera coalescent. Això significa que la GPU pot recuperar les dades en un sol cicle d'accés a la memòria, en comptes de diversos cicles [14].

7.1 Disseny versió memòria compartida

Un cop vist que és la memòria compartida, les implicacions que té utilitzar-la i en general el seu funcionament, podem passar a explicar com l'hem implementat en el nostre codi i en el nostre problema.

El problema parteix en que estem treballant en un conjunt de set-cents setanta una imatges, de cinc-cents dotze píxels per cinc-cents dotze, i que aquestes s'acaben dividint entre N número de blocs, depenent de la quantitat de threads dins d'un bloc. I que cada thread ha de fer les lectures a la memòria global més d'un cop.

En el nostre cas els blocs són de trenta dos per trenta dos fils, per tant ens queden setze per setze per set-cents setanta un blocs. Per a fer la divisió correctament, el número de píxels total de les imatges hauria de ser divisible pel número de blocs. Recordem que aquests paràmetres els definim abans de l'execució del kernel.

Com podrem veure en la figura 11 els threads dins un mateix bloc, que seria cada element de la taula/graella, van a llegir els valors corresponents a la memòria global de la GPU tantes vegades com faci falta, tot i que la memòria cau pot evitar alguna d'aquestes lectures.

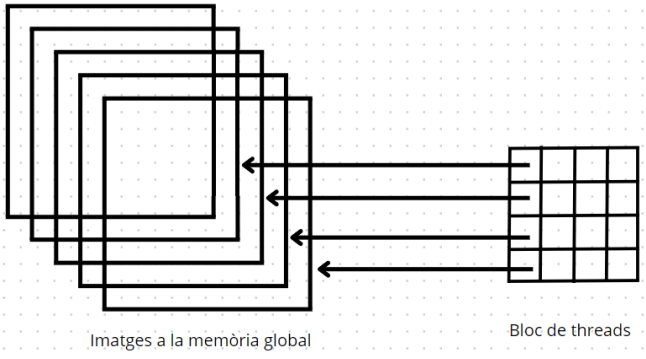


Fig. 11: Esquematització sense memòria compartida

Llavors el que volem intentar utilitzant la memòria compartida és el reduïr el nombre d'accessos a aquesta memòria global, per a intentar millorar el rendiment. El primer que vàrem pensar va ser fer la memòria compartida de la mida exacta del bloc, és a dir de trenta dos per trenta dos elements. Però com l'element estructurador, en el cas que el píxel actual sigui el límit del bloc tocaria accedir al bloc adjacent. En el cas d'estar al límit inferior tocaria accedir al bloc inferior respecte al que estem, si esetem al límit dret del bloc s'hauria d'accedir al bloc de la dreta respecte al que estem, i així per als altres límits.

El problema amb aquest fet, com ja hem dit, és que la intenció que tenim en utilitzar la memòria compartida és la de reduir els accessos el màxim possible a la memòria global, pel motius que hem vist al principi d'aquesta secció.

Per consegüent, i com veurem en la figura 12, hem de canviar el plantejament de la solució però sense abandonar l'idea de la memòria compartida.

Per tant, hem de fer que la mida de la memòria compartida sigui el suficientment gran com per a poder emmagatzemar els píxels que hi ha en un sol bloc, i tots els píxels adjacents a aquest, és a dir, els píxels dels blocs veïns. El que ens deixa en una mida d'aquesta memòria igual al número de threads més dos pel número de threads més dos. En el nostre cas de trenta quatre per trenta quatre.

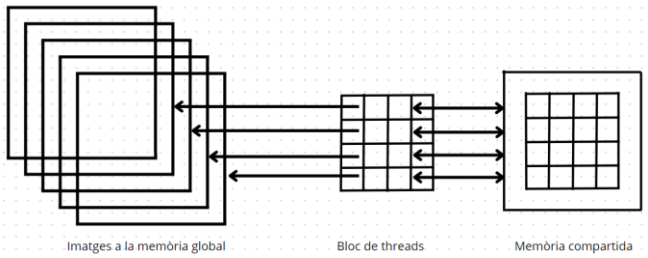


Fig. 12: Esquematització amb memòria compartida

Hi ha una gran diferència entre fer la memòria compartida de la mida exacta del bloc a fer-la el suficientment gran com per a que hi hagi espai per als píxels dels blocs veïns. En el primer cas els fils d'execució del límit del bloc haurien de fer les lectures dels blocs adjacents fora de la memòria compartida, és a dir, tornant a fer lectures a la memòria global. Però recordem que volem fer que aquestes lectures siguin les mínimes possibles, cosa que aconseguim si fem la memòria compartida amb més espai i l'inicialitzem prèviament. Llavors totes les lectures posterios quedaran dins de la memòria compartida millorant així el rendiment.

