Desenvolupament de Micro-kernels per l'anàlisi i sintonització de rendiment sobre accel·leradores (GPGPUs)

Laura Navarro Lorente

Resum– Aquest article presenta l'estudi i paral·lelització en acceleradores GPU de patrons basats en els problemes computacionals més habitualment utilitzats en el marc de la Computació d'Alt Rendiment. La paral·lelització en GPU s'ha basat en l'aplicació de diferents tècniques d'optimització mitjançant OpenACC i CUDA, seguit d'un anàlisi de l'impacte que aquestes tècniques provoquen en el rendiment obtingut per diverses mides de problema. També s'ha estudiat la importància que prenen les transferències de dades entre Device i Host a l'hora d'aconseguir acceleracions elevades. Per últim, s'ha realitzat una comparació dels resultats de rendiment obtinguts entre CUDA i OpenACC, amb el que s'ha pogut concloure que tot i que OpenACC és molt més ràpid i senzill d'implementar té un clar desavantatge envers el rendiment obtingut amb CUDA en la major part dels casos; aquest queda accentuat quan es tenen en compte els temps de transferència, que en OpenACC són molt més lents.

Paraules clau– Acceleració, Comparativa, Computació d'alt rendiment, CUDA, GPU, OpenACC, Paral·lelisme, Patró computacional

Abstract– This article presents the study of patterns based on computational problems most commonly used within High Performance Computing and their parallelization in GPU accelerators. Different optimization techniques via OpenACC and CUDA have been used for GPU parallelization, followed by an analysis of the impact these techniques caused on the performance obtained on several problem sizes. The importance of data transfers between Device and Host for achieving high speedups has also been studied. Finally, a performance comparison between CUDA and OpenACC has been made based on the obtained results, by which has been concluded that OpenACC, although, is much faster and easier to be implemented than CUDA, it has a clear disadvantage on performance compared to CUDA in most cases; this issue is accentuated when data transfer times are taken into account, which in OpenACC are much slower.

Keywords– Comparison, CUDA, GPU, High Performance Computing, Kernels, OpenACC, Parallelism, Speed-Up

1 Introducció

El High-Performance Computing (HPC) [1] té com a principal objectiu permetre l'execució d'aplicacions complexes i el processament de grans quantitats de dades de forma més ràpida mitjançant clústers de procesat, supercomputadores, i sistemes heterogenis de computació paral·lela a partir d'acceleradores. Per tant, solucionar i aconseguir un alt rendiment de problemes que són exigents computacionalment utilitzant acceleradores GPGPUs (General-Purpose GPU) [2] és una tasca complexa que ha evolucionat molt al llarg dels últims anys.

Aquest treball de final d'estudis forma part d'un projecte més gran i ambiciós en el que es vol automatitzar l'anàlisi de rendiment de possibles paral·lelitzacions realitzades en acceleradores. Aquest projecte requereix com a entrada un conjunt de dades sobre execucions en aquestes plataformes i per aquest motiu, és necessari identificar una sèrie de patrons que són representatius de l'espai de programes que actualment s'executen en GPUs. Així, la principal aportació d'aquest treball serà realitzar paral·lelitzacions amb sistemes heterogenis de CPU/GPU a través d'OpenACC [3] i CUDA (Compute Unified Device Architecture) [4] de NVIDIA [5] del codi que engloba aquest conjunt de patrons.

Per aquest propòsit, es durà a terme un anàlisi del rendiment del codi d'un conjunt de micro-kernels que ja ha tingut una primera implementació amb OpenACC [6] i proposar possibles millores. OpenACC ens ajuda a simplificar la paral·lelització de programes o aplicacions de manera senzilla gràcies a la seva aplicació a alt nivell mitjançant directives de compilació. Per altra banda, CUDA proporciona una aplicació més complexa però que ens permet tenir més control sobre les parts que es volen paral·lelitzar. A diferència d'OpenACC, el codi per a CUDA que s'executa al host pot decidir quina informació enviar a la memòria de la GPU (memòria compartida), i llança els kernels o funcions que s'executen al dispositiu. A més, CUDA permet tenir més control sobre la quantitat de threads que es volen

• **E-mail de contacte:** *laura.navarrolo@autonoma.cat*
• **Menció realitzada:** *Enginyeria de Computadors*
• **Treball tutoritzat per:** *Eduardo César-Galobardes (DACSO)*
• **Curs 2021/22**

utilitzar per executar cada part del codi, aconseguint així un comportament més dinàmic.

Aquest Treball de Final de Grau (TFG) està relacionat amb un projecte de recerca que es porta a terme pel Departament d'Arquitectura de Computadors i Sistemes Operatius (DACSO) [7] de la UAB anomenat "Computació Avançada per als Reptes de la sOcietat digitaL" (CAROL) [8][9]. Una part d'aquest projecte es desenvolupa al voltant de la implementació i l'entrenament de models generats amb tècniques de Maching Learning [10] per realitzar sintonització de rendiment d'aplicacions paral·leles [11]. D'aquesta manera, el model ajudarà a la presa de decisions de la millor estratègia a seguir a l'hora d'optimitzar un algorisme complet o de forma parcial.

Per tal d'aconseguir aquest objectiu, és necessari tenir un dataset [12] el més complert possible, amb diferents tipus de problemes de rendiment, per tal de poder entrenar el model en el ventall més ample possible de regions de codi paral·leles (kernels) més freqüentment utilitzats.

A través d'aquest TFG es pretén fer la comparativa dels rendiments obtinguts en les implementacions sèrie, OpenACC i CUDA per tal d'ajudar al model automàtic en la presa d'aquest tipus de decisions de rendiment.

La resta d'aquest document està organitzat de la següent manera. La Secció 2 presenta els objectius proposats per aquest projecte. A continuació, a la Secció 3 s'explica l'estat de l'art del camp de la paral·lelització CUDA envers OpenACC. Seguidament, a la Secció 4 es descriu la metodologia, és a dir, els passos seguits per a portar a terme els objectius. La Secció 5 mostra el desenvolupament i l'experimentació realitzada. Tot seguit, la Secció 6 exposa els resultats obtinguts en les experimentacions on juntament amb la Secció 7 es raonen les conclusions extretes dels resultats i línies futures.

2 OBJECTIUS

A continuació s'exposen els objectius d'aquest projecte:

1. Anàlisi del conjunt de kernels actual i recerca de possibles nous kernels amb diferents problemes de rendiment.
2. Anàlisi del rendiment obtingut en la codificació en sèrie dels kernels.
3. Revisió de la paral·lelització proporcionada en OpenACC i recerca de noves maneres d'implementar-ho per tal d'optimitzar el codi, acompanyat de la respectiva experimentació.
4. Implementació de la paral·lelització en CUDA de les regions de codi paral·leles (kernels), acompanyada de la respectiva experimentació.
5. Exposició dels resultats obtinguts a través d'un anàlisi i una comparativa de les implementacions realitzades. Extreure'n conclusions respecte sota quines condicions és millor utilitzar OpenACC o CUDA.

3 ESTAT DE L'ART

Durant l'última dècada, l'acceleració d'aplicacions ha anat evolucionant i s'ha cercat noves maneres de programació per aconseguir-ho. S'ha investigat l'ús de la programació per GPU per tal d'obtenir un rendiment a gran escala en aplicacions complexes de diverses tipologies com intel·ligència artificial o computació d'alt rendiment.

D'aquesta manera, CUDA compta amb un conjunt ampli de llibreries accelerades que permeten accelerar de manera eficient aplicacions d'alt còmput matemàtic.

Així doncs, una alternativa a CUDA és l'OpenACC que és una solució senzilla per aprofitar la computació en GPU, ja que es requereixen molt pocs canvis de codi per a habilitar el suport per a les acceleradores. Tot i que fins ara era necessari utilitzar el compilador *pgi*, NVIDIA l'ha reanomenat com a NVIDIA HPC Compiler (*nvc*) [13], eliminant la llicència necessària en *pgi* i mantenint la compatibilitat d'execució de codi juntament amb directives OpenMP, MPI i CUDA.

Hi ha estudis que conclouen que la diferència de rendiment entre les dues opcions és degut al fet que, amb OpenACC no sempre és possible extreure'n tot el rendiment possible a l'aplicació donat que té limitacions a la seva especificació que dificulten l'ús complet dels recursos hardware [14][15]. Per altra banda, existeixen estudis que adjudiquen aquesta diferència en el rendiment al compilador (fins ara *pgi*) a conseqüència d'haver de traduir els kernels d'OpenACC a codi objecte, això no és necessari amb CUDA.[16].

No obstant, en altres estudis es dictamina que la transferència de dades en OpenACC tendeix a ser més ràpida que en CUDA, però alhora requereix fer més copies de dades (*memcpy*) fent que el temps de transferència acabi sent més lent en OpenACC que en CUDA [15].

4 METODOLOGIA

La metodologia escollida per aconseguir els objectius d'aquest projecte ha estat la metodologia Agile [17]. Aquesta és del tipus Rapid Application Development (RAD) que tracta d'aplicar models de millora de forma continuada a través de la planificació, del desenvolupament, la comprovació i la millora d'aquestes tasques modificant aquells factors que no han estat realitzats correctament. Gràcies a aquesta metodologia es poden prendre millors decisions en quant a la planificació i organització de les tasques i solucionar els problemes que es presenten durant el desenvolupament del projecte.

Per tal d'aplicar aquesta metodologia al projecte, s'han realitzat reunions setmanals de seguiment on s'han presentat els avenços obtinguts en quant a la paral·lelització del conjunt de kernels, així com els resultats aconseguits. A més, en aquestes reunions s'ha decidit si la versió presentada era prou satisfactòria, en cas contrari s'han discutit possibles estratègies de millora.

