Anàlisis de paràmetres d'optimització d'aplicacions CUDA

Roger Torras Aguilera

Resum– En els últims anys s'ha anat fent més important l'optimització dels recursos informàtics per aconseguir un millor rendiment.En aquest article estudiarem diferents paràmetres que podran ajudar a millorar les aplicacions CUDA obtenint una millor eficiència dels recursos de la GPU. Per aconseguir aquest objectiu s'ha estudiat l'arquitectura de les GPUs i mitjançant una recerca s'han proposat diferents aspectes que milloraran el rendiment. S'ha estudiat en quines situacions es preferible l'us de les GPU davant les CPU i també en quines situacions no es favorable. Posteriorment s'ha realitzat diferents experiments on hem demostrat experimentalment els fets proposats anteriorment i hem extret les conclusions les pertients.

Paraules clau– GPU, CUDA, HPC, aplicacions paral·leles, anàlisi de rendiment

Abstract– In recent years, it has become increasingly important to optimize computer resources in order to achieve better performance. In this paper we study different parameters that can help CUDA applications to obtain a better efficiency of GPU resources. To achieve this goal, the architecture of GPUs has been studied and through an investigation different aspects that improve performance have been proposed. It has been studied in which situations the use of GPUs is preferable over CPUs and also in which situations it is not favorable. Subsequently, different experiments have been carried out in which what has been proposed above has been experimentally demonstrated and the conclusions have been extended.

Keywords– GPU, CUDA, HPC, optimitzation, parallel applications, performance analysis

1 Introducció

El HPC (High-Performance Computing) [1] es utilitzar la potencia de còmput per resoldre problemes complexos que estan dividits per diferents fils/processos que s'executen concurrentment en clústers o supercomputadors. Per poder ser capaços de resoldre problemes molt complexos en temps raonables, l'eficiència dels recursos de la màquina passa a ser un factor clau per aconseguir l'objectiu. Les aplicacions paral·leles són aquelles que donat un flux de treball el divideixen en tasques més petites que es poden executar independentment una de les altres. En el món de la paral·lelització podem trobar dos grans paradigmes, els sistemes amb memòria compartida on tots els nodes del sistema comparteixen el mateix espai de memòria i els sistemes amb memòria distribuïda, on la memòria no es troba centralitzada en un punt i es troba repartida entre els diferents nodes del sistema. Un factor clau son les GPU, que es un processador que esta format per molts nuclis (coneguts com GPU cores) petits i especialitzats. Els clústers o supercomputadors heterogenis (CPU/GPU) permeten elaborar dos nivells de paral·lelització, primer ens permet dividir un procés en varis threads o processos que s'executaran en diferents CPU i llavors cada CPU permet enviar el codi a la seva GPU. Per tant, quan es treballa amb gran quantitats de dades l'ús de clústers o supercomputadors i GPUs pot ser una bona opció per reduir temps de còmput gràcies al seu gran potencial de paral·lelitzar els algorismes [2].

El motiu pel qual es dura a terme aquesta recerca sobre els paràmetres d'optimització d'aplicacions CUDA i aquest anàlisis sobre el funcionament de les GPU es degut a què cada vegada mes en el mon del HPC s'estan utilitzant mes les GPU, a mes a mes, també en l'àmbit comercial son cada vegada mes comunes perquè aplicacions com els videojocs les usen per obtenir un millor rendiment i una millor qualitat d'imatge.

Hi ha diverses opcions d'implementar codi en les GPUs, hi ha aplicacions científiques que requereixen una gran

• E-mail de contacte: roger.torras@uab.cat
• Menció realitzada: Enginyeria de Computadors
• Treball tutoritzat per: Anna Bàrbara Sikora (Departament d'Arquitectura de Computadors i Sistemes Operatius)
• Curs 2021/22

quantitat de capacitat de còmput i per tant s'ha de realitzar un anàlisis molt exhaustiu del codi per determinar quines son les parts que s'han d'executar a la GPU i quines parts en la CPU per així reduir el màxim el overhead que genera el fet d'utilitzar CPU i GPU, com per exemple, el fet de copiar estructures de dades CPU a GPU i GPU a CPU.
En aquest treball ens centrarem en estudiar quins son els elements d'un programa que poden fer millorar o empitjorar el rendiment d'una aplicació executada en una GPU. Ens basarem en les aplicacions que estan escrites en CUDA per poder estudiar i analitzar el comportament de la GPU, tant estudiant els diferents paràmetres de CUDA com estudiant la diferència entre diferents GPUs.

L'objectiu principal es buscar els paràmetres d'optimització de les aplicacions CUDA. Inicialment fent recerca sobre diferents camins per optimitzar una aplicació CUDA i per tant veure diferents elements que ens poden ajudar a optimitzar una aplicació.
El treball seguira la següent metodologia: agafarem uns kernels implementats en CUDA, analitzarem tot el codi per saber com esta organitzat i saber exactament que fa, aplicarem les diferents possibilitats per optimitzar una aplicació CUDA i realitzarem diferents experiments (paral·lelitzar diferents parts, utilitzar diferents estratègies, variar la dimensió de grid).

El paper està organitzat de la següent manera. La Secció 2 descriu quin és el background del treball. La Secció 3 presenta diferents paràmetres d'optimització per la GPU. La Secció 4 presenta els kernels usats en la part pràctica. La Secció 5 presenta les proves realitzades estudiant els paràmetres de la Secció 3 en els kernels de la Secció 4. La Secció 6 presenta les conclusions extretes del treball realitzat.