7.2 Implementació CUDA

Un cop hem fet la descripció del disseny de la solució veurem com s'ha portat a terme la implementació en CUDA.

Com hem comentat anteriorment hi ha dues maneres d'implementar la memòria compartida en CUDA depenent si en temps de compilació sabem o no la mida d'aquesta memòria. En el nostre cas, com si la sabem podem fer servir la memòria estàtica, com veurem en la figura 13. I el que també s'ha implementat és la lectura del bloc des de la memòria global a la compartida i tractar els casos límit dels blocs que hem comentat en l'apartat anterior.

Per a fer aquesta comparació veiem que hi ha dos variables noves, tx i ty, que emmagatzemen el valor local del thread dins del bloc. La diferència amb els valors x i y és que aquests dos últims són l'identificador global dels fils d'execució, en canvi, aquests dos nous es repeteixen per cada bloc. Per tant, si aquests valors, tx i ty, són per exemple zero, sabem que necessitem copiar el valor o bé del bloc de l'esquerra o bé del bloc superior, segons si és l'identificador tx o ty.

També és important comentar que la constant SHARED_MEM_SIZE està definida com a THREADS_PER_BLOCK + 2.

Com es pot apreciar també, quan guardem els valors a la memòria compartida li sumem u als índexs ty i tx per a deixar espai a la fiila superior i a la columna de l'esquerra. Com per la dreta i per baix no arribarem al límit no passa res. Ara en la figura 14 veurem com es tracten els casos dels fils d'execució que estan en alguns dels límits.

```
int x = blockIdx.x * blockDim.x + threadIdx.x;
int y = blockIdx.y * blockDim.y + threadIdx.y;
int z = blockIdx.z * blockDim.z + threadIdx.z;

int tx = threadIdx.x;
int ty = threadIdx.y;

//Memòria compartida
__shared__ char shared[SHARED_MEM_SIZE][SHARED_MEM_SIZE];

if ((z < ZS) && (y < YS) && (x < XS))
{
    int local_value = inImg[z * ozs + y * XS + x];
    shared[ty + 1][tx + 1] = local_value;
```

Fig. 13: Declaració memòria compartida

```
if (threadIdx.x == 0)
{
    shared[ty + 1][0] = (x > 0) ? inImg[z * ozs + y * XS + (x - 1)] : 0; // Esquerra
}
else
{
    if (threadIdx.x == blockDim.x - 1)
        shared[ty + 1][blockDim.x + 1] = (x < XS - 1) ? inImg[z * ozs + y * XS + (x + 1)] : 0; // Dreta
}

if (threadIdx.y == 0)
{
    shared[0][tx + 1] = (y > 0) ? inImg[z * ozs + (y - 1) * XS + x] : 0; // Part superior
}
else
{
    if (threadIdx.y == blockDim.y - 1)
        shared[blockDim.y + 1][tx + 1] = (y < YS - 1) ? inImg[z * ozs + (y + 1) * XS + x] : 0; // Part inferior
}
__syncthreads();
```

Fig. 14: Control dels límits del bloc i sincronització dels threads

Si ens fixem en la figura 14 es veu que estem utilitzant l'operador ternari (? :), que és una forma concisa d'escriure una condició if-else en una sola línia. Si la condició que es troba després de l'igual es compleix, s'assignarà el primer valor després de l'interrogant. Per contra, si no es compleix, s'assigna el valor després dels dos punts. Com CUDA és capaç d'interpretar aquest operador i es veu més compacte i més net que un if-else hem optat per fer-lo servir. I per acabar amb aquesta figura, l'última instrucció, que és la primitiva __syncthreads(), que ja hem vist perquè serveix en un dels apartats anteriors, ens assegura que tots els fils dins del mateix bloc esperin al fet que l'últim d'ells acabi. De manera que un cop l'últim thread hagi executat aquesta funció, tots els altres fils podran continuar executant el que queda del kernel, però ens haurem assegurat que la memòria compartida estigui ben inicialitzada.