Degut als atacs informàtics soferts durant el mes d'octubre 2021 a la UAB, el sistema de virtualització, on estan allotjats la major part dels serveis que proporciona la universitat, es va veure afectat. Per aquest motiu, no es van poder utilitzar els serveis i aplicatius de la universitat durant un llarg període de temps. Això va suposar un gran inconvenient per la realització d'aquest Treball de Final de Grau donat que per al seu desenvolupament es requereix fer ús dels recursos proporcionats per DACSO, com són els clústers amb GPGPU instal·lada.

D'aquesta manera, es va haver de modificar la planificació proposada inicialment donat que es va veure greument afectada, implicant també la modificació de l'ordre d'execució de les tasques i objectius.

Aquest projecte va començar examinant el dataset actual. Seguidament, es va realitzar un profiling de la implementació en sèrie per tal de poder localitzar aquelles parts del codi que provocaven un cost computacional més alt i

dedicar-hi més esforços a l'hora de paral·lelitzar els micro-kernels. Així, es va aconseguir extreure el major rendiment possible.

A conseqüència de les incidències sofertes ja esmentades i a la falta de previsió per restablir els serveis proporcionats per la universitat, es va decidir aprofitar el temps d'espera fins a la restauració dels servidors del Departament incloent una tasca a la planificació. Aquesta va ser desenvolupar un motor d'execució que permetés gestionar de manera més senzilla i eficient l'execució del conjunt de kernels amb les diverses aproximacions de paral·lelització (sèrie, OpenACC, CUDA).

Per a realitzar aquesta tasca va ser necessari el muntatge d'un setup a l'ordinador portàtil personal per a la paral·lelització amb CUDA i OpenACC, això va permetre seguir desenvolupant el projecte amb els mínims inconvenients possibles.

Així doncs, es va procedir amb la instal·lació d'un dual-boot amb Windows i Linux com a sistemes operatius, incloent tant la recerca i instal·lació de tots els drivers necessaris per a la utilització de CUDA i OpenACC en el ordinador portàtil com dels toolkits requerits per a la realització del profiling. A continuació, s'ha realitzat les implementacions en CUDA i OpenACC.

Un cop es van solucionar els problemes tècnics de la UAB, es va realitzar un anàlisi del resultats del temps d'execució obtinguts en ambdós casos (OpenACC i CUDA) pels diferents kernels i per diferents mida de problema. D'aquesta manera es va aconseguir fer una comparativa exhaustiva respecte aquests resultats i es va poder extreure conclusions sobre quin és el mètode de paral·lelització més apropiat en cada un dels casos.

5 DESENVOLUPAMENT

5.1 Anàlisi dataset

Per tal de posar aquest projecte en context, s'ha analitzat el dataset proporcionat. Això ha aportat un coneixement més profund del conjunt de kernels que s'ha hagut de paral·lelitzar. D'aquesta manera, els patrons que componen el dataset actual són:

- **Kernel_Copy**: realitza una còpia d'un vector a un altre.

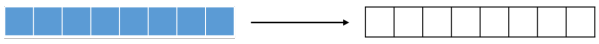

Fig. 1: Descripció de l'algorisme *Copy*

- **Kernel_Scale** : realitza una multiplicació d'un vector per un valor escalar.

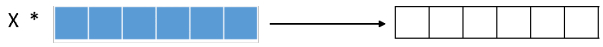

Fig. 2: Descripció de l'algorisme *Scale*

- **Kernel_Add**: duu a terme una suma de dos vectors.

Fig. 3: Descripció de l'algorisme *Add*

- **Kernel_Triad**: realitza una multiplicació d'un vector per un valor escalar i la suma del resultat amb un segon vector.

Fig. 4: Descripció de l'algorisme *Triad*

- **Kernel_Reduction**: realitza la suma de tots els valors d'un vector en un valor escalar.

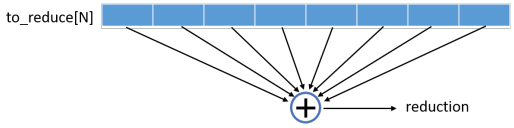

Fig. 5: Descripció de l'algorisme *Reduction*

- **Kernel_2PStencil**: fa la mitjana entre els dos veïns d'un element d'un vector. Per tant, com es pot observar a la següent imatge, si l'índex del vector senyala la posició X, llavors fa la mitjana dels dos elements remarcats en verd i emmagatzema el resultat a la posició X.

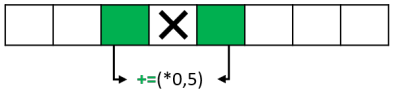

Fig. 6: Descripció de l'algorisme *2PStencil*

- **Kernel_2D4PStencil**: funciona de similar manera que el Kernel_2PStencil amb la diferència de que en aquest cas es realitzarà la mitjana dels quatre veïns que tenen connectivitat a 4 d'una posició d'una matriu.

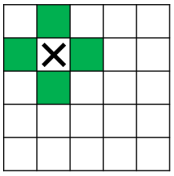

Fig. 7: Descripció de l'algorisme *2D4PStencil*

- **Kernel_Stencil**: calcula la mitjana dels sis elements veïns d'una posició d'una matriu en 3D, i a diferència dels dos anteriors també inclou el valor d'aquesta posició en el càlcul de la mitjana.

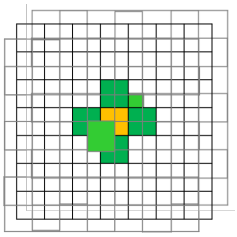

Fig. 8: Descripció de l'algorisme *Stencil*

- **Kernel_MatXVec**: realitza una multiplicació d'una matriu per un vector.

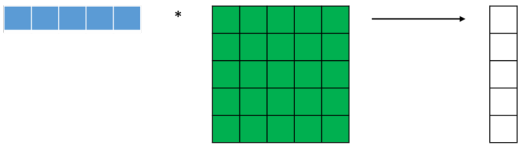

Fig. 9: Descripció de l'algorisme *MatXVec*

- **Kernel_MatMult**: realitza una multiplicació de dues matrius amb accessos a memòria optimitzats per l'execució en CPU.

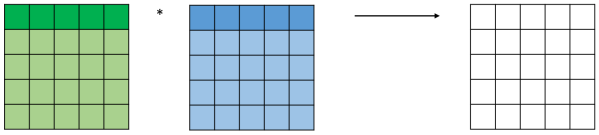

Fig. 10: Descripció de l'algorisme *MatMult*

- **Kernel_MatMultNoOpt**: tracta d'una multiplicació de dues matrius però, en aquest cas, sense optimitzacions

d'accessos a memòria.

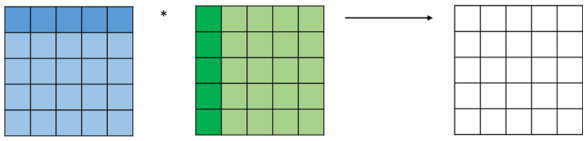

Fig. 11: Descripció de l'algorisme *MatMultNoOpt*

- **Kernel_Stride (2, 4, 16, 64)**: realitzen la còpia del vector mitjançant salts de mida stride (2, 4, 16, 64).

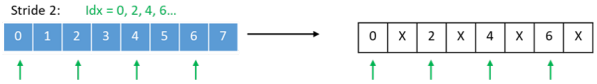

Fig. 12: Descripció de l'algorisme *Stride2*

- **Kernel_Rows**: realitza una còpia d'una matriu per columnes.

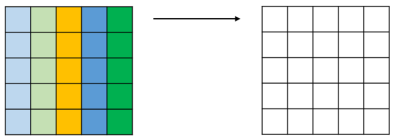

Fig. 13: Descripció de l'algorisme *Rows*

5.2 Motor d'execució

Per altra banda, s'ha implementat un motor d'execució per tal de facilitar l'execució i la diferenciació de les parts del codi a analitzar. S'ha distingit entre aquelles parts necessàries per a l'execució del codi i les funcions relacionades amb els kernels i la seva paral·lelització. D'aquesta manera s'ha dividit en dues parts:

- **StreamHost:** La part executora del codi, que és el fragment de codi que és comú per a totes les implementacions de paral·lelització. Aquí es defineixen i s'inicialitzen les variables i estructures, i es fa l'assignació i l'alliberació de memòria en CPU. A més, conté les funcions pertinents per al perfilat de rendiment, etc.
- **StreamKernels_XX:** S'ha generat un arxiu exclusiu d'execució per a cada tipus de paral·lelització que es portarà a terme en aquest projecte (sèrie, OpenACC, CUDA). Aquests arxius contenen el conjunt de kernels, assignació i alliberament de memòria en GPU i la còpia de dades entre Host-Device en el cas de que sigui necessari. En aquests arxius, s'hi inclouran les modificacions específiques per a cada tipus de paral·lelització.

A part dels arxius amb el codi font s'han generat un conjunt de scripts per tal de poder compilar i enllaçar de forma adient cadascun dels StreamKernels_XX. Aquests scripts es mostren a continuació.

```
commonFlags="-O2 -DN=$size -DNTIMES=10"
```

Sèrie

```
# Compiler Stage and Linker
gcc streamHost.c streamKernels.c $commonFlags -o
    streamExe -lm
```

CUDA

```
# Compiler Stage
gcc -c -g streamHost.c $commonFlags -o streamHost
    .o;
nvcc -c -arch=sm_60 $commonFlags
    streamKernels_cuda.cu -o streamKernels.o
# Linker Stage
gcc streamHost.o streamKernels.o -o streamExe -L/
    usr/local/cuda/lib64 -lcudart -lm
```

OpenACC

```
# Compiler Stage and Linker
nvc -acc=gpu -ta=tesla streamKernels_oacc.c
    streamHost.c $commonFlags -Minfo=all -o
    streamExe -lm
```

Inicialment és necessari fer ús de flags de compilació comuns per a totes les implementacions, on *$size* correspon a la mida del problema. Per compilar la codificació sèrie en llenguatge C, és necessari fer servir el compilador *gcc*. El mateix passa amb CUDA, però amb la diferència de que en aquest cas cal compilar la part del la part StreamKernels amb *nvcc* i enllaçar-ho amb *gcc*. Per altra banda, per compilar la implementació OpenACC és necessari fer servir el compilador *nvc*.

