Tema 2a

Introducció a CUDA

Programació paral·lela

Curs 2018 - 2019

Oscar Amorós

Paral·lelisme de dades massiu: CUDA

Introducció

Com dèiem al tema 1, les GPU's han passat de ser un procesador només per a gràfics, a ser un coprocessador o accelerador de càlculs matricials i vectorials, amb un ús molt estès en molts àmbits.

CUDA, va representar una revolució, ja que presentava una forma molt fàcil d'accedir a tota aquesta capacitat de càlcul. Actualment, ja en la versió 10.1, CUDA té una API molt extensa, i un conjunt de llibreries enorme i cada cop més gran.

Tot i amb això, CUDA no és l'únic llenguatge per programar les GPU's, per a propòsit general. La industria ha creat diversos llenguatges com ara OpenCL, HSA, SpirV i Vulkan. Aquest últim, més centrat en els gràfics, però també permet crear codi de càlcul.

En aquest curs, veurem els conceptes més comuns a totes les GPU's, utilitzant CUDA com a exemple per a teoria i pràctiques, ja que és el llenguatge més estès.

Tota la informació que necessiteu per a aquest tema, i més, la podeu trobar a: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Visió general del hardware GPU i del software (llenguatges)

Totes les GPU's tenen una organització general que és la mateixa. Utilitzarem nomenclatura de CUDA per a minimitzar confusions.

Aquesta organització es basa en el que podeu veure en la següent imatge:

Figura 1: Diagrama de blocks del chip Tesla P100

Ah, que no s'entén res? Bé, us ho explico. Esteu veient 3584 nuclis, d'una sola GPU NVIDIA Tesla P100. De fet, aquests 3584 nuclis son de precisió simple (float 32 bits), en color verd. A aquests se'ls ha d'afegir 1792 nuclis de precissió doble (double 64 bits), en color groc.

Aquests nuclis de càlcul, comercialment anomenats CUDA cores, son molt més simples que un nucli de CPU. Per això n'hi han tants. Tant mateix, estan organitzats en grups anomenats SM (Streaming Multiprocessros). Fem zoom i veiem-ne un.

Figura 2: Diagrama de blocs de un Streaming Multiprocessor

Aquest SM conté 64 CUDA cores de 32 bits (Core en verd), i 32 de 64 bits (DP Unit en groc). A la vegada, aquests SM estan organitzats en GPC (Graphics Processing Cluster).

El que a nosaltres ens interessa com a programadors, és conèixer el model de programació, el model d'execució, i la jerarquia de memòria. Aquests, són comuns (o molt similars) per a CUDA, OpenCL, HSA i Vulkan, tant per a GPU's de NVIDIA, AMD, Intel, ARM (Mali GPU), Imagination Technologies (PowerVR GPU), Qualcom (Adreno GPU), etc...

Per acabar de donar-li un punt més de complexitat, el programming model, fa referència no només al codi que s'executa a la GPU, si no que també fa referència a tot el codi que cal escriure per controlar la interacció de la GPU amb la CPU. El codi que s'executa a la GPU s'anomena Device Code, i el que s'executa a la CPU (incloent codi de control de transferencies entre CPU i GPU, i enviament de execució de kernels a la GPU) es diu Host Code. Aquesta nomenclatura es manté tant per a CUDA com per la resta de llenguatges.

Software a la GPU (Device code)

En aquest apartat parlarem del codi que s'executa a la GPU, i de tot allò que cal saber per poder entendre i programar codi de Device.

Model de programació (Device code)

Anem a parlar del model de programació. Per començar, el codi que s'executa a la GPU (Device Code), està basat en C, i en algunes característiques de C++. Es a dir, podem treballar amb objectes a la GPU, i podem fer altres coses avançades de C++ a la GPU com utilitzar templates i variadic templates. Per tant, no tenim disponible ni tot estándar ANSI C ni el de C++. Només un subconjunt d'aquests llenguatges.

A partir d'aquí, el canvi més notori ve per entendre qui executa el codi que escrivim.

Habitualment, el codi que escrivim, s'executarà en un sol fil o procés. A no ser que passem una mateixa funció com a funció de més d'un thread. De fet, aquest seria el símil ideal per interpretar el que passa amb el Device Code. Quan cridem una funció de Device (que s'anomena Kernel), abans de fer-ho, configurem el nombre de threads que executaràn aquest kernel.

Aquest nombre habitualment, per a poder aprofitar les capacitats de les GPU's, se situa en l'ordre dels milions de threads. Si, has llegit bé, al voltant de 1.000.000 de threads per executar una sola funció. Si no s'utilitzen tants threads, és més difícil aprofitar tots els recursos de la GPU.