I per acabar amb la subsecció ens queda veure, ho farem en la figura 15, com hem modificat la resta del kernel, és a dir, la part on fem l'operació de closing. El que hem canviat és el càlcul dels indexs lx i ly que fem servir per a accedir als píxels veïns. Ara com hem de passar a un nivell local i no global hem substituït els valors x i y per tx i ty respectivament. Després, també ens ha tocat modificar la comparació perquè comprovi amb els límits de la memòria local i no els de tota la imatge. I per últim, l'accés a la memòria global es substitueix per l'accés a la memòria compartida.

Pel que fa a la reserva de memòria, execució del kernel i la transferència de dades en CUDA segueix exactament igual que en la primera versió. Si haguéssim fet servir la memòria dinàmica i no l'estàtica algun paràmetre hauria tocat modificar o agregar, però com no és el cas no hi entrarem en detall.

```
for (int k = 0; k < 5; k++)
{
    int vi = 0;
    int aux_x = sex[k];
    int aux_y = sey[k];

    for (int i = 0; i < 5; i++)
    {

        int lx = tx + sex[i] + aux_x;
        int ly = ty + sey[i] + aux_y;


        //Comprovar limits per a lx i ly
        if (lx >= 0 && lx < SHARED_MEM_SIZE && ly >= 0 && ly < SHARED_MEM_SIZE)
        {
            // Accedir a la memòria compartida en més de a la global
            int possrc = shared[ly][lx];
            vi |= possrc;
        }

    }
    vo &= vi;

}
```

Fig 15: Closing amb memòria compartida

7.3 OpenCL C

Quan hem parlat al principi de la secció de la memòria compartida fèiem sobretot referència a CUDA, la terminologia era la seva. Pel que fa a OpenCL el terme memòria compartida no existeix, el que existeix és la memòria local. Però a efectes pràctics és el mateix.

La memòria local és accessible per a tots els work-items dins d'un work-group i així per a cada work-group. També s'ha de tenir en compte el fet de la sincronització on la primitiva no és la de __syncthreads(), sinó barrier(CLK_LOCAL_MEM_FENCE). Que bàsicament, fa que els work-items dins d'un work-group no puguin avançar fins que l'últim d'ells executi la funció.

Un cop vistes les petites diferències en CUDA, a l'hora de la implementació sí que hi ha canvis.

Aquests canvis a part de les diferències que ja vàrem veure en la primera versió, la memòria local es passa com a paràmetre del kernel, com si en CUDA utilitzéssim la memòria dinàmica. I també hi ha el canvi de com obtenim l'identificador local, tx i ty. En OpenCL per a l'identificador global no fa falta fer un càlcul com en CUDA, i per a l'identifacdor local hi ha una funció, get_local_id(), que ens el retorna. Ho podrem veure en la figura 16.

Després com podrem comprovar en la figura 17, l'operador ternari que hem vist en l'apartat anterior OpenCL no el soporta. Per tant, hem hagut de canviar-los per sentències if-else.

```
    const char* kernel_source_sh =
R"(__kernel void close(__global const char* inImg,
                       __global char* outImg,
                       const int XS, const int YS, const int ZS,
                       const int oxs, const int oys, const int ozs,
                         __local char shared[34][34])
    {

        int x = get_global_id(0);
        int y = get_global_id(1);
        int z = get_global_id(2);

        int tx = get_local_id(0);
        int ty = get_local_id(1);

        if ((z < ZS) && (y < YS) && (x < XS))
        {
            int local_value = inImg[z * ozs + y * XS + x];
            shared[ty + 1][tx + 1] = local_value;
```

Figura 16: Canvis local memory OpenCL i obtenció d'ids

```
            if(ty == blockDim_y - 1)
            {

                if(y < YS - 1)
                {
                    shared[blockDim_y + 1][tx + 1] = inImg[z * ozs + (y + 1) * XS + x];
                }
                else
                {
                    shared[blockDim_y + 1][tx + 1] = 0;
                }
            }
        }
    }

    barrier(CLK_LOCAL_MEM_FENCE);
```

Figura 17: canvis operador ternari i sincronització OpenCL

La resta del codi del kernel és exactament la mateixa que en CUDA.