2 Background

Abans de començar a entrar en matèria cal primer repassar uns termes que seran necessaris per entendre el context en el que es basa el treball. En primer lloc hem de parlar de la programació paral·lela. Es tracta de donat un problema dividir-lo en tasques mes petites que es puguin dur a terme simultàniament [3]. Per tant es molt important analitzar correctament el codi per poder identificar quines parts es poden executar en paral·lel i quines no. En segon lloc hem de parlar dels supercomputadors, que es una gran computadora que té una potencia i una velocitat de còmput molt elevades [4]. Per altre banda també hi ha els clústers de computadors, que son una sèrie de computadors que estan connectats entre si on la imatge que te l'usuari es d'una supercomputadora però internament es tracta de múltiples computadores [5]. El principal punt a favor que es ofereix un clúster davant un supercomputador es que es tracta d'un sistema distribuït on no depenem d'un únic node que realitzi tot el treball, en cas de falla en un clúster tenim altres dispositius on la nostra aplicació es pot executar.
Havent fet un petit repàs sobre les diferencies entre un supercomputador i un clúster ara cal seguir parlant de la programació paral·lela, que es pot dur a terme tant en una CPU com en una GPU, cosa que ens porta a estudiar l'arquitectura de cada una, ja que hi ha moltes diferències. L'arquitectura actual d'una CPU es anomenada multicore i l'arquitectura d'una GPU es anomenada manycore.
Actualment la majoria de les CPU disposen de mes d'un nucli (dual-core, quad-core, hexa-core fins a 32 nuclis, AMD Ryzen Threadripper 3970X) que permeten executar diferents processos en cada un de forma independent, per tant en una mateixa CPU es poder executar, en funció de la seva tecnologia, fins a 32 processos concurrentment [6]. També hi ha la possibilitat que un mateix core pugui tenir mes d'un thread, anomenat el hyperthreading. Un exemple pot ser l'últim processador d'intel , intel i7 11700KF [7], que conté 8 nuclis i té capacitat fins a 16 threads simultanis. Les CPU tenen memòries cache molt grans, unitats de control i ALUs molt sofisticades per poder maximitzar al màxim l'execució en sèrie dels programes.
Per altre costat les GPU disposen d'unes ALU molt simples, d'una memòria cache molt petita i unitats de control simples per així maximitzar l'execució en paral·lel de les aplicacions. Per exemple la última GPU de nvidia, la nvidia GeForce RTX 3080 Ti [8], conté 10240 cuda cores per a la execució de 10240 threads simultàniament. Per la organització correcta dels threads, les GPUs s'organitzen de forma jeràrquica on cada GPU te diferents Graphic Processing Cluster(GPC), on dintre de cada GPC hi ha diferents Texture Processing Clusters(TPC) i dins de cada TPC hi ha diferents Streaming Multiplocessors(SM) que es on hi ha els diferents cores per executar les instruccions (figura 2). Cada SM conte una memòria compartida i uns registres. Quan es programa una GPU s'ha de tenir en conte que els diferents cores d'un SM han d'executar en el mateix cicle de rellotge les mateixes instruccions, en cas de no complirse s'executarà tot en sèrie.

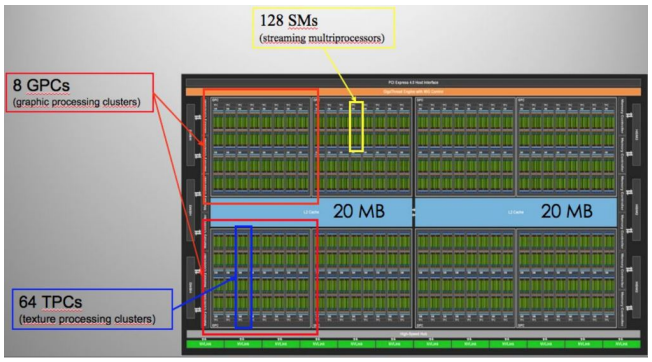

Fig. 1: Exemple d'estructura de GPU

Un cop analitzada l'estructura d'una GPU i havent vist que el seu funcionament es molt diferent que el d'una CPU hem de veure quins son els llenguatges de programació que tenim per poder programar en una GPU. Actualment existeixen varis llenguatges però aquí parlarem de 3 dels mes utilitzats: CUDA, OpenCL i OpenACC.
CUDA es un llenguatge de programació que es basa amb c/c++, on a partir d'un codi en c o c++ ens permet afegir unes extensions i així poder crear funcions que s'executen a la GPU podent decidir una sèrie de components com per exemple quants threads necessitem i com els volem distribuir dintre de la GPU [9].
OpenCL es una alternativa que permet programar tant en

CPU com en GPU i te una gran avantatge respecte CUDA, i es que es multi plataforma i esta suportat per IOS. Comparant CUDA i OpenCL en GPUs de NVIDIA, les proves de rendiment mostren que CUDA es bastant mes eficient que OpenCL [10]. En la referència [11] es poden trobar diferents exemples de OpenCL.
OpenACC es un estàndard de programació que permet que sobre un codi en c, c++ o Fortran escriure-hi una sèrie pragmes que et permeten adaptar aquell codi perquè es pugui executar en una GPU [12]. Esta dissenyat per simplificar la programació i l'adaptabilitat de codis que nomes s'executen en CPU a que es puguin executar en GPU.

```
#pragma acc data copyin( a[0:n], b[0:n] ), copyout( c[0:n] )
{
   #pragma acc kernels
   {
      #pragma acc loop independent
      for ( i =  0;  i < n; i++ ) {
         c[i] = a[i] + b[i];
      }
   }
}
```

Fig. 2: Exemple de codi c++ amb OpenACC

Com hem pogut comprovar l'arquitectura d'una GPU es molt diferent de l'arquitectura d'una CPU per tant requereix d'un estudi molt exhaustiu per poder entendre clarament el seu funcionament i poder ser capaços de trobar estratègies i distribucions de codi que permetin maximitzar la seva eficiència.