Per a que tot aquest paral·lelisme tingui sentit, lo ideal es que cada thread treballi sobre dades diferents, i no repetir la feina. Això és possible gràcies a que cada thread the un identificador. Amb aquest identificador, podem accedir a posicions de memoria diferents. Per exemple:

```
for (int i=0; i<N; ++i) {
        vector[i] = vector[i]*vector[i];
}
```

En aquest cas, podem substituir el bucle per els índexs dels threads:

```
vector[threadIdx.x] = vector[threadIdx.x] * vector[threadIdx.x];
```

La variable threadIdx es un struct que conté el id de cada thread. Aquest id, te fins a tres dimensions i s'accedeixen per les components "x", "y" i "z". Habitualment només s'utilitzen una o dues dimensions, i per simplicitat, en aquesta assignatura l'utilitzarem 2 (x i y).

```
struct threadIdx { int x; int y; int z; }
```

Ara bé. No és tan simple com això. Resulta que per motius que explicarem al model d'execució, els threads de CUDA s'agrupen en "thread blocks". I resulta que cada GPU imposa un límit en el nombre total de threads de CUDA, tenint en compte totes les dimensions. Es a dir, si el número de threads a l'eix de les x es "num_x", el mateix per a "num_y" i "num_z", llavors el total es: num_x * num_y * num_z . Habitualment (con en el cas de les Geforce GTX 1060 que teniu a l'aula ID) aquest límit es de 1024 threads per thread block.

Per a més detalls sobre les capacitats de les GPU's que teniu disponibles a l'aula ID, consulteu el següent link i mireu la columna que correspon a la Compute Capability 6.1.
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities

Això vol dir, que si en el bucle anterior, N > 1024, necessitarem més d'un block de threads per processar tot el bucle.

```
int N = 2048;
for (int i=0; i<N; ++i) {
        vector[i] = vector[i]*vector[i];
}
```

Aquests blocks viuen en un espai anomenat GRID. Per tant, resumint, un GRID és un espai que pot ser 1D, 2D o 3D. En aquest espai existeixen "thread blocks" que tenen els seus id's en les 3 dimensions. Per exemple, quan el grid es 1D, les mides dels blocs son (n, 1, 1) on n es la dimensiò x que depén del nombre de blocs total. Aquests blocks al mateix temps, contenen threads que tenen els seus id's també en les 3 dimensions. De la mateixa manera, els threads d'un thread block 1D, tindràn la mida (n, 1 , 1) on n dependrà del nombre total de threads en aquest bloc.

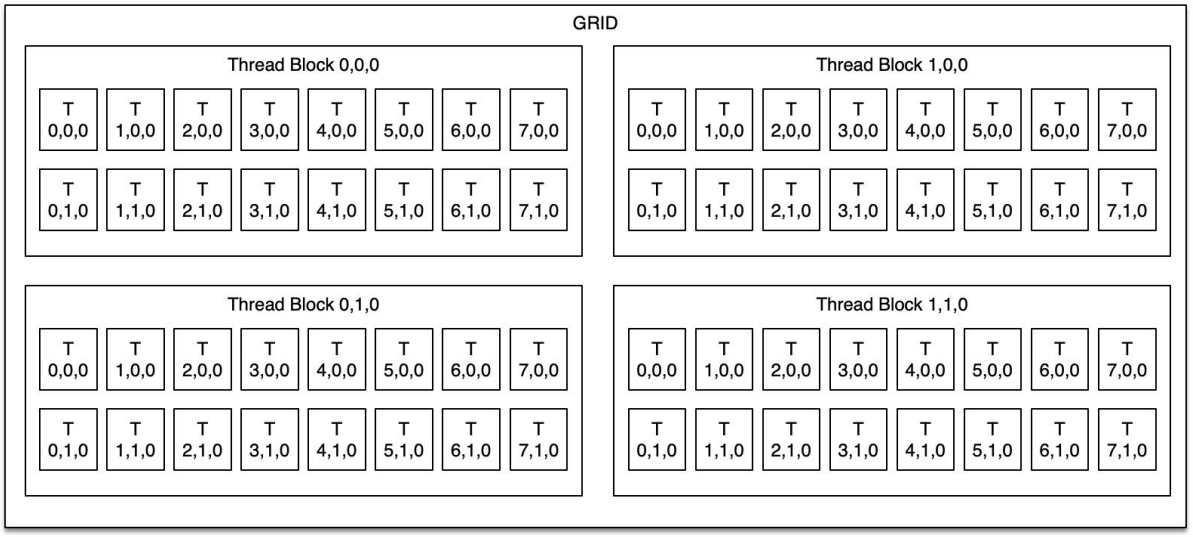

Figura 3: GRID 2D, amb Thread Blocks 2D amb els id's de block i de thread (T).

Paral·lelitzat en CUDA el bucle anterior, en un grid 1D amb thread blocks 1D, seria:

```
int N = 2048;
int thread_position = threadIdx.x + blockIdx.x * blockDim.x;
vector[thread_position] = vector[thread_position]  * vector[thread_position];
```

"blockIdx" és un struct que conté la informació de id de cada block, i "blockDim" es un struct que conté la informació de quants threads hi ha a dins de cada thread block, per a cada dimensió, x, y i z.