I pel que fa al codi que no forma part del kernel només hem afegit una instrucció nova, per a afegir l'argument de la memòria local, que és la següent:

err = clSetKernelArg(kernel, 8, sizeof(char)*mida_sh_mem, NULL);

7.4 PyOpenCL

Pel que fa a OpenCL i python el kernel és exactament el mateix que en l'apartat anterior, l'únic que canvia és la manera en què es passen els arguments. La memòria compartida es passa de la següent forma:

kernel.set_args(inImg, outImg, np.int32(pixels_x), np.int32(pixels_y), np.int32(total_f), np.int32(1), np.int32(pixels_x), np.int32(total_pixels_per_imatge), cl.LocalMemory(local_mem_size))

8 FPGA

En aquesta sección veurem que és una FPGA i perquè s'utilitza. Posteriorment, veurem com s'ha implementat el codi per a executar el que abans era un kernel en una GPU. Com hem esmentat en la introducció farem servir OmpSs@FPGA que és una extensió del model de programació OmpSs-2 que suporta la descàrrega de tasques a un dispositiu d'aquest tipus.

Una FPGA o Field Programable Gate Array és un conjunt de circuits integrats que conté blocs de lògica, on la interconnexió i la funcionalitat pot ser configurada a través d'un model de programación o un llenguatge, per exemple Verilog o VHDL [15].

L'avantatge de les FPGAs respecte d'altres circuits integrats que estan fets per a realitzar una única funció és que aquestes són reprogramables [15].

I pel que fa a les GPUs, en capacitat de còmput aquestes són molt millors, però això es reflecteix en el consum d'energia on les FPGAs fan un ús més eficient.

8.1 Implementació amb OmpSs@FPGA

En aquesta subsecció veurem com s'ha realitzat la implementació del nostre programa amb OmpSs per a FPGA.

Com hem vist en la introducció OmpSs és un model de programació desenvolupat pel BSC (Barcelona Supercomputing Center) amb l'objectiu d'estendre OpenMP en

noves directives per a suportar paral·lelisme asíncron i heterogeneïtat.

El primer que hem de fer és treballar sobre la versió seqüencial del programa, la que hem vist amb anterioritat que treballa amb tres fors anidats, i després treballa sobre l'element estructurador.

Partint d'aquest punt la primera cosa que hem de fer és definir els elements d'entrada i de sortida de la funció, amb la mida inclosa, i especificar que la funció anirà destinada a una FPGA. Això ho fem posant les següents sentències a sobre de la declaració de la funció d'aquesta forma:

#pragma oss task in([TOTAL_PIXELS]inImg) out([TOTAL_PIXELS]outImg)

#pragma oss target device(fpga) copy_deps num_instances(1)

Un cop hem definit el bàsic podem especificar més sentències que ajudin a incrementar el rendiment. Una de les instruccions que podem fer servir és HLS UNROLL. Aquesta directiva provoca que el bucle on s'ha declarat es desenrotlli. El desenrotllament del bucle és una optimització que farà que es multipliqui el cos del bucle diverses vegades segons un factor x que podem especificar. En el nostre cas serà de cinc, és a dir, tot el bucle. Això redueix el nombre d'iteracions i potencialment pot millorar el rendiment del programa.

Per tant, aquesta directiva l'hem posat sobre els dos bucles més interns, els que es recorren l'element estructurador.