5.3 Muntatge setup

Per tal de ser capaç de compilar i executar el codi implementat s'ha muntat un setup a l'ordinador portàtil personal. Les especificacions d'aquest ordinador són, un processador Intel® Core™ i7-9750H, amb 8 GB DDR4 de RAM i una targeta gràfica NVIDIA GeForce® GTX 1650 de 8GB [18]. D'aquesta manera s'ha instal·lat un sistema dual-boot amb els sistemes operatius Windows i Linux. A continuació, s'ha cercat i instal·lat tots els drivers necessaris per poder utilitzar CUDA a la targeta gràfica de NVIDIA [19]. També ha estat necessari fer una recerca d'eines per a la posterior realització del perfilat dels resultats [20].

Finalment, els resultats de la Secció 6 s'han obtingut d'utilitzar el clúster del Departament, que consta de les següents especificacions:

- Host: aolin-login.uab.es (AOLIN23)
- Processador: Intel® Core™ i5-2400 CPU (3.10GHz).
- Sistema operatiu: CentOS Linux 7 (Core).
- Memòria: 8 GB memory.
- Targeta gràfica: NVIDIA GeForce RTX 3080.

5.4 Implementació amb CUDA

Com ja s'ha comentat anteriorment, la primera aproximació de paral·lelització que s'ha portat a terme és amb CUDA.

5.4.1 Implementació comú

Per a això, ha estat necessari definir els punters de les estructures de dades que es faran servir dins el dispositiu (GPU). Així doncs, s'ha hagut d'afegir al codi la següent definició per poder fer servir les estructures des de la GPU. S'ha decidit utilitzar la nomenclatura $d\_$ per indicar aquells punters utilitzats en el Device.

Per altra banda, també ha estat necessari realitzar l'assignació i l'alliberació de memòria d'aquelles estructures que es faran servir al Device [21]. Per fer-ho s'ha utilitzat respectivament `cudaMalloc()` i `cudaFree()`.

En aquest cas, també ha estat necessari afegir extern "C" per tal de poder cridar aquesta funció des de la StreamHost del codi en CPU. Ja que CUDA es compila com si fos un arxiu C++ i, en canvi, la part executora es compila com si fos un arxiu de llenguatge C. Així mateix, per aconseguir fer les còpies de la informació des de fora del Device, s'ha definit una funció de copia per a cada estructura, segons si la informació de l'estructura és necessària copiar-la de Host-Device o de Device-Host. D'aquesta manera s'aconsegueix alliberar el temps d'execució dels kernels del temps requerit per a les còpies.

5.4.2 Paral·lelització dels kernels

Un cop estructurada la base, s'ha pogut començar amb la implementació de la paral·lelització CUDA al conjunt de patrons. Per al seu desenvolupament s'han de tenir en compte diversos factors o regles que influeixen en la obtenció de bons resultats al utilitzar GPUs. Aquests són:

- Reutilització de dades dins de la GPGPU a través de la memòria compartida (SMP) [22] i de registres.
- Optimització dels accessos a memòria (coalesced access).
- Maximitzar la relació entre el nombre d'operacions i els accessos a memòria.
- S'ha de mantenir la regularitat de l'execució i procurar reduir els controls de flux, instruccions o crides a funció per tal de minimitzar la divergència.
- Maximitzar l'ocupació del multi-processador. És a dir, maximitzar la relació entre el nombre de warps actius i el nombre màxim de warps admesos en un multi-processador de la GPU.

Seguint aquest conjunt de regles es podrà aconseguir evitar limitacions d'amplada de banda de memòria i assegurar que la càrrega computacional del problema és suficient per compensar el cost d'ús de la GPU (overhead) amb els resultats obtinguts. Això s'ha de tenir en compte pel fet que la sobrecàrrega de comunicació entre el host (CPU) i el device (GPU) és generalment un dels colls d'ampolla en el rendiment de sistemes HPC que utilitzen acceleradors. És per aquest motiu que per a problemes senzills no és una bona estratègia executar-ho des de la GPU.

Kernel_Copy, Scale, Add, Triad

Inicialment, s'ha paral·lelitzat el Kernel_Copy, per fer-ho ha estat necessari fer servir un GridDim d'una dimensió i $ceil(N2/N\_Threads)$ blocs i un BlockDim de 1024 threads. D'aquesta manera, el codi s'estructura de forma que hi hagi el mateix nombre d'elements del vector que threads. Així, s'aconsegueix que cada thread faci només una operació de còpia.

D'igual manera, s'ha paral·lelitzat el patró Kernel_Scale, tot i que ara cada thread multiplica el valor del vector per un valor escalar donat per paràmetre d'entrada.

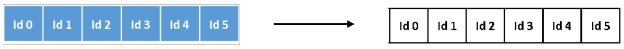

Fig. 14: Implementació en CUDA del Kernel_Copy.

El Kernel_Add realitza la suma de dos vectors, per aquest motiu s'ha afegit un tercer vector a l'execució del kernel. En aquest cas, cada thread fa dos lectures a memòria global, un per cada vector d'entrada. Per altra banda, també ha estat necessari canviar la mida del GridDim per $ceil(N3/N\_Threads)$.

Per acabar, en el Kernel_Triad cada thread calcula primerament la multiplicació d'un vector per un valor escalar i, a continuació, es suma el resultat anterior amb un segon vector.

Kernel_Reduction

Aquest patró consisteix en fer la suma de tots els valors d'un vector i emmagatzemar el resultat en un valor escalar. Per fer-ho ha estat necessari utilitzar un GridDim de $N/N\_Threads$ i mantenir el valor del BlockDim en $N\_Threads$ threads. D'aquesta manera, cudaKernel_Reduction rebrà per paràmetre el vector i la variable escalar on desar el resultat final.

Per implementar el codi per executar-ho amb CUDA ha estat necessari definir un vector auxiliar on s'anirà acumulant la suma dels valors del vector, aquest s'anomena *to_reduce*. Aquest vector és de tipus *__shared__* i, així, tots els threads del mateix bloc tenen accés a la mateixa memòria compartida [23]. També és necessari inicialitzar-lo amb els valors del vector passat per paràmetre, *d_a1*.

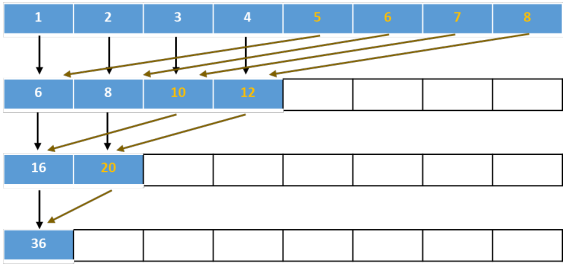

Fig. 15: Implementació en CUDA del Kernel_Reduction.

Per altra banda, per portar a terme aquest algorisme amb CUDA, es divideix el vector per la meitat $stride = N/2$ i es suma el primer element de la primera meitat amb el primer de la segona, el segon element de cada partició entre ells, etc. i s'emmagatzema el resultat a la posició agafada de la primera meitat del vector, reduint així el nombre de posicions a sumar. Aquest procediment es repeteix fins que només quedi un sol valor.

Per tal d'evitar carrera de dades cal sincronitzar els threads d'un mateix bloc entre iteracions mitjançant barreres *__syncthreads()* [23]. Així, s'aconsegueix que els threads s'esperin a la barrera fins que tots hagin executat la iteració abans de poder continuar amb l'execució.

Finalment, donat que quan *N* és més gran al nombre de threads per bloc, s'ha d'utilitzar més d'un bloc i com que la memòria compartida no es comparteix entre blocs, cal que cada bloc calculi el seu *reduction* i, al acabar, cada thread amb ID 0 utilitza un *atomicAdd* per realitzar la reducció final entre blocs.

Kernel_2PStencil, 2D4PStencil, Stencill

Pels diferents *Stencils* s'ha de tenir en compte que els marges del vector o matriu no generen cap sortida pel fet que no tenen veïns suficients per calcular la mitjana. Per tant, pel *Kernel_2PStencil* s'han iniciat els càlculs a partir de l'element 1 del vector i s'ha utilitzat un GridDim de $N2/N\_Threads$. Això succeeix d'igual manera per a l'últim element del vector, ja que l'índex només arribarà fins a $N2 - 1$. Per tal d'extreure un millor rendiment fent ús de la memòria compartida, cada thread carregarà a la memòria compartida del seu bloc un element, exceptuant el primer i últim thread, que n'hauran de carregar dos. Gràcies a això, s'aconsegueix evitar accessos innecessaris a memòria global, donat que els veïns ja hauran carregat la informació necessària per cada thread.

En el cas de *Kernel_2D4PStencil*, s'ha definit una finestra quadrada de $N\_Threads * N\_Threads$ on cada thread guarda a memòria compartida el seu element i, en el cas que sigui un thread del perímetre de la finestra, ha de llegir els veïns exteriors, com es pot observar a la Figura 16. Per últim, al tractar-se d'una matriu és necessari utilitzar un GridDim de dues dimensions mitjançant el tipus de dades $dim3(n/N\_Threads, n/N\_Threads)$.