Seguint l'exemple de la Figura 3, la variable blockIdx del block situat a la cantonada superior esquerra, tindrà els id's (x,y,z) = (0,0,0). La variable blockDim en canvi serà igual per a tots els blocks i tindrà els següents valors: (x, y, z) = (8, 2, 1).
Existeix una altre variable, "gridDim", que en el mateix exemple sería (x,y,x) = (2,2,1).

En el cas del bucle on N = 2048, una possible configuració seria la següent:

gridDim = (4, 1, 1)
blockDim = (512, 1, 1)

4 x 512 = 2048 = N

Però com es configura la quantitat de blocks i de threads per block per a cada GRID? Doncs es fa al codi de Host, que s'executa a la CPU. Ho veurem a l'apartat de Host code.

Jerarquia de memòries.

Les GPU's tenen una jerarquia de memòries força més complexa que la de la CPU, però que a la vegada és molt responsable de la arquitectura que permet fer càlculs paral·lels molt més ràpids que la CPU.

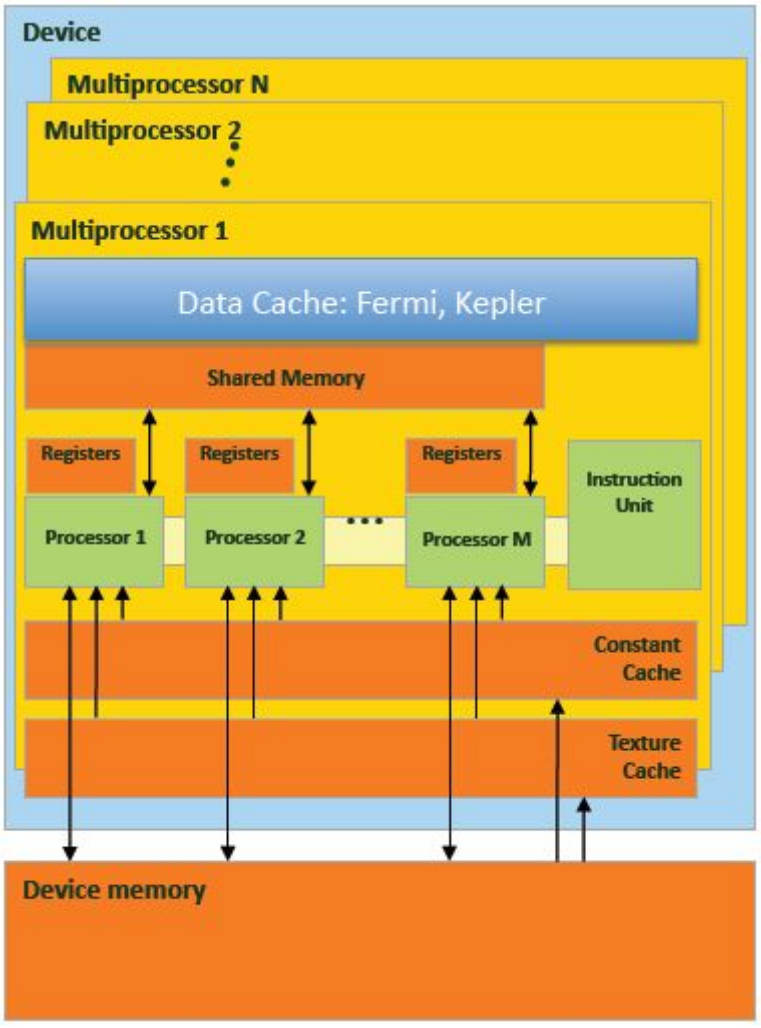

Figura 4: Jerarquía de memoria de la GPU.

Seguint l'esquema de la Figura 4, començarem per parlar de la Device memory. Aquesta memòria, vindria a ser l'equivalent a la memoria RAM de les CPU's. El tipus de memòria més utilitzat és la GDDR 5, tot i que ja existeix la versió 6, i també hi han GPU's que incorporen un altre tipus de memoria molt més ràpida, però també més cara, anomenada HBM 2 (High Bandwith Memory).

En general, el que heu de saber d'aquestes memories, és que de totes les memòries que disposa la GPU, la Device memory és la més gran, i la més lenta. Sol tenir una mida que ve de 1GB a 48GB, i uns amples de banda de 100Gbps a 1Tbps.

No només és la més lenta en quant a ample de banda (bits transferits per segon), si no que a més accedir-hi té una latencia molt superior que qualsevol de les altres memories.

Que vol dir "latencia"? Per exemple, per transferir 10MB, si tenim un ample de banda de 400Gbps, podem calcular el temps necessari per transferir aquests 10MB, suposant que aprofitem tot l'ample de banda disponible (cosa que ja veurem que depén del codi). Però a aquest temps, li hem de sumar el temps degut a la latencia. Per a cada transferencia, independentment de si son 10MB o 10GB, li hem de sumar el temps de latencia, que és el temps necessari per a iniciar la transferencia, que implica amplificacions de senyal, traducció i codificació de bits en senyals electromagnètiques complexes, etc…

Aproximadament, el temps perdut en latencia per a cada transferencia, equival a uns 600 cicles de rellotge de la GPU. Si una instrucció de suma (per a 32 parells de numeros), requereix uns 5 cicles, podeu imaginar lo car que és accedir a aquesta memória.