Després també existeix la possibilitat d'utilitzar la instrucción HLS ARRAY PARTITION variable TYPE factor dim. La sentència permet dividir un array en diverses submatrius més petites, dividint-les en altres memòries, cosa que permet un accés més paral·lelitzat i eficaç. En el nostre cas el tipus que fem servir és la partició cíclica. Aquesta partició divideix l'array en blocs cíclics més petits, distribuïts uniformement entre diverses memòries. Això vol dir que els elements de l'array es distribueixen en aquestes memòries de manera alternada segons un patró cíclic, la qual cosa pot millorar significativament el rendiment de l'aplicació en termes de latència i amplada de banda. El valor factor específica de quants elements ha de ser la partició cíclica. Hem posat la directiva en el bucle més intern de tots. S'ha de dir que existeixen altres tipus de particions, la partició en bloc i la completa, però com per al nostre cas la que s'adapta millor és la partició cíclica, no entrarem en detall en les altres.

I també hi ha l'opció de fer servir la directiva HLS pipeline II. On definim a través del valor II el nombre de cicles que han de passar entre dues iteracions consecutives. És a dir si II és igual a u, pot començar una nova iteració a cada cicle de rellotge. Cosa que també hem afegit en el for més intern de tots.

9 RESULTATS

Un cop vistes les diferents versions del programa amb les seves respectives implementacions i les eines o els llenguatges que s'utilitzen per a optimitzar, podem passar a veure els resultats i fer un anàlisi, per a posteriorment treure unes conclusions.

Primer de tot en la taula 4 podem veure que també hem fet servir l'eina openMP per a veure com és la millora a nivell de CPU utilitzant més d'un thread. Podem observar que l'speedUp és de 3'6 respecte la versió seqüencial quan utilitzem vuit threads.

En les versions de CUDA executades en la primera GPU hem pogut treure un perfilatge del codi molt més detallat gràcies al programa Nsight Compute propi d'NVIDIA, com apreciarem en la taula 1.

En la taula podem veure dues versions, la primera no utilitza memòria compartida, en canvi, la segona versió si que ho fa.

També podem veure els detalls de la memòria per a la versió u en la taula 2 i de la versió dos en la taula 3.

A banda de les mètriques mostrades en les taules anteriors, també podem veure l'anàlisi en concret de les transferències entre els diferents tipus de memòria de la GPU. La captura en qüestió es mostra a la figura 18 i a la figura 19.

En l'execució en CUDA aquesta manera d'agrupar els threads i del grid ens dona speedup de 47'11. L'únic que passa és que amb aquesta GPU ja esmentada, no podem arribar a posar més threads/block, ja que el màxim és de 1024 al qual hem arribat. I segons l'eina per a perfilar aquest és l'òptim. Tot i ser un speed up considerable si entrem en el detall de l'anàlisi i no ens quedem amb el màxim de threads que comentàvem, podem veure que el memory throughput és d'un 23'25 GBytes/segon. I que la quantitat de memòria que es mou entre la memòria del dispositiu i la caché L2 és de 5'23 GB en el cas de lectura i de 15'89 GB per a les escriptures. Si mirem el codi, podem veure que en el dos bucles anidats que ambdós iteren 5 vegades, és a dir, un total de vint-i-cinc cops, hi ha una lectura al vector que guarda tots els píxels. Si cada un dels threads accedeix a la memòria global un total de vint-i-cinc cops ens surten aproximadament aquests 5'23 GB (202'11 MB*25 = 5.052 GB).

També el profiler ens diu que 34% dels accessos són no fusionats i que hi ha una possibilitat de fer speedUp. Els accessos no fusionats son accessos que ocorren quan els fils accedeixen a la memòria global de manera dispersa o no alineada. Com a resultat, cada accés a la memòria ha de ser tractat individualment en lloc de ser combinat. Això pot resultar en múltiples operacions de memòria, disminuint així el rendiment i l'eficiència.

D'aquests problemes esmentats anteriorment el següent pas ha sigut la implementació de la versió número dos. Aquesta versió implementa l'ús de memòria compartida. Aquesta memòria s'encarregarà, en la majoria dels casos, de fer un únic accés per thread a la memòria global i guardar per a cada bloc els píxels corresponents. D'aquesta manera podem evitar les vint-i-cinc lectures a la memòria global i fer-les a la memòria compartida, reduint així el temps d'accés i la reutilització de les dades. Això fa que tant l'IPC com el compute throughput sigui major que la versió anterior. Millorant el temps d'execució. També podem veure que el número d'accessos a la memoria global s'han reduit i que aquests han passat a ser peticions a la memòria compartida.