Per últim, en el Kernel_Stencil, la finestra té forma de cub de $N\_Threads*N\_Threads*N\_Threads$ cada thread carregarà a la memòria compartida el seu element i dels veïns

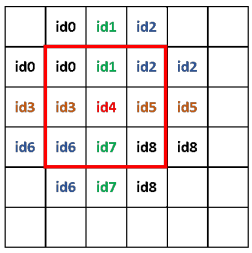

Fig. 16: 2DP4Stencil: Elements a carregar per cada thread

si és necessari. En aquest cas, com que l'element d'entrada és una matriu en 3D de $n * n * n$, s'ha de fer servir un *grid* en tres dimensions on cada eix té $n/N\_Threads$.

Kernel_MatXVec

Aquest kernel realitza la multiplicació d'una matriu per un vector. Inicialment es crea un vector on es guardarà el valor de la reducció realitzada per obtenir el resultat final. Després es realitzen els càlculs intermedis a la memòria compartida. Així, es divideix la matriu en submatrius de $N\_Threads * N\_Threads$ elements i el vector en subvectors de $N\_Threads$ elements. Per aquest motiu, es llegeix de memòria global aquestes regions per tal de reaprofitar el màxim possible i reduir accessos a memòria. Un cop cada bloc ha realitzat els seus càlculs parcials de matriu per vector, es realitza un *atomicAdd* per fer la reducció entre blocs.

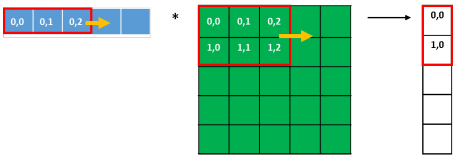

Fig. 17: Moviment de la finestra per MatXVect

Kernel_MatMult

El Kernel_MatMult realitza la multiplicació de dues matrius amb accessos a memòria optimitzats. Per tal de reduir el nombre d'accessos a memòria, s'ha decidit utilitzar l'estratègia de *tiling*, que consisteix en subdividir les matrius en finestres més petites, en aquest cas de $N\_Threads * N\_Threads$. Per això, és necessari llegir aquesta part de matriu més petita de les dues matrius d'entrada i es realitzen les multiplicacions de matrius parcials. Un cop realitzades, es mou la finestra $N\_Threads$ elements de les matrius d'entrada per continuar fent els càlculs i anar afegint el resultat al valor obtingut anteriorment.

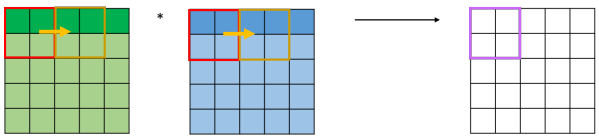

Fig. 18: Moviment de la finestra del *tiling*

S'ha de tenir en compte que si les dimensions de les matrius no són múltiple de $N\_Threads$, la última iteració del moviment de la finestra no es calcularà de manera completa donat que, les últimes posicions aniran a parar fora de la matriu original. Aquest problema, s'ha solucionat aplicant la tècnica de *loop peeling*, que consisteix en eliminar aquelles condicions dins dels bucles que només afecten al principi o al final de la iteració. En aquest cas, s'ha utilitzat per extreure la condició que controla el marge final de la matriu, ja que només prenia efecte en la última iteració del bucle.

Kernel_MatMultNoOpt

El Kernel_MatMultNoOpt, de la mateixa manera que el Kernel_MatMult, realitza la multiplicació de dues matrius però aquesta vegada sense accessos optimitzats a memòria. En aquest cas també s'ha utilitzat l'estratègia de *tiling*, amb finestres de $N\_Threads * N\_Threads$. Així doncs, és necessari llegir les posicions englobades per les finestres de les dues matrius d'entrada i realitzar els càlculs necessaris. Un cop realitzats, s'ha de moure la finestra $N\_Threads$ elements per continuar fent els càlculs i anar afegint el resultat al valor obtingut anteriorment. Per últim i d'igual forma que amb el Kernel_MatMult, s'ha de tenir en compte que si les dimensions de les matrius no són múltiple de $N\_Threads$, la última iteració del moviment de la finestra no es calcularà de manera completa, aquest problema també s'ha solucionat aplicant la tècnica de *loop peeling*.

Kernel_Stride (2, 4, 16, 64)

Aquests kernels fan una còpia d'un vector realitzant salts de mida *stride* (2, 4, 16, 64). Això s'ha implementat de manera que hi hagi el mateix nombre de threads que d'elements als vectors, i que cada thread utilitzi el seu index `j=blockIdx.x*blockDim.x+threadIdx.x` a l'hora de calcular l'índex de la posició que s'ha de fer la copia. Per fer-ho, es multiplica l'índex j per l'$stride$ i s'aplica un mòdul de la mida del vector per recorre'l de forma circular ($(j \times stride)\%n$) i, d'aquesta manera, cada thread fa la còpia d'un sol element.

Kernel_Rows

En el cas del Kernel_Rows es du a terme la còpia d'una matriu per columnes accedint cada thread a una posició i realitzant la còpia. Per la naturalesa d'aquest kernel, s´ha decidit que els blocs tinguin forma de columna en comptes de fila, per tant $dim3(1, N\_Threads)$ i al tractar-se d'una copia d'una matriu, el *grid* té la mida $dim3(n, n/N\_Threads)$.

5.5 Implementació amb OpenACC

A continuació, s'ha realitzat la revisió de la paral·lelització proporcionada en OpenACC cercant diferents formes d'optimitzar cadascun dels patrons.

5.5.1 Implementació comú

Inicialment, per tal de fer les còpies de la informació des de fora del Device i així excloure'n el temps de còpia del temps d'execució dels kernels, s'ha definit una funció de còpia per a cada estructura de la mateixa manera que a la implementació amb CUDA. Conjuntament s'ha obert una regió data per a la transferència de dades entre Host i Device amb les que s'ha realitzat les operacions pertinents dins de la GPU. D'aquesta manera, les directives *acc enter data* i *acc exit data* defineixen on comença i acaba una regió de dades.

5.5.2 Paral·lelització dels kernels

Un cop estructurada la base, s'ha pogut començar amb la implementació de la paral·lelització OpenACC al conjunt de patrons, fent les millores oportunes per cada kernel.

Kernel_Copy, Scale, Add, Triad

Per aquests kernels, s'ha modificat completament la directiva donat que prèviament es feia la paral·lelització del kernel juntament amb les còpies d'entrada i sortida dels vectors corresponents. Ha estat necessari paral·lelitzar el bucle amb *parallel loop* i s'ha afegit la definició de la mida del

vector a utilitzar *vector_length*, i la clàusula *present* informa la GPU que ja compta amb les dades. D'aquesta manera, la directiva proposada és la següent:

```
#pragma acc parallel loop present(c2,b2)
    vector_length(N_THREADS)
```

Kernel_Reduction

En el cas del Reduction, donat que fins ara comptava amb la paral·lelització del bucle i la clàusula per accelerar el *reduction*, s'ha experimentat amb la mida del vector, ja que això ajuda a millorar el rendiment.

```
#pragma acc parallel loop present(a1) reduction
   (+:reduc) vector_length(N_THREADS)
```

Kernel_2PStencil

En aquest kernel s'ha afegit la clàusula per escollir la mida del vector que s'ha utilitzat *vector_length*.

```
#pragma acc parallel loop  present(b2, c2)
    vector_length(N_THREADS)
```

Kernel_2D4PStencil

Donat que aquest patró prèviament ja tenia la directiva on es paral·lelitzava el bucle i utilitzava la clàusula *tile* [24], s'ha experimentat amb la mida d'aquesta directiva. L'esmentada clàusula li dóna al programador una forma d'expressar la localitat dins de bucles niats, així, guia al compilador a opcions d'optimització addicionals . D'aquesta manera, s'ha intentat millorar el rendiment explotant tant la localitat de dades com la reutilització de dades dins dels bucles.

```
#pragma acc parallel loop present(b2,c2)
   tile(N_THREADS,N_THREADS)
```

Kernel_Stencil

El procediment inicial per al Kernel_Stencil era aplicar directives de paral·lelització de bucle per a cadascun dels tres nivells de bucle niats. La solució proposada és la paral·lelització del bucle més extern utilitzant la clàusula *tile* en tres dimensions per tal de reutilitzar el major nombre de dades possible.

```
#pragma acc parallel loop present(b2, c2) tile(
    N_THREADS, N_THREADS, N_THREADS)
```

Kernel_MatXVec

Per al Kernel_MatXVec ha estat necessari paral·lelitzar a dos nivells de bucle. Per una banda, al nivell extern ha estat necessari la paral·lelització dels bucles, juntament amb la selecció de la mida del vector a utilitzar.

```
#pragma acc parallel loop present(mat_atax, vxmi,
     vxmo) vector_length(N_THREADS)
```

Per altra banda, al nivell intern ha requerit utilitzar la paral·lelització independent del bucle conjuntament amb la clàusula *reduction* per a la reducció dels valors de la variable auxiliar.

```
#pragma acc loop reduction(+:aux)
```

Kernel_MatMult

Inicialment, en aquest patró s'utilitzaven directives de paral·lelització de bucle per a cadascun dels tres nivells de bucle niats. La modificació que es presenta per al Kernel_MatMult és paral·lelitzar el nivell més extern utilitzant la clàusula *tile*, i el bucle més intern aplicant un *reduction* per realitzar la reducció.

```
#pragma acc parallel loop present(d3,e3,f3)
    tile(N_THREADS, N_THREADS)
```

```
#pragma acc loop reduction(+:aux)
```