És per això, que la majoria d'optimitzacions en qualsevol kernel de CUDA se centren en minimitzar els accessos de memoria Device, i a fer-los de la forma més correcta possible, per maximitzar l'ample de banda.

Com a últim apunt sobre la Device memory, tots els threads de un mateix GRID, tenen accés la informació emmagatzemada en aquesta memòria.

Una altre memòria molt important a les GPU's és la Shared memory. Aquesta memòria, és molt més petita, però molt més ràpida i propera als nuclis de càlcul. De fet, vindria a ser com una memòria cau, que en lloc d'estar fora del chip com la Device memory, està a dins, però amb la diferència respecte a la memòria cau, de que no té cap mecanisme de coherencia, de manera que ocupa molt menys espai que la memòria cau.

En aquest cas, parlem de mides de 48KB a 96KB, depenent del model de GPU. L'ample de banda es de fins a 200Tbps, i la latència és de uns 2 cicles de rellotge per accés. En quant a latencia és virtualment gratis.

En quant a la visibilitat d'aquesta memòria respecte dels threads, és força més complexe. Primer hem de saber, que els nuclis de les GPU's s'agrupen en SM (Streaming Multiprocessors). Aquests SM son l'equivalent als Thread Blocks en quant al software. Per

tant, un Thread Block s'executa en un sol SM, i un SM pot executar entre 16 i 32 Thread Blocks al mateix temps.

Si un SM te 64KB de memoria Shared, i tenim 16 Thread Block disponibles per ser executats en aquest SM, això vol dir que tindrem 4KB de Sared memory disponibles per a cada TB. Si necessitem utilitzar més shared memory, podem, però llavors aquest SM tindrà menys TB concurrents executant-se.

Al model d'execució explicarem perquè és bo tenir el màxim possible de TB executant-se concurrentment a cada SM.

Ara, parlem de la visibilitat. Cada TB veu només els 4KB (o el que sigui que utilitzi) de Shared memory. Cap TB pot accedir a la Shared memory assignada a un altre TB (això incolu no poder accedir a la Shared de qualsevol altre SM).

La única manera de compartir les dades de la Shared memory amb altres threads es movent les dades a la Device memory.

Finalment, private memory. Aquesta memòria es la memoria del banc de registres. Algunes GPU's arriben a tenir fins a 20MB, sumant tots els bancs de registres. Les CPU's modernes, tenen uns 0,16KB per nucli. Si comparem, estem parlant de 20.000KB vs <1KB.

La conseqüència d'això és que la forma de programar es força diferent. Ho veurem durant el curs.

Constant memory: existeix una porció petita (64KB) de la Device memory, que es pot accedir des de la GPU només per a lectura (i des de la CPU per escriptura) de forma molt ràpida. Personalment, desaconsello el seu ús, perquè és totalment incompatible amb el multithreading. Si dos threads de CPU criden el mateix kernel de CUDA, i aquest kernel utilitza Constant memory, i els dos kernels necessiten valors diferents en aquesta constant memory, els resultats seran aleatòriament erronis.

Texture memory: aquesta memòria es força diferent d'utilitzar, ja que no s'hi accedeix per posicions discretes de memoria, si no per posicions definides per números decimals del 0 al 1. El seu funcionament també és força diferent. Existeix a la GPU una memoria cau dedicada a la memoria de textura, que desa blocs de memoria en 2D, de manera que els accessos que no siguin contigus en 1D, però que siguin propers en 2D, seran molt més ràpids utilitzant la memoria de textura que la memoria normal. Una altra característica de la memoria de textura, és que al demanar una posició de memòria, retornarà un valor aproximat als punts més propers a aquesta posició demanada, tenint en compte totes les dimensions utilitzades. Aquest procés s'anomena interpolació. S'utilitza molt per a fer zoom in i zoom out digital en imatges, en aplicacions de processament de imatges.

Model d'execució.

El model d'execució, explica com es comporten els CUDA Threads a l'hora d'executar accessos a memoria i instruccions de càlcul, com se sincronitzen entre ells, etc…

Primer de tot, a l'hora d'executar-se, els threads d'un mateix Thread Block, s'organitzen en grups de 32 threads, contigus en la dimensiò de les x. Aquests grups s'anomenen Warps.

Si el Thread block té menys de 32 threads en la dimensiò x, es crearán warps incomplets, on els threads que falten, en realitat seràn recursos no utilitzats. Això és degut a que els Stream Multiprocessors internament, executen instruccions vectorials de 32 valors. Per tant, un warp és com un thread que executa en serie, un conjunt d'instruccions vectorials de 32 operacions paral·leles, per a dades diferents.