Una possible millora segons Nsight Compute podria ser

la d'intentar equilibrar la càrrega entre les diferents "slices" de la memòria cau de nivell 2 (L2), ja que ens diu que un "slice" te fins un 18'62% més de cicles actius que la mitjana, mentre que d'altres tenen fins un 7'75% menys.

En la taula 4 veurem els resultats de totes les execucions i la diferència que hi ha.

Pel que fa a l'execució de totes les versions, taula 4, aquestes són en la GPU NVIDIA GEFORCE RTX 2080 TI, que és una GPU més potent que l'altra. Com podem veure l'speedUp respecte la versió seqüencial en el cas de CUDA és de 678'53. Aquesta és una millora molt significativa respecte a la versió original, i també respecte la primera GPU utilitzant CUDA, que suposa un increment de 22'69.

Com podem veure la versió que va més ràpid és la que utilitza CUDA, seguida per la que utilitza C i openCL i per últim la que utilitza Python i openCL. Això es deu a que la versió en CUDA és la que tarda menys a l'hora de copiar les dades des de la memòria del host al dispositiu, amb bastant diferència que les altres, uns 20 ms. El kernel també és el més ràpid de tots, uns 1'325 ms, i per al pas de dades del dispositiu al host no és el més ràpid, però la diferència amb C OpenCL no arriba ni a 1 ms. Tot seguit, podem veure com aquesta última versió esmentada pel que fa a l'execuicó del kernel és practicament igual a la versió implementada amb Python. On existeix una diferència bastant significativa és en la transmissió de dades en amdues direccions, la implementació amb Python li costa uns 20 ms segons més copiar les dades de l'amfitrió al dispositiu i 33'638 ms més en fer la còpia en l'altre sentit.

10 Conclusió

Un cop vistos els resultats i haver fet una anàlisi, podem intentar extreure conclusions.
Pel que fa a les execucions en CUDA, podem afirmar que, donada la quantitat de dades i les característiques del nostre problema, la segona GPU proporciona avantatges significatius en termes d'ample de banda, més SMs, memòria GDDR6, entre altres. No obstant això, l'execució en CUDA amb memòria compartida és més ràpida en la primera GPU i més lenta en la segona a causa de les instruccions addicionals necessàries per gestionar aquesta memòria.

Pel que fa a les execucions amb la segona GPU, podem considerar que la versió amb CUDA és la més ràpida de totes gràcies a l'optimització específica per a la propia GPU de NVIDIA i la mateixa arquitectura, especialmente en la transferència de dades i l'execució del kernel. Després, en les versions amb OpenCL el rendiment en l'execució del kernel podriem dir que és el mateix, però hi ha una diferència molt gran i que afecta molt al temps total d'execució de la versió amb Python, que és la transferència de dades. Per tant, utilizar Python amb OpenCL pot ser que no sigui la millor idea, ja que Python al ser un llenguatge interpretat afegeix un cert overhead adicional durant l'execució afectant negativament a la transferència. I també hi ha un cost associat entre el binding de les llibreries de baix nivell com OpenCL a la interfície que utilitza Python.

Però com podem observar en la Taula 4, l'execució amb FPGA i openCL no arriba a ser tan eficient com en les GPUs, ja que aquesta no ens ofereix tan capacitat de còmput. Tot i que aquesta diferència de capacitat es veu reflectida en la diferència del consum d'energia que pot arribar a ser un aspecte important.

Tot i no haver pogut provar d'executar la versió amb FPGA amb OmpSs quedo bastant satisfet en el desenvolupament del treball i els resultats obtinguts, ja que he la versió s'ha desenvolupat igualment, però no s'ha pogut provar per falta de temps.

A nivell personal he aprés a utilitzar amb molt més detall CUDA. I pel que fa a OpenCL i OmpSs@FPGA són dos eines/frameworks que he hagut d'aprendre de zero i que m'han semblat molt satisfactòries.

Per a acabar, m'agradaria agrair al meu tutor, David Castells, l'ajuda que m'ha ofert. Sense ell, això no hauria estat possible.