Kernel_MatMultNoOpt

El Kernel_MatMultNoOpt ha requerit replantejar les directives inicials degut a que, de la mateixa manera que al Kernel_Rows, es proposava l'ús de la clàusula *collapse* al bucle extern, i després paral·lelitzar el bucle més intern junt a la clàusula *reduction*. Això no és necessari si, al bucle més extern se li aplica la clàusula *tile*.

```
#pragma acc parallel loop present(d3, e3, f3)
    tile(N_THREADS, N_THREADS)
```

Kernel_Stride (2, 4, 16, 64)

Per als Stride s'ha mantingut la paral·lelització del bucle i s'ha afegit la selecció de la longitud del vector.

```
#pragma acc parallel loop present(b2,c2)
    vector_length(N_THREADS)
```

Kernel_Rows

En aquest cas ha estat necessari canviar el plantejament inicial per tal de millorar el rendiment, donat que s'aplicava la clàusula *collapse* que aplana els bucles niats en un únic. Per tant, aquesta clàusula és útil quan existeixen molts bucles niats o aquests són molt curts. En canvi, la solució proposada com a alternativa és utilitzar la clàusula *tile* amb la finalitat de dividir els bucles en finestres i aprofitar l'alta localitat que presenta el kernel.

```
#pragma acc parallel loop present(b2,c2)
    tile(N_THREADS, N_THREADS)
```

6 EXPERIMENTACIÓ

Aquesta secció mostra els resultats obtinguts en les diferents experimentacions que s'han realitzat amb CUDA i OpenACC. Com ja s'ha comentat anteriorment, els resultats s'han extret de l'execució de les implementacions al clúster del Departament, concretament a AOLIN23.

Aquestes execucions s'han realitzat amb diferents mides de problemes, però per simplificar l'exposició dels resultats s'ha reduït el nombre de mides de problema, utilitzant únicament quatre mides: una menuda, dues intermèdies i una de gran.

6.1 Resultats

Kernel_Copy, Scale, Add, Triad

Si s'observen els resultats de la Taula 1 es pot veure com en el cas dels kernels Copy, Scale i Add tenen un comportament similar entre sí. Això és degut a que es tracta d'operacions lineals o vectorials, on cada thread calcula un element del vector.

Per altra banda, a la Figura 19, el kernel Triad tot i tractar-se del mateix tipus d'operació, OpenACC no és capaç d'obtenir els mateixos temps d'execució que a CUDA.

TAULA 1: Speed up dels Kernels Copy, Scale i Add

Mida	Copy		Scale		Add	
N *float*	CUDA	OpenACC	CUDA	OpenACC	CUDA	OpenACC
7936	0,44	0,41	0,31	0,54	0,29	0,27
130560	4,71	3,72	9,44	7,05	6,69	4,59
1310720	39,39	34,10	58,08	50,64	38,14	35,99
9437184	70,37	73,10	84,45	86,43	73,23	70,35

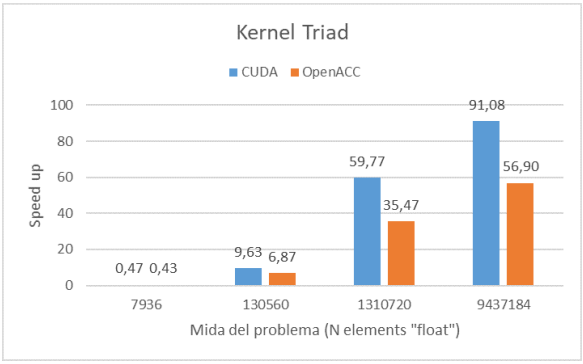

Fig. 19: Resultats obtinguts al Kernel_Triad.

Kernel_Reduction

A la Figura 20, es poden observar els resultats obtinguts per al kernel de reducció. En aquest cas, els speedup no són tant elevats donat que el compilador és prou intel·ligent com per treure'n un bon rendiment en CPU. També s'ha de tenir en compte que en GPU, l'algoritme *reduction*, per definició no pot igualar la càrrega entre threads i, per tant, no pot fer un aprofitament de la paral·lelització massiva.

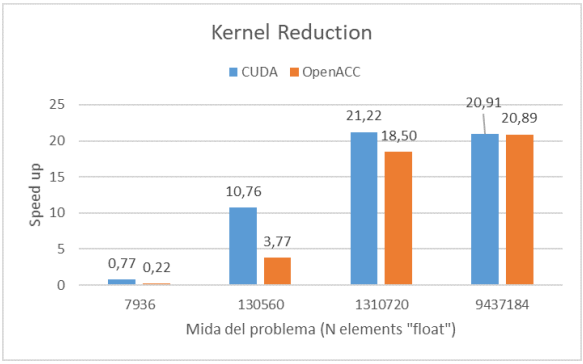

Fig. 20: Resultats obtinguts al Kernel_Reduction.

Kernel_2PStencil, 2D4PStencil, Stencil

En el cas de 2PStencil i 2D4PStencil el comportament obtingut és molt similar tot i haver obtingut resultats lleugerament millors per al kernel 2D4PStencil. En canvi, el Stencil, al realitzar l'accés a les dades diferent dels dos anteriors, únicament ha aconseguit la meitat del speedup d'aquests. Això es pot observar a la taula següent:

TAULA 2: Speed up de 2PStencil, 2D4PStencil i Stencil

Mida	**2PStencil**		**2D4PStencil**		**Stencil**	
N *float*	CUDA	OpenACC	CUDA	OpenACC	CUDA	OpenACC
7936	0,43	0,42	0,51	0,39	1,02	0,74
130560	9,35	7,35	13,49	11,46	16,95	11,24
1310720	56,92	52,43	62,95	70,17	63,14	40,02
9437184	93,85	93,67	108,91	115,36	78,11	55,49

Kernel_MatXVec, MatMult, MatMultNoOpt

En aquest cas es pot comprovar com en implementacions d'alta complexitat matemàtica, com pot ser el tractament de matrius, els speedups creixen de manera important. Per MatXVec, al tractar-se d'una matriu per un vector, la reutilització de dades es realitza només per la matriu i, per tant, els resultats no són tant elevats com en els altres dos casos.

Tant mateix, es pot observar a la Taula 3 com per als kernels MatMult i el MatMultNoOpt s'accentua la millora dels speedups al aconseguir reutilitzar un gran nombre de dades. La millora del kernel MatMultNoOpt respecte el MatMult és molt destacable i és a conseqüència de la forma d'accedir a la memòria, ja que el MatMult té optimitzats els accessos per a CPU amb memòria *cache*, però a nivell de GPU no hi ha diferència entre aquestes dues implementacions.

TAULA 3: Speed up: MatxVect, MatMult, MatMultNoOpt

Mida	**MatxVect**		**MatMult**		**MatMultNoOpt**	
N float	**CUDA**	**OpenACC**	**CUDA**	**OpenACC**	**CUDA**	**OpenACC**
7936	0,58	0,75	11,96	14,14	12,88	9,79
130560	14,03	9,64	243,08	273,12	358,58	294,00
1310720	37,77	38,52	304,50	461,84	1511,96	1187,21
9437184	46,21	65,16	330,63	472,06	2367,07	1827,90

Kernel_Stride (2, 4, 16, 64)

Aquesta ocasió ha estat possible unificar els quatre kernels Stride degut a que el comportament obtingut en tots ells ha estat molt similar tant en CUDA com en OpenACC.

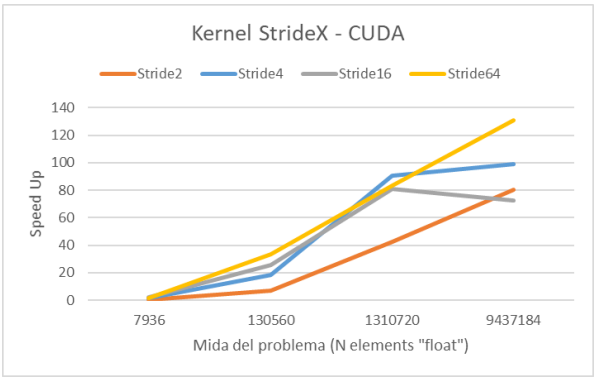

Fig. 21: Resultats obtinguts en CUDA al Kernel_StrideX.

No obstant, tot i que els accessos realitzats a memòria no són consecutius els speedups aconseguits són favorables, sobretot amb mides de problema grans.

TAULA 4: OpenACC Speed up de Kernel_Stride

Mida N float	**Stride2**	**Stride4**	**Stride16**	**Stride64**
7936	0.41	1.09	1.98	1.83
130560	5.89	16.03	23.37	31.47
1310720	32.52	74.41	75.33	76.60
9437184	87.22	96.41	71.96	132.23

Kernel_Rows

Finalment, el Kernel_Rows es tracta d'operacions vectorials, tot i que a diferència del kernel_Copy, els accessos a memòria es realizen de manera consecutiva, això causa una reducció del temps d'execució a nivell de GPU i, per tant, assolint *SpeedUps* majors.

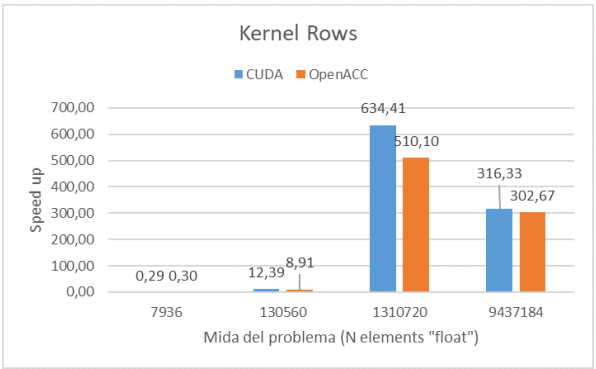

Fig. 22: Resultats obtinguts al Kernel_Rows.