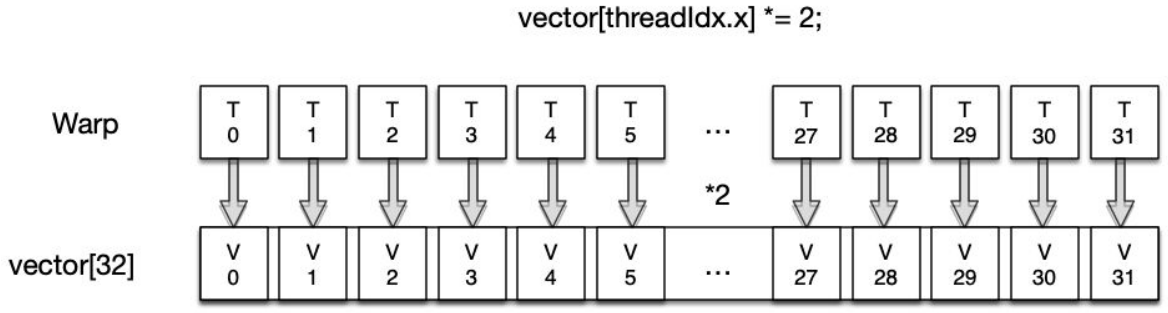

Figura 5: Warp accedint a un vector per realitzar en paral·lel la multiplicació per 2 de cada element del vector.

Per exemple, a la Figura 5 podem veure un warp executant una de les possible instruccions que pot tenir assignades a dins d'un programa.

Per tant, podem asumir, que tots els threads dins d'un warp, per defecte, estarán sempre sincronitzats entre ells. Tot i que en les últimes arquitectures (Volta i Turing), això canvia. De totes maneres, com que podem compilar en mode de compatibilitat, on aquesta assumpció segueix sent vàlida, no explicarem aquests canvis, que complicarien molt el temari, i tampoc teniu disponible les GPU's per a poder provar-ho.

Ara bé, podem sincronitzar els threads de dos warps diferents? Si, si aquests formen part del mateix Thread Block. La crida es:

__synchthreads();

Podem sincronitzar els threads de diferents Thread blocks? No (a no se que utilitzem l'arquitectura Pascal o superior, i CUDA 9 o superior). A l'aula, les GPU's son Pascal, però la versiò de CUDA instalada es la 8.

Per tant, l'únic que podem sincronitzar, son els threads de dins d'un threadBlock. Com podem assegurar doncs, que el que escriuen els threads a la memoria Device, no pateix de condicions de carrera i dependències de dades? Doncs basicament, estem forçats a assignar a cada thread un o més punts de la memoria Device, on només aquell thread hi escriurà.

En quant als accessos a memória, els warps també llegeixen 32 elements de 32 bits o 64 bits concurrentment. Aquesta manera de llegir s'anomena coalescent. Si les posicions de memòria llegides pels threads no son contigües, llavors es perd aquesta coalescencia, i es necessitaràn més accessos a memòria per a llegir totes les dades que necessita el warp.

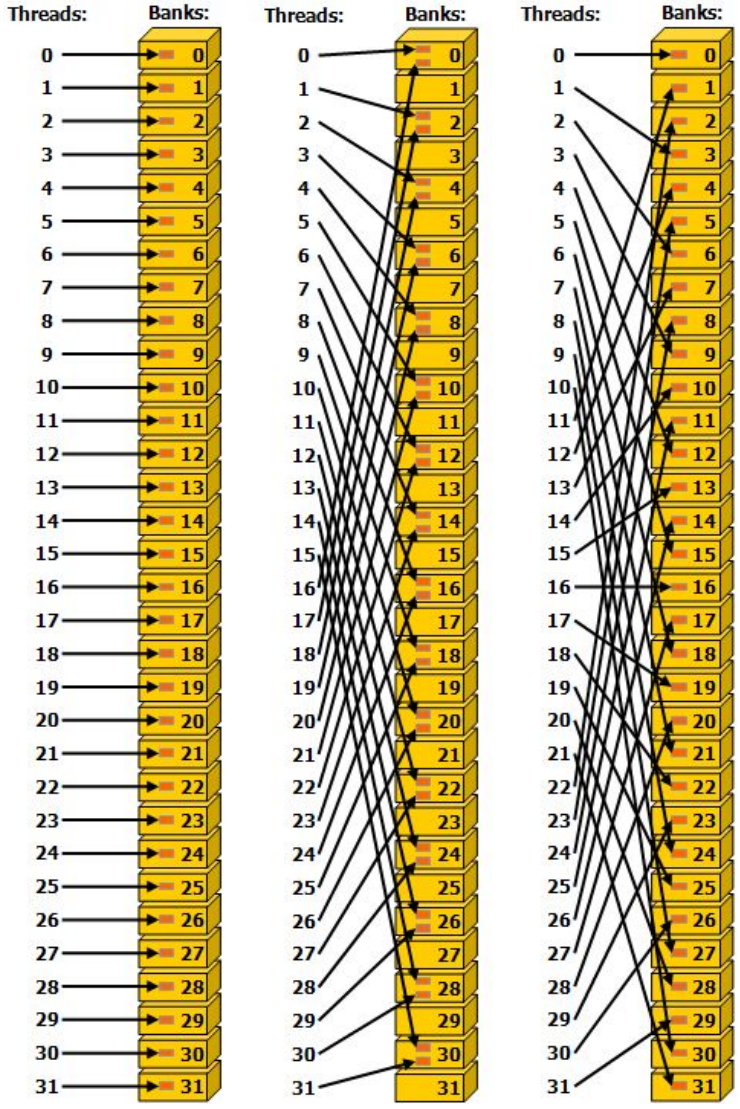

Figura 6: Accesos coalescents (esquerra i dreta) i no coalescents (centre).