3 RENDIMENT GPU

L'objectiu es centra en trobar diferents elements que puguin millorar el rendiment de les aplicacions CUDA i estudiar en experiments pràctics sota quines circumstàncies podem aplicar cada paràmetre.
En primer lloc s'ha realitzat un treball de recerca per tal de poder establir les bases sobre les que treballarem i s'han consultat diferents papers però bàsicament ens centrarem en un [13], que tracta de diferents elements que afecten el rendiment de les GPU, que ens servirà per establir les bases del nostre treball. Amb l'objectiu d'establir una bona estructura per començar s'ha elaborat una llista amb diferents paràmetres de la GPU que poder ser rellevants en el rendiment.

3.1 Característiques CUDA

Abans d'entrar en matèria amb la llista de paràmetres és convenient tenir present les característiques de CUDA. Les aplicacions CUDA necessiten que el programador especifiqui l'espai de memòria que utilitzarà. Les GPUs disposen de molts threads, i per organitzar-se usen blocs. Cada bloc pot tenir fins a 1024 threads i la mida mínima és d'1 thread. Encara que CUDA permeti blocs de menys de 32 threads no és recomanable fer-ne ús perquè l'arquitectura de les GPU estableix que els warp són de 32 threads, un warp és la unitat mínima que té una GPU on tots els threads que conté han d'executar la mateixa operació, i si no es compleix aquest requisit, s'executaran totes les instruccions en sèrie. Per tant, si fem servir blocs de menys de 32 threads podrem generar que hi hagi certes operacions que s'executin en sèrie. L'espai de memòria a utilitzar s'especifica en la crida del kernel, on s'envien dos paràmetres on el primer és el nombre de blocs i el segon el número de threads que conté cada bloc. En la figura 3 veiem que enviem dos paràmetres, en la segona posició especifiquem que volem blocs de 1024 threads i en primera posició fem una petita operació perquè quan dividim la mida del problema entre els 1024 threads sempre doni un resultat arrodonit cap a un número superior, per exemple, 2059/1024 és 2,01, per tant, necessitem 3 blocs de 1024 per tractar totes les posicions de memòria.

```
Kernel_Reduction_OPT<<<(N+1024-1)/1024, 1024>>>(d_a1, d_res, N);
```

Fig. 3: Exemple crida a un kernel

Amb l'espai de memòria definit correctament ara només cal que en el kernel es calculi l'índex de cada thread, això es fa utilitzant l'índex que té cada bloc i l'índex que té assignat cada thread dins del seu bloc, en podem veure un exemple a la figura 4.

```
unsigned int tid = blockIdx.x * blockDim.x + threadIdx.x;
```

Fig. 4: Exemple de càlcul de l'índex dins d'un kernel

3.2 Opcions de paral·lelització

És molt important saber en quines situacions és favorable l'ús de la GPU i en quines altres situacions la GPU no és la millor opció.

La GPU es desenvolupa bé en casos on el patró d'accés a memòria és seqüencial, perquè permet fàcilment distribuir les dades entre els diferents cores. En cada core es desenvolupa un gran còmput de dades i implica copiar-hi totes les dades necessàries, fet que comporta que per operacions petites el temps de còpia és superior al temps d'execució. També es desenvolupa be en aplicacions que requereixin un gran ample de banda de dades ja que al tractar-se d'una arquitectura manycore on els cores estan dividits en diferents grups que permeten poder dividir fàcilment les tasques a realitzar i a l'hora proporcionar un bon ample de banda.

Per altre banda la GPU no és favorable en situacions on el nombre d'iteracions no és conegut, degut a que com per cada iteració d'un bucle s'executarà en un core diferent si la mida del problema no és coneguda no podrem crear el grid necessari per afrontar el problema. Un altre factor són les operacions condicionals dintre dels bucles, que no permeten poder paral·lelitzar correctament l'aplicació perquè, com hem comentat en el punt 3.1, totes el threads de un warp han d'executar la mateixa instrucció, i si no es compleix es serialitza tot el warp. Finalment, també és molt important evitar els accessos a memòria que són a posicions no contigües, perquè llavors es farà molt complicat poder dir a cada thread de la GPU a quina cel·la de memòria ha d'accedir per procedir amb la seva tasca.

3.3 Dimensió grid

El primer paràmetre a tractar serà la mida del grid ja que una mala elecció de la dimensió del bloc que es vol utilitzar o una mala distribució del grid pot suposar un baix rendiment de la GPU per culpa de què hi hagi threads que no estiguin executant res (que estiguin idle). Per tant, és un altre punt al qual hi hem de posar especial atenció, perquè un algorisme que sigui capaç d'usar més eficientment els recursos de la GPU serà capaç d'aconseguir millor temps d'execució.