Bibliografia

[1] Castells-Rufas, D., Sánchez, C., Gil, D., & Carrabina, J. (2021). Accelerating Morphological Operations with Heterogeneous Computing Platforms in Python.

[2] B. Jähne, Digital Image Processing. [Llibre]. 5a ed., Springer, 2005.

[3] NVIDIA Corporation, "What is CUDA?". [En línia]. Disponible: https://blogs.nvidia.com/blog/what-is-cuda-2/

[4] Khronos Group, "OpenCL Overview". [En línia]. Disponible: https://www.khronos.org/api/opencl

[5] Barcelona Supercomputing Center, "OmpSs". [En línia]. Disponible: https://pm.bsc.es/ompss

[6] Zheng, S., Guo, J., Cui, X., Veldhuis, R.N., Oudkerk, M., Van Ooijen, P.M.: Automatic pulmonary nodule detection in CT scans using convolutional neural networks based on maximum intensity projection. IEEE transactions on medical imaging 39(3), 797–805 (2019)

[7] Maier, O., Spann, S.M., Bödenler, M., Stollberger, R.: Pyqmri: an accelerated python-based quantitative MRI toolbox. Journal of Open Source Software 5(56), 2727 (2020)

[8] Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., Fasih, A.: Pycuda and pyopencl: A scripting-based approach to GPU run-time code generation. Parallel Computing 38(3), 157–174 (2012)

[9] Garcia-Uceda, A., Selvan, R., Saghir, Z., Tiddens, H.A., de Bruijne, M.: Automatic airway segmentation from computed tomography using robust and efficient 3-D convolutional neural networks. Scientific Reports 11(1), 1–15 (2021)

[10] NVIDIA Corporation, "Nsight Systems 2023.2". [Programari d'ordinador], 2023. [En línia]. Disponible: https://developer.nvidia.com/nsight-systems

[11] NVIDIA. (2018, setembre 14). NVIDIA Turing architecture in-depth. Recuperat de https://developer.nvidia.com/blog/nvidia-turing-architecture-in-depth/

[12] TechPowerUp. (n.d.). GeForce RTX 2080 Ti Specs. Retrieved from https://www.techpowerup.com/gpu-specs/geforce-rtx-2080-ti.c3305

[13] Xilinx Inc., "Xilinx Alveo U200 Data Center Accelerator Card Specifications". [En línia]. Disponible: https://www.xilinx.com/products/boards-and-kits/alveo/u200.html#specifications

[14] Harris, M. (2013, January 28). Using shared memory in CUDA C/C++. NVIDIA Developer Blog. https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/

[15] Geeknetic. (n.d.). ¿Qué es una FPGA y para qué sirve? Recuperado de https://www.geeknetic.es/FPGA/que-es-y-para-que-sirve

APÈNDIX
A1. Resultats cuda GPU 1

Versió	Time	Compute Throughput %	Memory Throughput Gbyte/sec	Executed IPC Active
V1	908'78 ms	59'46	23'25	1'50
V2	807'43 ms	72'30	8'26	1'83

Taula 1: Resultat execucions CUDA

Mem Throughput [Gbyte/s] 8,26	Mem Busy [%] 5'33
L1 Hit Rate [%] 87'08%	Max Bandwidth [%] 7'75
L2 Hit Rate [%] 95'12%	Mem Pipes Busy [%] 7'75

Taula 2: Resultats memòria CUDA V1

Mem Throughput [Gbyte/s] 23'25	Mem Busy [%] 15'26
L1 Hit Rate [%] 60'26	Max Bandwidth [%] 20'81
L2 Hit Rate [%] 88'95	Mem Pipes Busy [%] 9'36