A més, la memoria de la GPU está alineada en grups de 32bit o 4 bytes, que es la mida dels int i float. Si llegim un char o uchar, estem perdent ample de banda efectiu. Si cada thread del warp llegeix un uchar3, encara que estiguin en posicions de memoria contigües, com és un tipus de dada que no es múltiple de 4 bytes, es farán 3 accesos a memoria per cada uchar3.

Software de interacció entre la CPU i la GPU (Host code).

Abans de començar aquesta secció, farem una menció a un concepte que és important de cara a consultar la documentació oficial. CUDA, a part d'un llenguatge, és una llibreria amb funcions, llur utilitat explicarem tot seguit. Aquesta llibreria, té dues versions, una de més alt nivell (runtime API) i una altre de més baix nivell (driver API). Nosaltres farem referència només a la runtime API, què és el punt lògic per on començar a aprendre CUDA.

Fent referencia al apartat anterior, els kernels de CUDA s'executen a la GPU, peró en quin moment s'executen, no ho decideix la GPU. De fet, la GPU no decideix res. Es la CPU la que ordena cada una de les accions que passaran a la GPU o entre la CPU y la GPU, mitjançat la runtime API.

La GPU te la seva propia memoria, i per defecte no pot accedir a la memoria de la CPU (tot i que existeixen tecnologies que canvien això). Pertant, les dades que volguem que utilitzi la GPU, s'han de transferir explícitament desde la memoria de la CPU cap a la memoria de la GPU.

La crida d'un kernel, es similar a una crida d'una funció, amb la diferència de que els punters (i només els punters) han d'estar a la GPU. Els paràmetres que no son punters, es poden passar per valor, com a qualsevol altre funció.

La seqüència típica d'un programa en CUDA és la següent:

1. Alocatem memoria a la CPU (ex. punter float* h_A).
2. Alocatem memoria a la GPU (ex. punter float* d_A).
3. Obtenim d'alguna manera les dades a processar a la CPU.
4. Copiem les dades de h_A a d_A.
5. Ordenem l'execució d'un kernel a la GPU, que llegeix i escriu d_A.
6. S'executa el kernel.
7. Copiem els resultats de l'execució del kernel, de d_A a h_A.

D'aquests 6 punts, només el punt 6 succeeix a la GPU. Tota la resta està definit per codi que s'executa a la CPU. Aquest codi, s'anomena codi de Host.

La seqüència previa defineix conceptualment totes les accions possibles, moure dades entre la GPU i la CPU, i executar codi. Però, recordeu que la GPU te una jerarquia de memòries complexa. Per tant, la API de CUDA conté una varietat de crides força gran, per tal de poder interaccionar amb aquesta jerarquia de memòries, i per definir la manera de interaccionar amb la GPU.

A les pràctiques veurem algunes d'aquestes crides, però si voleu veure la especificació completa podeu fer-ho consultant la CUDA toolkit programming guide.

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Asincronisme entre la CPU i la GPU, streams i events.

Una de les propietats de les crides del codi de Host de CUDA, es que poden ser síncrones o asíncrones, respecte del thread de CPU que les crida. La seqüència d'execució que hem mostrat a la introducció d'aquest apartat, asumia que les crides son totes síncrones, i pertant, la CPU no segueix fins que no s'hagi acabat d'executar tot el que demana cada crida. De fet, aquest és el comportament esperat per un codi C/C++ que no utilitzi threads.

Aquesta manera de funcionar té un problema, i es que hi ha un retard des del moment que la CPU demana alguna cosa, fins que aquesta cosa es realitza (copiar dades, o executar un kerne). Si la CPU espera a que s'acabi la crida (i el que implica aquesta crida), llavors al temps total d'execució se li ha de sumar la latencia d'aquestes crides, que pot ser molta.

Per a solucionar aquest problema, i evitar les esperes mútues entre CPU i GPU (la CPU espera que la GPU acabi, i la GPU ha d'esperar a que la CPU li demani les coses), la runtime API de CUDA proporciona un mecanisme per fer les crides asíncrones respecte del thread que les crida. Aquest mecanisme està basat essencialment en els CUDA Streams.