3.4 Duplicitat de dades

En el punt 3.2 s'ha esmentat que la còpia de dades entre la CPU i la GPU pot afectar el temps d'execució, per tant, és important controlar exactament quines són les dades que es necessiten copiar d'un dispositiu a un altre. La duplicitat de dades implica que estem copiant elements que ja hi són, en conseqüència, estem malgastant temps i recursos, per tant, és un punt al qual es poden optimitzar algunes aplicacions.
En primer lloc, s'ha d'analitzar correctament cada kernel per veure les dades d'entrada que necessita i el resultat que ens proporciona, d'aquesta manera estarem reduint la majoria de les còpies no necessàries. Però, no ens assegurarem que aconseguim optimitzar les còpies de tots els kernels, perquè ens podem trobar el cas que de bon inici no es sàpiga quins blocs de memòria es modificaran i quins no, per tant, en aquest aspecte com no sabem els blocs de memòria que han estat modificats, els hauríem de copiar tots.
Per evitar aquest fet hem d'aconseguir la manera d'identificar quins blocs han estat modificats i quins no. Una bona opció és indexar tots els blocs en una taula hash i abans de fer la còpia comparar els índex de cada bloc i si l'índex coincideix vol dir que no ha estat modificat i no cal copiar-lo i si l'índex no coincideix vol dir que s'ha modificat i és necessari copiar-lo.

3.5 Sincronitzacions

En últim lloc, és important tenir en compte les sincronitzacions entre el host i el device. Quan la GPU està executant algun kernel ho fa de forma independent a la CPU, en canvi, sí que existeix un element de sincronització quan volem realitzar alguna còpia entre host i device o device i host. Llavors gestionar els moments quan dur a terme les còpies és un factor clau per buscar una millor eficiència de la GPU. Si enviem un kernel i tot just després posem la còpia del device al host, la CPU estarà esperant tot el temps que trigui la GPU a executar el kernel, per tant, podem aprofitar aquest temps per executar altres parts del programa independents del resultat de la GPU i així no tenir la CPU en espera.

Com podem veure a la figura 5 una bona distribució de les còpies ens permetrà una millora en el temps d'execució.

3.6 Punts de sincronització entre CPU i GPU

Com he comentat en el punt 3.5 els elements de sincronització poden influir en el temps final d'execució. Quan es fa una crida d'un kernel s'envia al GPU i es posa en el flux d'execució. En aquest punt l'execució del kernel és independent de la CPU. Tenim dues principals execucions que sincronitzen CPU amb GPU. En primer lloc, tenim les

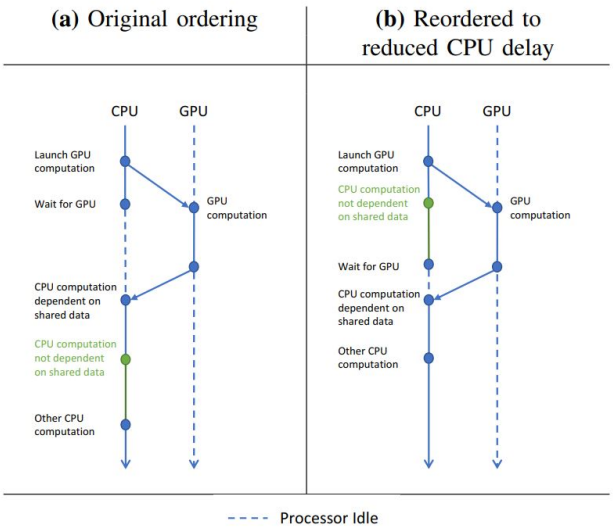

Fig. 5: Còpies Host-Device

còpies de memòria on es necessiten els dos dispositius sincronitzats per enviar/rebre dades. En segon lloc, tenim la instrucció *cudaDeviceSynchronize()* que crea una barrera en el flux d'execució de la GPU i posa en espera a la CPU fins que ambdues estan llestes.

4 Adaptació del codi inicial

Com a punt de partida, per poder realitzar proves, hem agafat un programa que està compost per diferents kernels que realitzen una funció especifica. Els kernels inicials estaven programats en c i no estaven adaptats per poder-se executar en GPU. Per tant, el primer pas a ha estat seleccionar els kernels que hem considerat que podien representar una millor mostra pels experiments i els hem programat amb CUDA.

4.1 Entorn de programació

Abans d'entrar en matèria és important establir l'entorn de programació en el qual es duran a terme totes les proves. Inicialment, estava previst utilitzar el clúster aolin o wilma de la UAB, que disposen d'unes NVIDA 3080Ti. Malauradament per culpa de problemes externs no ha estat possible utilitzar-los de forma regular, per tant, hem fet servir un sistema en local que disposa dels següents components: Intel(R) Core (TM) i7-10700 CPU @ 2.90 GHz, 32 GB de RAM i NVIDIA GeForce RTX 2060 SUPER, que disposa de 2176 cuda cores.