Total de threads per kernel

Els temps d'execució anteriorment presentats són el resultat de les versions implementades que després de certa sintonització han obtingut els millors rendiments. A la Taula 5 s'han recollit el nombre de threads per bloc utilitzats en cadascun dels kernels que composen el dataset, tan en CUDA com en OpenACC.

TAULA 5: Nombre de threads per bloc utilitzats a cada kernel

	CUDA			OpenACC		
	X	Y	Z	X	Y	Z
Copy/Scale/Add/Triad	1024	-	-	128	-	-
Reduction	128	-	-	128	-	-
2PStencil	128	-	-	128	-	-
2S4PStencil	16	16	-	16	16	-
Stencil	8	8	8	16	4	32
MatXVec	64	4		128	-	-
MatMult	16	16	-	8	8	-
MatMultNoOpt	16	16	-	16	16	-
Stride (2, 4, 16, 64)	1024	-	-	128	-	-
Rows	256	-	-	1	128	-

Transferència de dades entre Device i Host

Atès que els temps d'execució presentats no compten amb el temps necessari per a la transferència de dades bidireccionals entre Host i Device, no són del tot realistes. No obstant, s'ha decidit exposar-los d'aquesta manera ja que en un programa real s'hauria de tenir en compte el nombre de kernels a executar i si aquests poden reutilitzar dades ja existents al Device.

La Taula 6 mostra la quantitat de dades a transferir segons la mida del problema i el temps que tarda CUDA i OpenACC en realitzar aquestes transferències. També es pot observar com CUDA va molt més ràpid a fer les transferències que OpenACC.

TAULA 6: Temps de transferència de dades Device-Host

Mida N *float*	**Mida transferència (kB)**	**CUDA (ms)**	**OpenACC (ms)**
7936	31	0.04	9.48
130560	510	0.16	8.75
1310720	5,120	1.18	10.87
9437184	36,864	6.97	21.52

Si afegim els temps de transferència de dades als temps d'execució dels kernels MatXVec, MatMult i MatMultNoOpt, que ja s'ha observat anteriorment que tenen *speedups* molt diversos entre ells, es pot veure a la Taula 7 que l'acceleració assolida s'ha vist minvada considerablement, especialment en el cas de OpenACC.

TAULA 7: Speed up amb transferència de dades

Mida	**MatxVect**		**MatMult**		**MatMultNoOpt**	
N float	CUDA	OpenACC	CUDA	OpenACC	CUDA	OpenACC
7936	0.20	0.001	3.37	0.02	3.23	0.02
130560	1.09	0.02	53.20	1.26	49.86	1.07
1310720	1.17	0.13	133.40	24.25	327.53	43.79
9437184	1.34	0.44	234.03	167.63	1066.52	467.84

En general, es pot observar que pel kernel MatxVect, l'execució és igual o molt més lenta que en CPU, i passa igual en els altres kernels amb la mida de problema més petita. Per altra banda, la parel·lització en GPU segueix sent efectiva en els kernels MatMult i MatMultNoOpt per mides de problema grans.

6.2 Anàlisi dels resultats

En general, cal destacar que s'ha assolit una acceleració del rendiment similar en tots aquells kernels que realitzen operacions vectorials o lineals. En aquests casos, on la reutilització de memòria és mínima, s'ha pogut observar que OpenACC i CUDA són capaços d'obtenir resultats similars. En canvi, en problemes on la complexitat la marcaven els accessos a memòria, s'ha pogut observar en OpenACC una relació directa entre les optimitzacions d'accessos a memòria en caché (MatMult) i l'augment del rendiment en comparació amb CUDA. No obstant, en el MatMultNoOpt, CUDA ha tret molt més rendiment i, la única diferència entre els kernels és la manera en com s'accedeix a memòria. Aquest fet també es pot apreciar en els kernels Stencil, on CUDA té un Speed-up major en el cas de les tres dimensions, però OpenACC mostra millors resultats en dues (2D4PStencil).

D'altra banda, s'ha pogut observar que la mida del problema influeix considerablement en el rendiment final de les execucions. Generalment, les mides petites resulten més lentes en GPU que en CPU, exceptuant els casos en que per la naturalesa del problema els accessos a memòria solen ser nombrosos i ineficients per a la CPU, com en els kernels MatMult o MatMultNoOpt. En aquests casos, la GPU assoleix millors resultats en totes les mides de problema.

Finalment, com s'ha pogut observar a l'apartat anterior, el temps de transferència de dades té un gran impacte en el rendiment final obtingut. També s'ha pogut apreciar que aquestes transferències prenen molt més temps en el cas de OpenACC que en CUDA. No obstant això, es pot concloure que en aplicacions on es puguin reutilitzar dades dins del Device entre diferents problemes o kernels, reduint així el nombre de transferències entre Device i Host, es poden aconseguir acceleracions molt elevades.

Aquesta resolució es podria confirmar amb la llei d'Amdahl (Equació 1), la qual assegura que "*el màxim Speed-up que es pot aconseguir d'una aplicació està limitat pel fragment de codi que no es pot paral·lelitzar*". Així doncs, la llei de Gustafson va afegir una limitació al Speed-up d'Amdahl, afirmant que "*si la mida del problema varia, la part sèrie no ho farà*"[25]. Amb això, es recolza la conclusió extreta anteriorment, on es determina que si es reduïssin les transferències de dades els resultats millorarien, ja que són la part no paral·lelitzable dels kernels.

$$Speed - Up = \frac{1}{f + \frac{(1-f)}{p}} \quad (1)$$

On f és la fracció de programa no paral·lelitzable i p és el nombre de processadors a utilitzar.

7 Conclusions

Un cop finalitzat el projecte, s'ha pogut comprovar que s'han completat tots els objectius, exceptuant la part d'introduir nous kernels al dataset malgrat haver realitzat la recerca. No obstant, s'han assolit tot i els inconvenients soferts arran de l'atac informàtic patit per la UAB i que això provoqués el replantejament de la major part de la planificació, incloent-hi la realització del muntatge d'un setup casolà per tal de poder continuar amb el desenvolupament del projecte i la implementació d'un motor d'execució. Aquest fet també demostra que haver escollit la metodologia Agile ha estat una bona decisió, ja que ha permès realitzar els canvis esmentats a la planificació el més aviat possible i així poder continuar amb aquest projecte reduint els obstacles.

Per una banda, els resultats obtinguts en acceleradores demostren que els patrons amb gran quantitat d'accessos a memòria tenen un impacte crític en el rendiment dels nuclis de còmput i que la mida del problema repercuteix greument en el rendiment. Per altra banda, els temps obtinguts demostren, tal com s'esperava, que amb CUDA s'assoleixen Speed-ups més destacables per norma general que en OpenACC, exceptuant per mides petites de problema.

D'aquesta manera es pot concloure que OpenACC és molt més senzill i ràpid d'implementar que CUDA, però

requereix una exhaustiva avaluació i una selecció idònia de les clàusules a utilitzar. En canvi, CUDA dona molta més llibertat a l'hora de programar i control del comportament del kernel dins de la GPU.

També es pot confirmar l'afirmació donada en [15] esmentada a l'estat de l'art, i és que el temps de transferència és inferior en CUDA que en OpenACC, això fa que no s'obtinguin millores tan elevades com en CUDA. Malgrat tot, si s'utilitzen les clàusules ideals per a cada problema, OpenACC pot aconseguir igualar el rendiment de CUDA en la major part dels casos.

Finalment, cal esmentar que com a treball futur quedaria acabar de completar el ventall de micro-kernels del dataset actual. Addicionalment es podria fer una revisió minuciosa de la implementació dels kernels en CUDA donat que possiblement s'hi pugui realitzar alguna aportació que desencadeni en una millora respecte a l'obtingut en aquest projecte.

Agraïments

Un especial agraïment al meu tutor i assessors, Eduardo César i Anna Sikora, tant per l'ajuda i l'orientació que m'han brindat durant tot el projecte com per oferir-me la possibilitat de fer un tastet del món de la recerca universitària. Finalment, també m'agradaria agrair a la meva parella el suport que m'ha donat durant aquests darrers mesos.

Referències

[1] "What is HPC? Introduction to high-performance computing — IBM." https://www.ibm.com/topics/hpc. Últim accés: 2021-10-07.

[2] Intel, "What Is a GPU? Graphics Processing Units Defined." https://www.intel.co.uk/content/www/uk/en/products/docs/processors/what-is-a-gpu.html, 2021. Últim accés: 2021-10-07.

[3] "OpenACC Getting Started Guide Version 20.4 for x86 and NVIDIA Processors." https://docs.nvidia.com/hpc-sdk/pgi-compilers/20.4/x86/openacc-gs/index.htm. Últim accés: 2021-10-07.

[4] NVIDIA Corporation, "CUDA C++ Best Practices Guide Design Guide." https://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf, 2022. Últim accés: 2021-12-21.

[5] NVIDIA, "About Nvidia." http://www.nvidia.com/object/about-nvidia.html. Últim accés: 2021-10-07.

[6] J. C. Bermúdez-Rodríguez, "Adaptació de kernels d'OpenMP a GPGPU — Universitat Autònoma de Barcelona," 2020. Últim accés: 2021-10-07.

[7] Universitat Autònoma de Barcelona, "Departament d'Arquitectura de Computadors i Sistemes Operatius." https://www.uab.cat/web/departament-d-arquitectura-de-computadors-i-sistemes-operatius-1345733222033.html, 2021. Últim accés: 2021-10-07.