Els CUDA Streams, son principalment ques d'ordres (o crides de la runtime API), que enmagatzement en ordre les crides de la runtime API que la CPU ha executat amb els paràmetres (una copia dels paràmetres, no les variables) que es passin en el moment de fer la crida. Aquests streams son una estructura de dades que permeten a la CPU oblidar-se de les crides que ha fet al runtime, i seguir executant el codi que hem escrit.

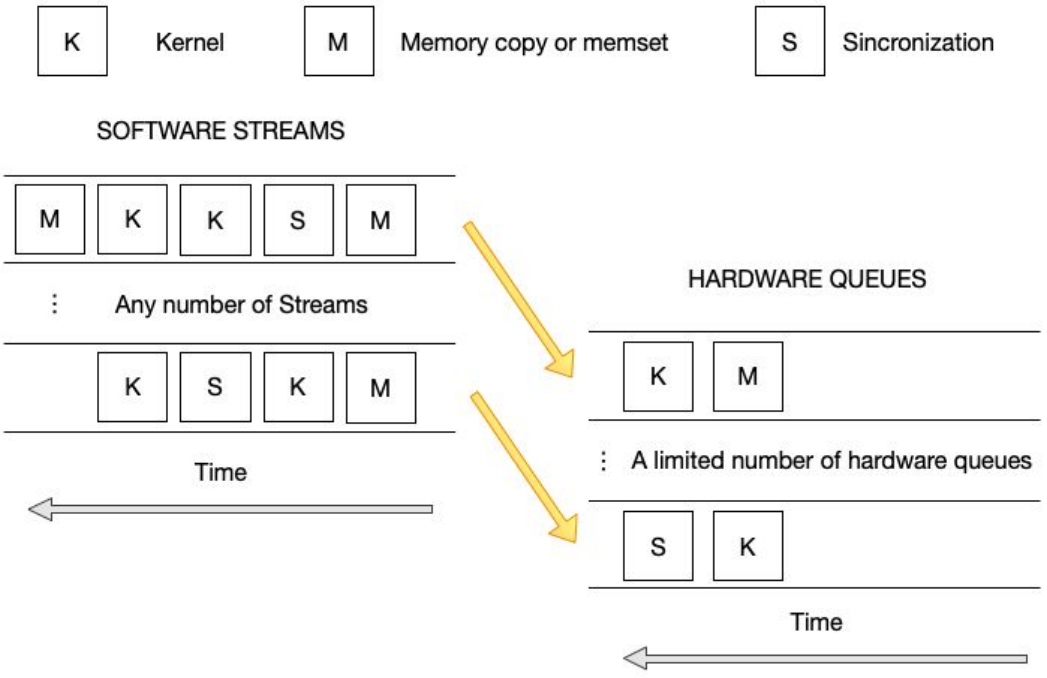

Figura 7: Streams i cues de hardware.

I que passa amb aquestes crides? Bé, el runtime API les executarà. Què és aquest runtime API? Un thread? Bé, de fet les GPU's tenen una mena de equivalent als Streams, però en hardware, de manera que la GPU va buidant els Streams, per colocar les ordres en les seves ques hardware. Tal com podeu veure a la Figura 7.

Per tant, podem pensar que gràcies als Streams, la CPU pot demanar coses al runtime de CUDA, i seguir executant sabent que el que s'ha demanat, ja s'executarà quan toqui.

Molt bé, i que passa quan necessitem els resultats a la CPU? Com sabem si la GPU ha acabat o no? Doncs tenim tres mecanismes per fer que el thread de la CPU s'esperi a que acabi tot el que està demanat a la GPU, o tot el que s'ha demanat a un Stream concret, fins a cert punt del programa.

Aquestes crides son:

- cudaDeviceSynchronize() : el thread de CPU que faci aquesta crida esperarà a que acabi tota la feina encuada a la GPU fins a aquest moment.
- cudaStreamSynchronize(cudaStream_t stream) : el thread de CPU que faci aquesta crida esperarà a que acabi tota la feina encuada a l'Stream "stream" fins a aquest moment.
- cudaEventSynchronize(cudaEvent_t event) : el thread de CPU que faci aquesta crida esperarà a que acabi tota la feina encuada a l'Stream on s'agi registrat l'event "event", i només fins al punt de l'Stream on s'ha registrat l'event.

Un altre mecanisme que es complementari als Streams, son els Events de CUDA. Els Events, permeten sincronitzar streams entre si, i fer que un thread de CPU, esperi l'execució de totes les coses encuades a un Stream, fins a cert punt especificat per l'Event.