4.2 Kernel Copy

El primer kernel que hem agafat és un molt senzill on la principal tasca que fa és copiar els elements d'un vector a un altre, a la figura 6 veiem el codi inicial. A la figura 7 podem veure el resultat final del kernel on, en primer lloc, calculem en funció del grid creat, l'índex de cada thread i després cada thread fa la còpia.
En la figura 6 podem observar que la mida del problema es N2, això es degut a que N es la mida del problema, per tant

si en un kernel utilitzem 2 vectors, cada vector serà de N/2 elements.

```
for (j=0; j<N2; j++)
        c2[j] = b2[j];
```

Fig. 6: Codi inicial de Kernel_Copy

```
__global__ void  Kernel_Copy_CUDA ( float *b2, float *c2 )
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < N2) c2[i] = b2[i];
}
```

Fig. 7: Codi de Kernel_Copy

4.3 Kernel Scalar

El segon kernel escollit ha estat un que donat un vector d'entrada el que feia era multiplicar tots els elements del vector per un número escalar, com es pot veure a la figura 8. L'estructura escollida per programar el kernel ha estat com el primer kernel, a la figura 9 podem veure el resultat final.

```
for (j=0; j<N2; j++)
        c2[j] = scalar*b2[j];
```

Fig. 8: Codi inicial de Kernel_Scalar

```
__global__ void  Kernel_Scale_CUDA ( float *b2, float *c2, DATA_TYPE scalar )
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < N2) c2[i] = b2[i] * scalar;
}
```

Fig. 9: Codi de Kernel_Scalar

4.4 Kernel Add

El tercer kernel és una suma de dos vectors, on el resultat queda emmagatzemat en un tercer, com es pot veure a la figura 10. Un altre cop utilitzem l'estructura dels anteriors kernels per realitzar la suma. A la figura 11 es veu el resultat final.

```
for (j=0; j<N3; j++)
        f3[j] = d3[j]+e3[j];
```

Fig. 10: Codi inicial de Kernel_Add

```
__global__ void  Kernel_Add_CUDA ( float *d3, float *e3, float *f3)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < N3) f3[i] = d3[i]+e3[i];
}
```

Fig. 11: Codi del Kernel_Add

4.5 Kernel Reduce

Finalment, l'últim kernel que hem programat ha estat un kernel reduce, on donat un vector d'entrada es sumen tots els valors en un únic valor. En aquest cas concret hem elaborat dues versions diferents, perquè com veurem, aquest kernels ens presenta uns problemes molt concrets. El codi inicial el podem veure a la figura 12.

```
float res = 0.0;
for(int i = 0; i < N ; i++)
{
    res+=a1[i];
}
```

Fig. 12: Codi inicial de Kernel_Reduce

La primera versió que hem creat la podem veure a la figura 13. En aquesta versió, en primer lloc, creem uns vectors per cada bloc de la GPU on copiem el valor que li correspon a cada thread, llavors fem una operació de reduce en cada bloc de la GPU i finalment passem a tenir la suma de tots els valors assignats a aquell bloc en la posició 0 de cada bloc i mitjançant un atomic_add ho sumem tot. Aquesta operació final es realitza en serie, per tant, quants mes blocs tinguem mes tardarà, cosa que implica que ens podria interessar generar blocs de 1024 threads abans de blocs de 32 threads.

```
__global__ void  Kernel_Reduction_CUDA ( float *d_a1, float *d_res)
{
    __shared__ float sdata[1024];

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = d_a1[i];
    __syncthreads();

    // do reduction in shared mem
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }


    if (tid == 0) atomicAdd(d_res, sdata[0]);
}
```

Fig. 13: Codi de Kernel_Reduction versió 1

La principal problemàtica que hi ha a resoldre la versió anterior és que a mesura que es van fent iteracions el número de threads que utilitzem són menys perquè hi ha menys sumes a fer, per exemple, si de bon inici tenim N elements, en la primera iteració farem N/2 sumes, però en la segona iteració en farem menys perquè la nostra mida del problema passa a ser N/2 elements. A la figura 14 tenim una representació gràfica de com es realitzen les sumes dintre de cada bloc. Davant aquesta problemàtica s'ha de buscar la manera d'optimitzar el número de threads que usem.

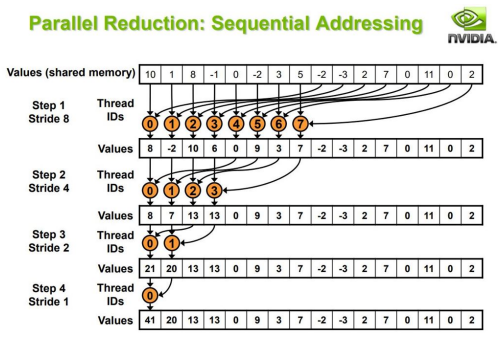

Fig. 14: Passos reduce

Llavors, com es pot observar a la figura 16 hem aplicat

un altre algorisme on fem ús de primitives d'intercanvi de dades de CUDA. En el primer for del kernel veiem que el valor inicial del for és l'índex de cada thread i que cada iteració se li sumen gridDim.x*blockDim.x.x, que són el número de threads de cada bloc. Per tant, cada thread del bloc tindrà la suma del seu valor mes la del mateix thread que ell de cada bloc. En el segon for el que es fa és fent ús de la primitiva __shfl__down_sync() que ens permet que un thread rebi informació d'un mateix thread dins d'un warp, on així tenim el resultat de cada warp en una variable. Finalment utilitzant atomicAdd fem la suma dels resultats de cada warp en un sol valor. En la figura 15 es veu un exemple de la reducció d'un vector usar atomicAdd.