[8] "Computació avançada per als reptes de la societat digital (CAROL) (Ministerio de Ciencia e Innovación bajo contrat PID2020-113614RB-C21) — UAB," 2020. Últim accés: 2021-10-07.

[9] J. Alcaraz, A. Sikora, and E. César, "Hardware counters' space reduction for code region characterization," in *Euro-Par 2019: Parallel Processing* (R. Yahyapour, ed.), (Cham), pp. 74–86, Springer International Publishing, 2019. Últim accés: 2021-10-07.

[10] A. Panesar, "What Is Machine Learning? Machine Learning and AI for Healthcare." https://www.ibm.com/cloud/learn/machine-learning, 2019. Últim accés: 2021-10-07.

[11] A. Martinez, A. Sikora, E. Cesar, and J. Sorribes, "How to scale dynamic tuning to large parallel applications," in *Proceedings of the 2013 IEEE 27th*, IPDPSW '13, (USA), p. 355–364, IEEE Computer Society, 2013. Últim accés: 2021-10-07.

[12] J. Alcaraz-Rodriguez, S. Sleder, A. TehraniJamsaz, A. Sikora, A. Jannesari, J. S. Gomis, and E. Cesar-Galobardes, "Building a Dataset for Classifying OpenMP Parallel Patterns via Machine Learning." https://doi.org/10.5281/zenodo.3865286, May 2020. Últim accés: 2021-10-07.

[13] NVIDIA Corporation, "High Performance Computing (HPC) SDK — NVIDIA." https://developer.nvidia.com/hpc-sdk. Últim accés: 2022-01-04.

[14] T. Hoshino, N. Maruyama, S. Matsuoka, and R. Takaki, "CUDA vs OpenACC: Performance Case Studies with Kernel Benchmarks and a Memory-Bound CFD Application," in *13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, 2013. Últim accés: 2022-01-04.

[15] X. Li and P.-C. Shih, "Performance Comparison of CUDA and OpenACC Based on Optimizations," in *Proceedings of the 2018*, HPCCT 2018, p. 53–57, Association for Computing Machinery, 2018. Últim accés: 2022-01-04.

[16] X. Li and P.-C. Shih, "An Early Performance Comparison of CUDA and OpenACC," *MATEC Web of Conferences*, vol. 208, p. 05002, 01 2018. Últim accés: 2021-12-21.

[17] A. Bjork, "What is Agile? - Azure DevOps — Microsoft Docs." https://docs.microsoft.com/en-us/azure/devops/learn/agile/what-is-agile, 2017. Últim accés: 2021-10-07.

[18] "Lenovo L340 Gaming Laptop." https://www.lenovo.com/us/en/p/laptops/ideapad/ideapad-gaming-laptops/ideapad-l340-15irh-gaming/88ipl301161. Últim accés: 2021-11-08.

[19] Nvidia, "NVIDIA CUDA Installation and Verification on Linux," no. July, 2021. Últim accés: 2021-11-08.

[20] "Installation Guide Linux :: CUDA Toolkit Documentation." https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html. Últim accés: 2021-11-08.

[21] E. Cesar-Galobardes, "Parallel Programming of Massively Parallel Processors. ," pp. 1–46, 2020. Últim accés: 2021-10-07.

[22] A. S.-S. Fong, K. Y. Wu, R. Chen, and L.-M. Cheng, "A heterogeneous multiprocessing computer system with shared memory," *Proceedings of TENCON '93*. Últim accés: 2021-11-08.

[23] M. Harris, "Using Shared Memory in CUDA C/C++ — NVIDIA Developer Blog." https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/, 2013. Últim accés: 2021-11-07.

[24] Jeff Larkin, "7 Powerful New Features in OpenACC 2.0." https://developer.nvidia.com/blog/7-powerful-new-features-openacc-2-0/. Últim accés: 2022-01-10.

[25] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, 2008.