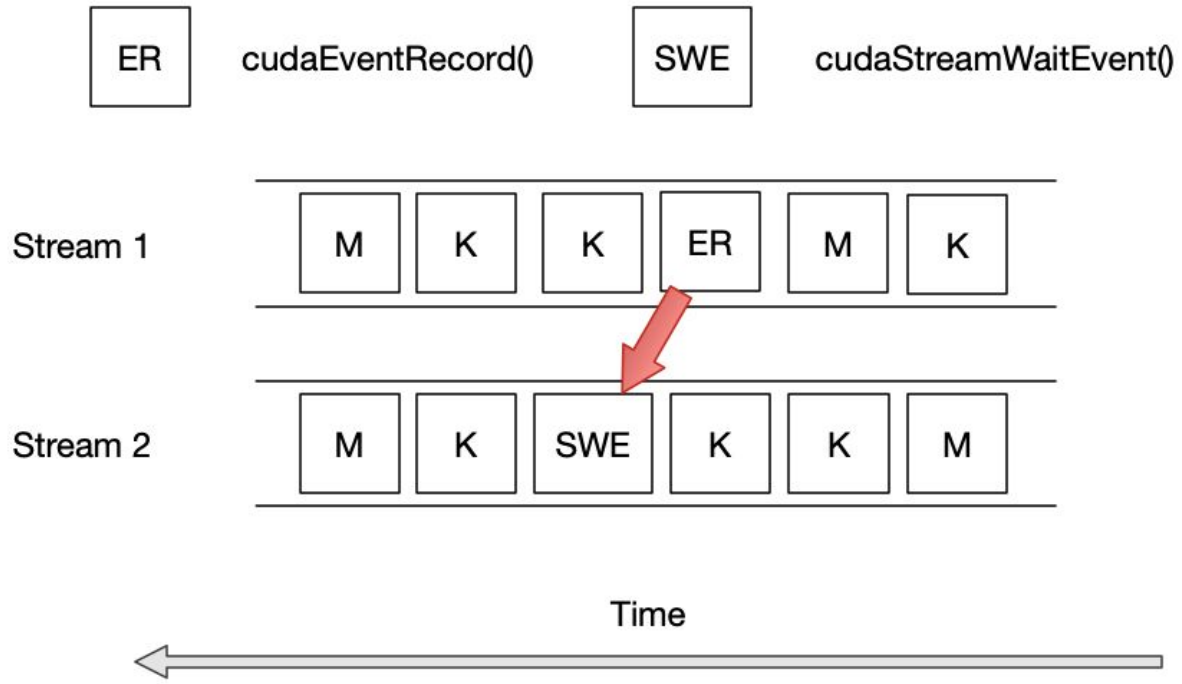

Figura 8: Sincronització entre dos Streams.

Per exemple, a la Figura 8 veiem com un event ha sigut registrat al Stream 1, mitjançat la crida cudaEventRecord(cudaEvent_t event, cudaStream_t stream). Ara, podem utilitzar aquest event, per fer que el Stream 2 esperi que s'acabi d'executar tot el que estava encuat fins a aquell moment a l'Stream1. Per fer això, cridarem cudaStreamWaitEvent(cudaStream_t stream, cudaEvent_t event).

Si volem que sigui un thread de CPU el que s'esperi a l'event, utilitzarem cudaEventSynchronize(cudaEvent_t event).

Multithreading i CUDA.

Existeix la possibilitat d'utilitzar més d'un thread de CPU per executar comandes de CUDA. Per exemple, si volem que dos threads li demanin coses independents entre si a la GPU, o si tenim més d'una GPU, i volem que cada thread executi coses només en una sola GPU.

Aquesta última motivació, ve donada pel fet de que per a que un sol thread de CPU pugui utilitzar una GPU o una altre, cal que el thread faci un canvi de context, i aquest canvi de context de CUDA, te un overhead d'execució. Un altre factor sería que volem que dos threads puguin encuar coses a dues GPU's diferents, en paral·lel, per a que tot s'acabi executant abans.

Les crides de sincronització que hem explicat al apartat anterior, tenen les següents interaccions en un entorn multithread, multi-GPU.

- cudaDeviceSynchronize() : Si un altre thread de CPU encua alguna cosa en algun Stream per a aquesta mateixa GPU, no s'executarà fins que no s'hagi acabat d'executar tot el que ja estava encuat en el moment de cridar cudaDeviceSynchronize().
- cudaStreamSynchronize(cudaStream_t stream) : Si algun altre thread encua algun cosa en altres Streams, es podrà executar. Si ho fan en aquest mateix Stream, no s'executarà fins que no acabi d'executar-se la sincronització.
- cudaEventSynchronize(cudaEvent_t event) : El thread de CPU només s'espera a que hagi acabat la feina fins el punt on està registrat l'event. Qualsevol altre thread de CPU pot encuar coses al mateix o diferent Stream. La feina en altres Streams es podrà executar mentres s'està efectuant aquesta sincronització. Es casi lo mateix que un cudaStreamSincronize, només que aquesta crida és més eficient.

Ecosistema de llibreries.

Com a programadors, és interessant que conegueu l'ecosistema de llibreries de CUDA, per no reinventar la roda.

Igual que per a CPU existeixen llibreries com BLAS i LAPACK, per CUDA existeix cuSOLVER. Per a operacions bàsiques com map, reduce, convolucions etc, existeix la llibreria npp (NVIDIA performance primitives), Thrust, i CUB.

La llista completa de llibreries la trobareu a:
https://developer.nvidia.com/gpu-accelerated-libraries