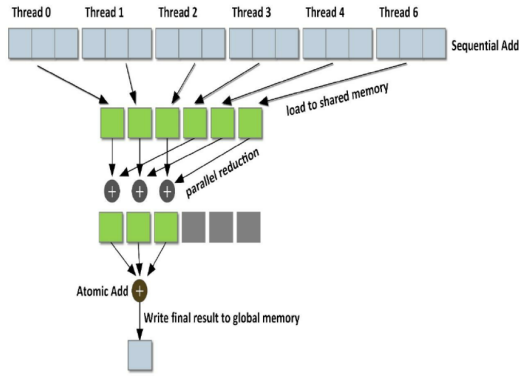

Fig. 15: Exemple atomicAdd()

```
__global__ void Kernel_Reduction_OPT( float *data, float *result, unsigned int n)
{
    unsigned int tid = blockIdx.x * blockDim.x + threadIdx.x;
    float accum = 0.0f;

    for(unsigned int i = tid; i < n; i += gridDim.x*blockDim.x)
    {
        accum += data[i];
    }
    for( int off = warpSize/2; off > 0; off/=2)
    {
        accum += __shfl_down_sync(0xffffffff, accum, off, warpSize);
    }

    if(tid%32 == 0) atomicAdd(result, accum);
}
```

Fig. 16: Codi de Kernel_Reduction versió 2

5 PART EXPERIMENTAL

Abans de començar a fer proves hem d'establir com farem les mesures de rendiment de la nostra aplicació (en particular de cada kernel i de les còpies de dades). Tenim dues opcions [14], en primer lloc, tenim la possibilitat de fer la crida a un kernel i just després posar un esdeveniment de sincronització i calcular el temps. Aquesta opció ens presenta un gran inconvenient i és que col·locant aquest mecanisme de sincronització el que estem fent és parar el flux de la GPU. Per altra banda tenim l'opció d'utilitzar els *cudaEvent_t*, el que fan és col·locar un fragment de codi que a l'executar-se la GPU ho reconeix i no para el flux d'execució. Com podem veure a la figura 17 al principi creem dos esdeveniments i després amb un esdeveniment agafem el temps inicial i amb l'altre el temps final i mitjançant *cudaEventElapsedTime()* ens calcula la diferència de temps entre els dos esdeveniments i ho guarda en una variable.
Utilitzarem els events de cuda perquè ens garanteixen que no es talla el flux d'execució de la GPU i ens permet calcular el temps amb CUDA, no necessitem funcions o llibreries externes.

```
void __attribute__((noinline)) Kernel_Scale( int k, DATA_TYPE scalar )
{
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaEventRecord(start);
    // kernel 2: Scale
    Kernel_Scale_CUDA<<<(N2+1024-1)/1024, 1024>>>(d_b2, d_c2, scalar);

    cudaEventRecord(stop);
    cudaEventSynchronize(stop);
    time_scalar=0;
    cudaEventElapsedTime(&time_scalar, start, stop);
}
```

Fig. 17: Us de cudaEvent_t

5.1 Modificacions dimensió problema

Les primeres proves realitzades han estat modificant la dimensió al problema que estem afrontant. A dimensió de problema ens referim a la quantitat d'elements que ha de tractar. Per tant, si en una funció tenim dos vectors cada vector serà de mida N/2 i si, per altra banda, una funció té un sol vector, aquell serà de mida N. Havent especificat aquest aspecte ara toca decidir quina mida de bloc volem començar. Hem decidit utilitzar una mida de bloc de 32 threads perquè és la mida d'un warp que és la quantitat més petita de threads que es poden agrupar. Per escollir la mida de bloc és molt important tenir en compte que els warp són de 32 threads perquè tots els threads dintre d'un warp han d'executar la mateixa operació. Si no és així les instruccions s'executaran en sèrie. Per tant, considerant aquest aspecte, serà convenient que quan escollim una mida de bloc sempre sigui múltiple de 32.

N	1000	10000	100000	1000000
Copy Host → Device	0.000026	0.000037	0.000122	0.000912
Kernel_Copy	0.000052	0.000031	0.000062	0.000094
Kernel_Scale	0.000014	0.000012	0.000017	0.000016
Kernel_Add	0.000013	0.000009	0.000014	0.000018
Copy Device → Host	0.000018	0.000025	0.000074	0.000583

Fig. 18: Temps d'execució amb diferents mida de problema

Com es pot observar a la figura 19, hem realitzat proves amb diferents mides i hem mesurat el temps de còmput i el temps de copiar les dades entre device/host i host/device. Les conclusions que podem extreure és que hi ha una gran quantitat de temps que va destinat a la còpia de dades. Si per exemple les diferents funcions compartissin el set de dades, seria molt menor o bé, si nosaltres féssim múltiples operacions sobre el mateix set de dades, només s'hauria de copiar una sola vegada a la GPU. També veiem que el temps d'execució no augmenta en excés a mesura que anem augmentant la N, això és perquè són unes funcions molt senzilles o només es fa una operació en cada thread i gràcies a la gran capacitat que té de paral·lelització la GPU ens permet obtenir temps molt ràpids per moltes operacions.