APÈNDIX A: GRÀFICS D'INTERÉS

A.1 SpeedUp dels diversos micro-kernels

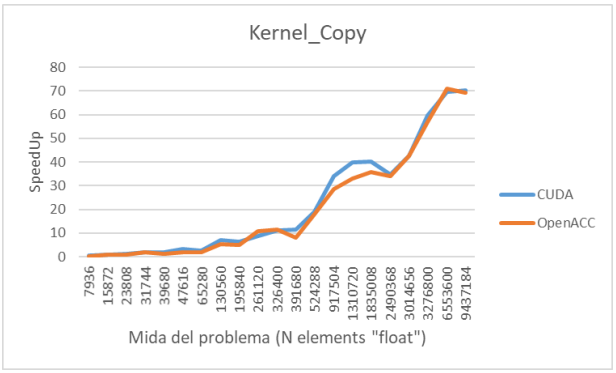

Fig. A.1: SpeedUp de Copy per diverses mides de problema

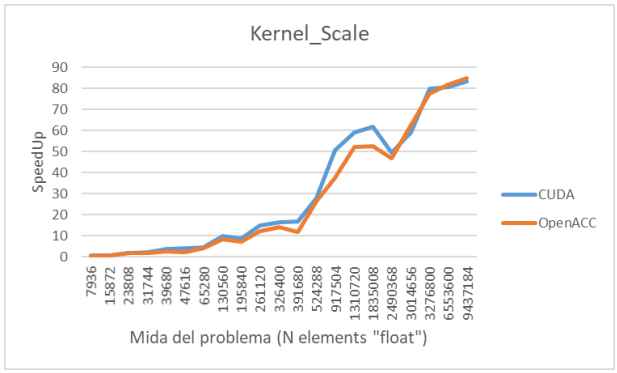

Fig. A.2: SpeedUp de Scale per diverses mides de problema

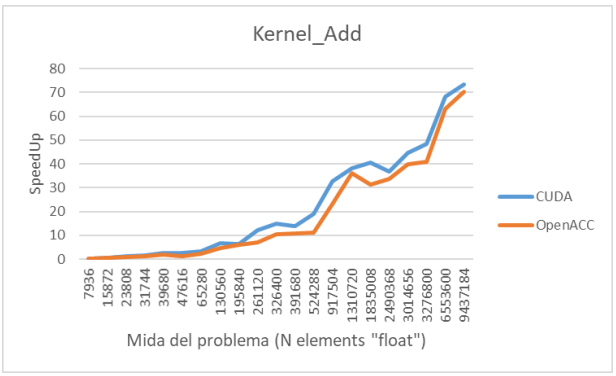

Fig. A.3: SpeedUp de Add per diverses mides de problema

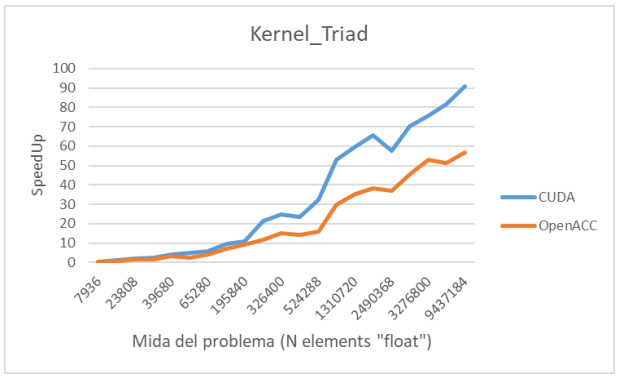

Fig. A.4: SpeedUp de Triad per diverses mides de problema

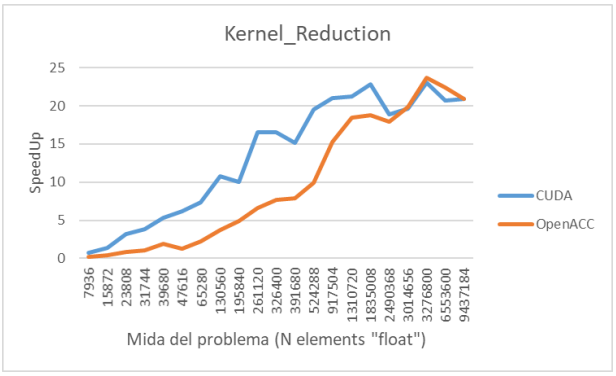

Fig. A.5: SpeedUp de Reduction per diverses mides de problema

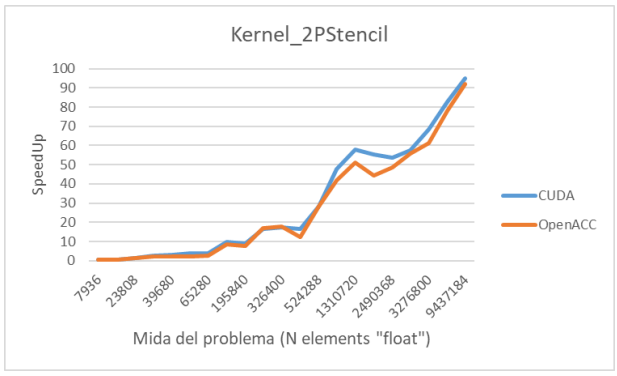

Fig. A.6: SpeedUp de 2PStencil per diverses mides de problema

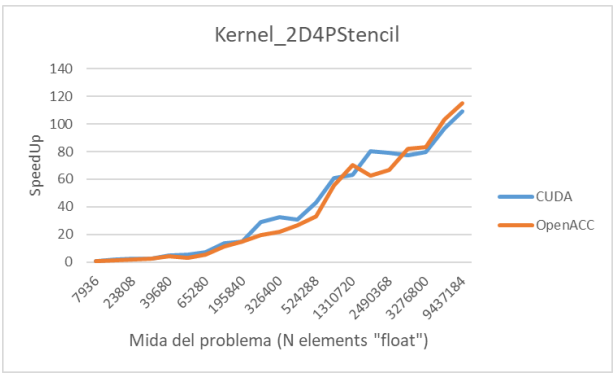

Fig. A.7: SpeedUp de 2DPStencil per diverses mides de problema

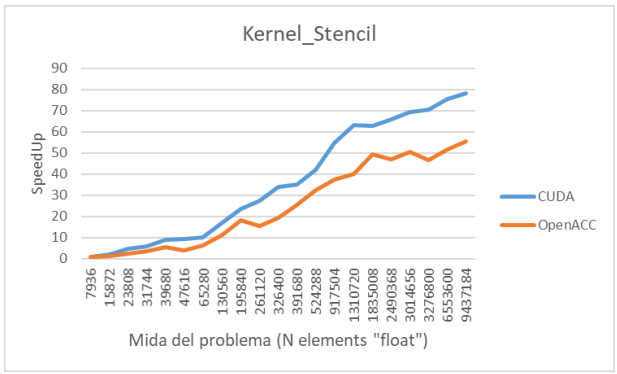

Fig. A.8: SpeedUp de Stencil per diverses mides de problema

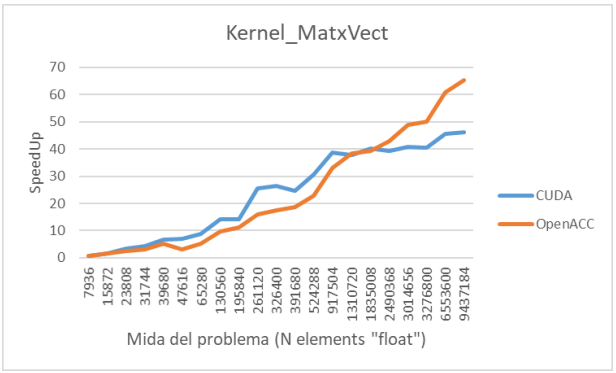

Fig. A.9: SpeedUp de MatxVect per diverses mides de problema

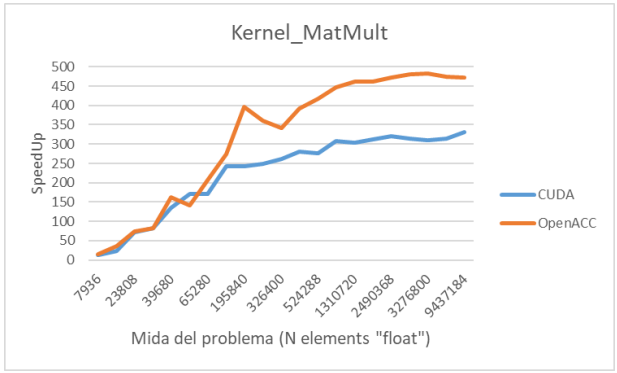

Fig. A.10: SpeedUp de MatMult per diverses mides de problema

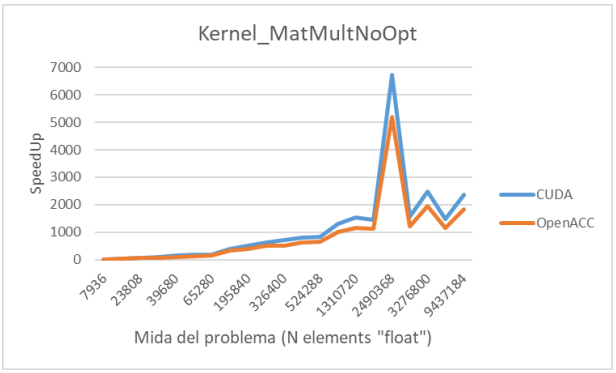

Fig. A.11: SpeedUp de MatMultNoOpt per diverses mides de problema

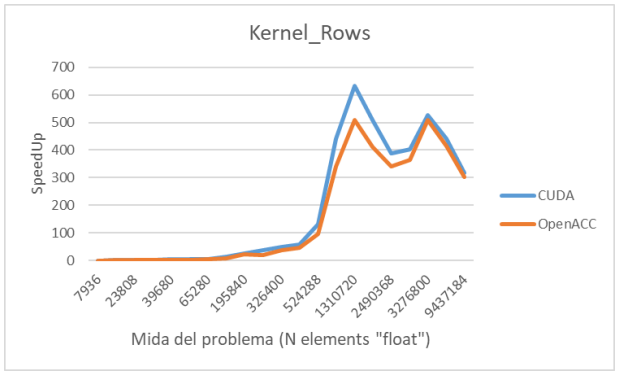

Fig. A.12: SpeedUp de Rows per diverses mides de problema

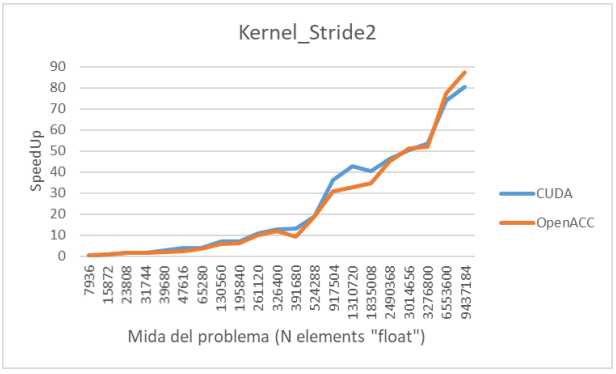

Fig. A.13: SpeedUp de Stride2 per diverses mides de problema

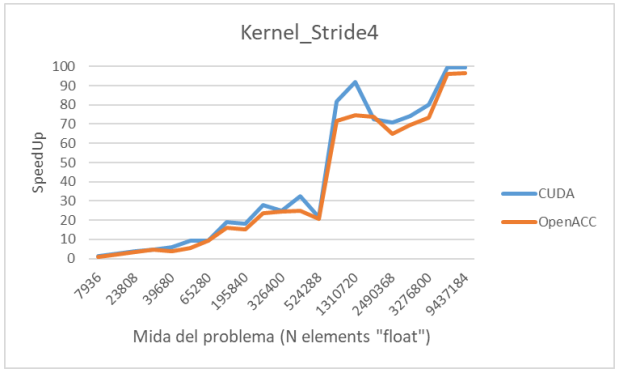

Fig. A.14: SpeedUp de Stride4 per diverses mides de problema

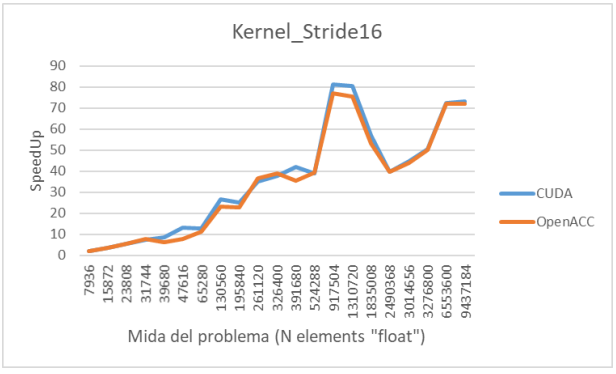

Fig. A.15: SpeedUp de Stride16 per diverses mides de problema

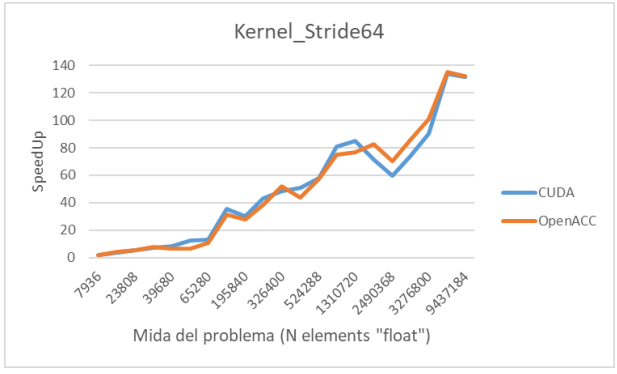

Fig. A.16: SpeedUp de Stride64 per diverses mides de problema

A.2 Temps de transferències entre Host i Device

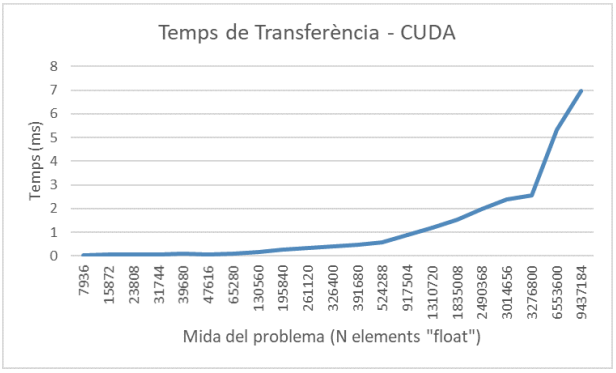

Fig. A.17: Temps de transferències per CUDA

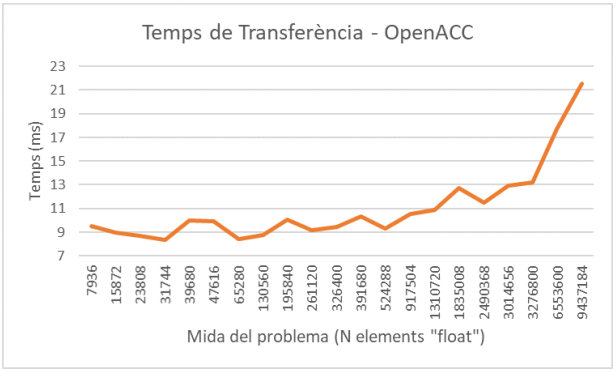

Fig. A.18: Temps de transferències per OpenACC