Taula 3: Resultats memòria CUDA V2

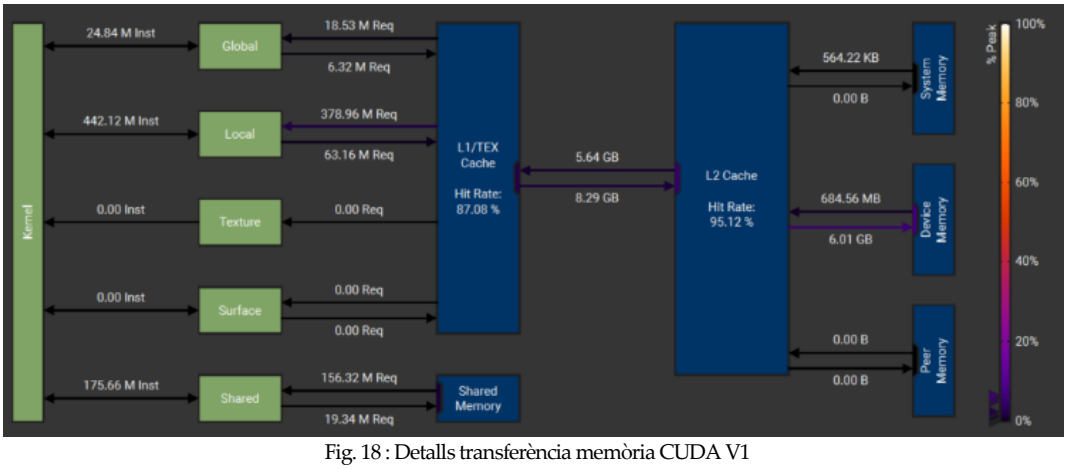


Fig. 18 : Detalls transferència memòria CUDA V1

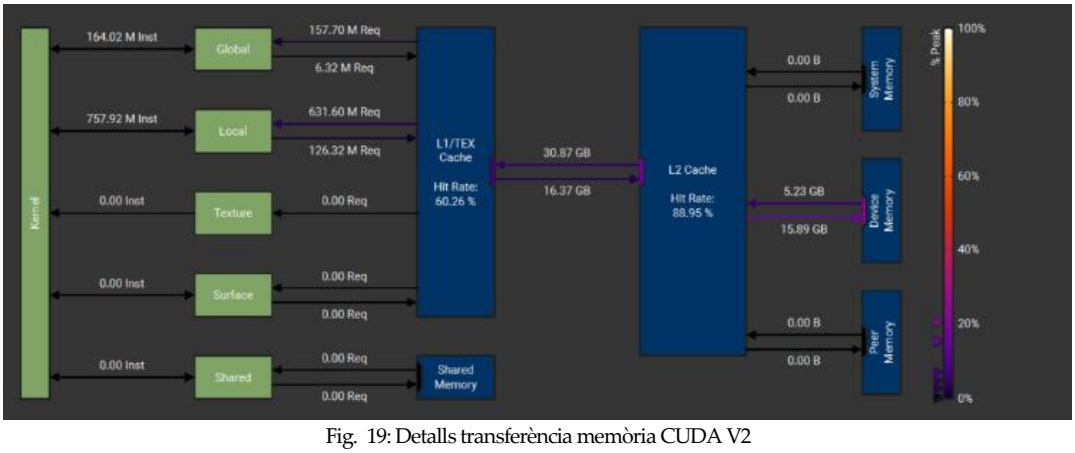


Fig. 19: Detalls transferència memòria CUDA V2

A2. Resultats amb GPU 2

Versió	GridSize	BlockSize	Tr In	Kernel	Tr out	**Temps total**
Seqüencial	-	-	-	-	-	**29,960s**
OMP	-	-	-	-	-	**8,280s**
CUDA	16, 16, 771	32, 32, 1	21,073ms	4,655ms	18,426ms	**44,154ms**
CUDA_SH	16, 16, 771	32, 32, 1	22,009ms	6,776ms	18,350ms	**47,135ms**
C_opennCL	16, 16, 771	32, 32, 1	48,066ms	6,014ms	17,654ms	**71,734ms**
C_openCL_SH	16, 16, 771	32, 32, 1	47,812ms	6,880 ms	17,659ms	**72,351ms**
PyCL	16, 16, 771	32, 32, 1	68,367ms	6,007ms	51,292ms	**125,666ms**
PyCL_SH	16, 16, 771	32, 32, 1	52,103ms	8,273ms	52,103ms	**112,479ms**
FPGA_openCL	-	-	-	-	-	**0,89 s [1]**

Taula 4: Resultat execució de totes les versions en la GPU NVIDIA GeForce RTX 2080 Ti