N	1000	10000	100000	1000000
Kernel_Copy	0.000001	0.000010	0.000095	0.000975
Kernel_Scale	0.000001	0.000007	0.000074	0.000754
Kernel_Add	0.000001	0.000006	0.000054	0.000570

Fig. 19: Temps d'execució dels kernels executats a la CPU

5.2 Modificació de la mida de bloc

En el punt anterior hem estudiat com influeix modificar la mida del problema i ara ens centrarem en quins impactes pot tenir modificar la dimensió del grid.
Les primeres proves que hem realitzat són, utilitzant els kernels anteriors, modificar la mida del bloc per veure les repercussions que pot tenir. Com es pot observar a la figura 20, en aquests casos concrets, la mida del bloc no ha condicionat molt el resultat del temps d'execució. El que si hem vist en les proves realitzades és que per N = 10.000.000 si executàvem amb una mida de bloc de 32 threads, el resultat era erroni. La causa d'aquest problema es que CUDA emmagatzema el número de bloc en una variable de 16 bits, per tant, no es poden crear més de 65.536 blocs. Tot programa que generi més d'aquesta quantitat de blocs estarà generant un overflow en aquesta variable i, per tant, tots els blocs que tinguin un id igual o superior a 65.536 quedaran ignorats perquè CUDA no tindrà forma d'identificar-los.

Block_dim (N=1000000)	32	256	1024
Copy Host → Device	0.000900	0.000880	0.000917
Kernel Copy	0.000109	0.000097	0.000100
Kernel Scale	0.000029	0.000014	0.000015
Kernel Add	0.000031	0.000017	0.000017
Copy Device → Host	0.000583	0.000554	0.000527

Fig. 20: Resultats modificació del grid

5.3 Us de les sincronitzacions

En l'apartat 3.4 hem parlat de la importància que pot tenir un bon ús de les sincronitzacions per així obtenir un millor rendiment. Com hem pogut observar en les taules anteriors hem aconseguit resultats de temps molt ràpids. Però no sempre serà així, dependrà en la complexitat del kernel i del volum de dades que haguem de processar. Llavors en els nostres exemples com es tracta d'una aplicació per fer proves ens interessa poder calcular el temps de forma precisa. Per tant, mentrestant s'executava el kernel a la GPU, la CPU estava a l'espera. És precisament en aquest moment quan estem malgastant temps de l'execució de la CPU i per optimitzar-ho es podrien dur a terme diferents millores, com per exemple, la reestructuració del codi per aprofitar els temps d'espera.

La nostra aplicació està formada per 3 blocs principalment. En primer lloc, generem tots els set de dades necessaris. En segon lloc, executem tots els kernels que necessitem. Finalment, processem els resultats i mostrem els temps d'execució. Una proposta que podria optimitzar l'ús de la CPU seria que a l'inici només es prepari el set de dades del primer kernel i que durant l'execució de cada kernel es prepari el set de dades del següent. D'aquesta manera mantindríem la CPU ocupada i estaríem aconseguint fer feina en els dos dispositius simultàniament.

5.4 Optimització dels recursos

L'últim punt que es tractarà, serà l'optimització dels recursos, a recursos de la GPU ens referim als cuda cores de la GPU. Per abordar aquest aspecte utilitzarem una aplicació de reduce, on donat un vector hem de sumar tots els valors en una sola variable. En l'apartat 4.5 podem veure els dos fragments de codi que usarem per veure dues maneres diferents de sumar tots els elements d'un vector. Hem procedit a realitzar diferents execucions amb diferents mides de problema per veure com reacciona cada versió. Els resultats obtinguts estan situats a la taula 21.

n	Versió 1	Versió 2
100.000	0,022528	0,014336
1.000.000	0,121856	0,075776
10.000.000	1,098752	0,65536
100.000.000	9,817056	5,819392

Fig. 21: Resultats de les dues versions del kernel reduce

Es pot observar que la versió 2 és clarament més ràpida respecte a la versió 1. També és cert que amb relació a la versió 1 tenim diverses optimitzacions que podrien fer-la millorar lleugerament com per exemple quan copiem les dades a la GPU al principi podem aprofitar la còpia per fer la suma i ens estalviaríem la 1a iteració, però que no té un impacte tant gran en el rendiment. El que ens permet veure aquest exemple és la importància que té un algorisme en l'execució i que és preferible en primera instància buscar diferents algorismes per elaborar la mateixa tasca i llavors escollir el més adient i posteriorment adaptar-lo i optimitzar-lo.

6 Conclusions

Després de tota la recerca i totes les proves realitzades es poden extreure diferents conclusions. En primer lloc, hem pogut veure que hi ha molts factors que poden afectar en el rendiment d'una aplicació, es poden dividir en dos grans tipus, els generals i els específics. Els paràmetres més genèrics són aquells que es poden aplicar en qualsevol aplicació com seria per exemple moure les sincronitzacions per aprofitar millor el temps, modificar la mida de bloc o bé analitzar la mida del problema per saber quan és més convenient una GPU. Tots aquests paràmetres independentment de la naturalesa de l'aplicació serem capaços d'aplicar-los.
En canvi, els paràmetres més específics de cada kernel són aquells que depèn de la naturalesa de cada problema haurem d'investigar quin és el coll d'ampolla i tractar-ho. Per exemple tenim el cas del patró reduce on podem ajustar com accedeix a memòria per obtenir un millor ús dels recursos de la GPU. Per trobar aquests factors cal realitzar un estudi exhaustiu de cada cas per trobar-los.
El segon punt a tractar són els resultats obtinguts en el punt 5.1, on es pot observar que la GPU ens dona una gran velocitat de còmput quan tractem amb molta quantitat de dades. El principal punt feble és el temps de còpia de dades entre dispositius. Per tant, ens mostra que si la nostra aplicació ha de tractar amb un gran volum de dades, pot ser òptim usar una GPU.
En tercer lloc, tenim un dels resultats més sorprenents que hem tingut, pel fet que ha estat un fet que inicialment no estava previst. En el punt 5.2 s'han realitzat diferents proves amb diferents mides de problema i hem vist que

el nombre de blocs que pots crear està representat amb una variable de 16 bits. El problema del overflow és un problema molt comú en el món de la informàtica i les GPUs no se n'escapen. A priori no ha de ser un problema, ja que les GPUs d'última generació poder tenir blocs de fins a 1024 threads, per tant, al poder crear 65.536 blocs de 1024 threads tenim una capacitat de tractar amb més de 67 milions de dades.
En quart lloc, hem estudiat les sincronitzacions, probablement és la millora més senzilla que podem dur a terme a l'aplicació, perquè només cal analitzar el codi i estructurar-lo de tal manera d'aprofitar al màxim el temps que la GPU està executant un kernel fent altres tasques. Per tant, una bona col·locació de les sincronitzacions ens permetrà tenir una millora en el temps d'execució.

Finalment, hem tractat l'optimització dels recursos de la GPU. Com hem explicat anteriorment, es tracta d'una optimització molt especifica per aquest cas concret, per tant, és difícil d'extreure'n un patró genèric perquè pugui ser adaptat a un cas general. Però, el que sí que es pot fer és quedar-nos amb la idea que en funció de com repartim la feina als threads i com creem els blocs el resultat pot ser mes o menys eficient.
Les conclusions generals que podem extreure són que el més important de fer al principi és escollir un bon algorisme relacionat amb la tasca que es va a realitzar. Un cop l'algorisme ja és escollit s'ha de fer una correcta adaptació de l'algorisme a la nostra aplicació, posteriorment elaborar una anàlisi en profunditat del codi utilitzant eines de profiling i debugging i amb els resultats d'aquestes treballar en els colls d'ampolla. Un cop amb el kernel optimitzat es poden dur a terme proves amb diferents dimensions de grid i per acabar verificar que el temps que la GPU està executant el kernel, en la mesura que sigui possible, mantenir la CPU activa amb altres tasques independents al kernel.

Com a treball futur es poden realitzar proves analitzant les dades duplicades que pugui tenir cada kernel. Per altra banda, també es pot estudiar la multiplicació entre matriu i vector i la multiplicació entre matrius. Ens aportarien un altre punt de vista de com generar el grid per accedir a matrius. Una altra proposta pot ser analitzar els comptadors hardware de la GPU, per tal d'analitzar el comportament de cada kernel.

AGRAÏMENTS

A l'equip de recerca i en especial a la meva tutora per la seva ajuda, paciència i dedicació.

REFERÈNCIES

[1] Intel, vist el dia (06/10/2021). ¿Cómo funciona la HPC?. url: https://www.intel.es/content/www/es/es/high-performance-computing/what-is-hpc.html

[2] Ikoula (19/04/2018) La importancia de la GPU en alta computación: Big Data Analytics y Ai. url: https://www.interempresas.net/Seguridad/Articulos/215959-La-importancia-de-la-GPU-en-alta-computacion-Big-Data-Analytics-y-AI.html

[3] Ortiz, J. (14/07/2020) La computación paralela: alta capacidad de procesamiento. url: https://www.teldat.com/blog/es/computacion-paralela-capacidad-procesamiento/

[4] Lázaro, A.R. vist el dia (06/10/2021). Mainframes url:https://www.teldat.com/blog/es/computacion-paralela-capacidad-procesamiento/

[5] Lázaro, A.R. vist el dia (06/10/2021). Cluster (Computadoras) url:https://www.ecured.cu/Cluster_(computadoras)

[6] Techlib, vist el dia (06/10/2021). Multi-core url: https://techlib.net/definition/multi-core.html

[7] Intel, vist el dia (06/10/2021) Procesador intel i7 11700KF. url: https://www.intel.es/content/www/es/es/products/sku/212048/intel-core-i711700kf-processor-16m-cache-up-to-5-00-ghz/specifications.html

[8] Nvidia, vist el dia (06/10/2021) GAMA GEFORCE RTX 3080. url: https://www.nvidia.com/es-es/geforce/graphics-cards/30-series/rtx-3080-3080ti/

[9] Harris, M. (01/2017) An Easy introduction to CUDA C and C++. url: https://developer.nvidia.com/blog/easy-introduction-cuda-c-and-c/

[10] Santamaría, P. (25/11/2010), OpenCL, la alternativa libre a CUDA y el futuro del la computación GPGPU. url: https://www.applesfera.com/aplicaciones-os-x-1/opencl-la-alternativa-libre-a-cuda-y-el-futuro-de-la-computacion-gpgpu

[11] rsnemmen/OpenCL-examples (14/06/2021) url:https://github.com/rsnemmen/OpenCL-examples

[12] Wkipedia (08/07/2021), OpenACC. url: https://en.wikipedia.org/wiki/OpenACC

[13] Benjamin Walton and Barton P. Miller, Exposing Hidden Performance Opportunities in High Performance GPU Aplications. Madison, WI.

[14] Harris, M. (07/2012)How to Implement Performance Metrics in CUDA C/C++ url: https://developer.nvidia.com/blog/how-implement-performance-metrics-cuda-cc/