RÉPUBLIQUE
FRANÇAISE
Liberté
Égalité
Fraternité

ENAC

Programmation CUDA

S. Puechmorel

2023

RÉPUBLIQUE FRANÇAISE
Liberté Égalité Fraternité

ENAC

Plan

L'évolution des cartes graphiques

Années 1980 : contrôleurs video

Ces circuits permettaient d'afficher sur un tube cathodique des informations stockées en mémoire. Ils fournissaient des fonctionnalités de base, essentiellement orientées autour de la gestion de la mémoire vidéo et de la génération des signaux de synchronisation.

Figure: Le contrôleur vidéo MC6845. [1]

[1] https://commons.wikimedia.org/w/index.php?curid=976920

L'évolution des cartes graphiques

Contrôleurs graphiques

Apparus vers la fin des années 1980, ils apportent des fonctionnalités graphiques, telles le tracé de segment, et gèrent une mémoire distincte de celle de l'unité centrale.

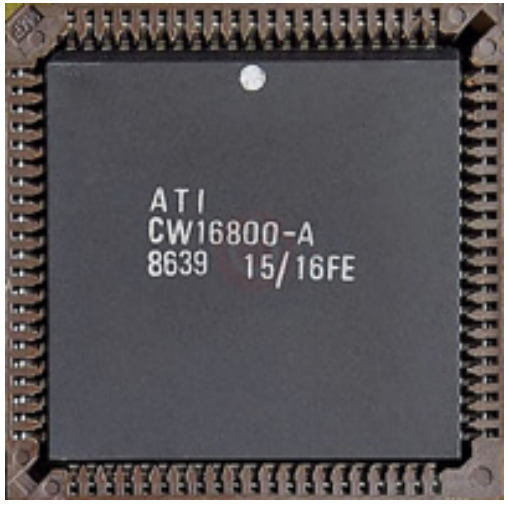

Figure: Contrôleurs graphiques.

RÉPUBLIQUE FRANÇAISE
Liberté Égalité Fraternité

ENAC

L'évolution des cartes graphiques

Les processeurs graphiques 3D.

Disponibles pour le grand public depuis le milieu des années 1990, ils incluent des fonctionnalités d'affichage en trois dimensions. Parallèlement, des bibliothèques logicielles font leur apparition (OpenGL, Direct3D). L'affichage d'une scène est réalisé à travers un pipeline graphique : transformation de sommets, projection, traçage.

Figure: Contrôleur 3D ATI Rage.

L'évolution des cartes graphiques

Les processeurs progammables

En 2001, la société NVIDIA introduit sur le marché la gamme de processeurs graphiques GeForce 3 qui permettent de programmer les étapes du pipeline graphique.

Figure: Contrôleur 3D programmable.

L'évolution des cartes graphiques

Le calcul

Fin 2006, NVIDIA lance la gamme GeForce 8 et l'environnement de développement CUDA qui permet d'exploiter la puissance de calcul des cartes graphiques pour des applications générales. Les performances théoriques sont impressionnantes : de l'ordre de celles obtenues avec un superordinateur, mais pour une enveloppe énergétique bien inférieure.

Figure: Carte graphique CUDA.

RÉPUBLIQUE FRANÇAISE
Liberté Égalité Fraternité

Synthèse de l'évolution des cartes NVIDIA

Année	Carte	Architecture	Cœurs	RAM	Puissance
1995	NV1	Dizaines de µm	?	4 Mo	2 Watts
...	...	...	...	...	...
2017	GTX 1080 Ti	Volta 16 nm	3584	11 Go	257 W
2019	GTX 2080 Ti	Turing 12 nm	4352	11 Go	290 W
2020	RTX 3090	Ampere 8 nm	10 496	24 Go	350 W
2022	RTX 4090	Ada Lovelace 5 nm	18 000	24 Go	450-600 W

Les multiprocesseurs de flux (SM)

- Héritier du pipeline graphique, le multiprocesseur de flux ("Streaming multiprocessor", SM) est une entité de traitement comportant des séquenceurs, plusieurs unités de traitement numérique et une mémoire locale.
- Un processeur graphique (GPU) regroupe plusieurs multiprocesseurs.
- Les multiprocesseurs exécutent des blocs de processus de façon **indépendante** et peuvent accéder à une mémoire partagée.
- Pour un développeur sur une architecture conventionnelle, un multiprocesseur s'apparente à un cœur de calcul vectoriel.

Architecture matérielle

Les multiprocesseurs de flux (SM)

- À l'intérieur d'un multiprocesseur, les processus s'exécutent de façon concurrente, mais peuvent communiquer via la mémoire locale ou être synchronisés.
- Les processus sont regroupés par blocs, appelés "warps" (chaînes), qui se voient affecter le même séquenceur d'instructions.
- Le modèle associé est dit "SIMT" pour "Single Instruction Multiple Thread".

RÉPUBLIQUE
FRANÇAISE
Liberté
Égalité
Fraternité

ENAC

Les unités de calcul

- Le nombre d'unités de calcul par multiprocesseur dépend des générations de cartes. Pour l'architecture Ampère, on trouve 64 ou 128 unités flottantes 32bits, 32 ou 2 unités flottantes 64bits, 64 unités de calcul entier sur 32 bits, 16 unités spéciales (fonctions transcendantes), 4 cœurs de calcul tensoriel et 4 ordonnanceurs.
- Un ordonnanceur est affecté à une chaîne. Tous les processus de la chaîne exécutent la même Instruction au même moment.
- En cas d'instruction conditionnelle, il peut y avoir divergence de code à l'intérieur d'une chaîne, ce qui se traduit par la mise en attente d'un ou plusieurs processus dont l'exécution se poursuivra après celle de la branche principale.

Architecture matérielle

L'ordonnancement des processus

- Le code à exécuter sur le GPU est appelé noyau ("kernel".)
- Le programmeur décide du nombre de processus affectés à un même noyau et les répartit en blocs.
- Un bloc sera pris en charge par un multiprocesseur libre.
- Dans un bloc, des chaînes de 32 processus identifiés par des entiers consécutifs sont constituées.
- Depuis l'architecture Volta, chaque processus possède ses propres compteur de programme et pile d'appel, ce qui permet un contrôle plus fin, en particulier en cas de divergence.

Architecture matérielle

La mémoire

- Chaque multiprocesseur possède une mémoire locale très rapide, pouvant être partagée entre les processus d'un même bloc. Elle est organisée en 32 banques pouvant être utilisées simultanément. Idéalement, chaque processus d'une chaîne accède à sa propre banque. La capacité de cette mémoire varie entre 64kB et 228kB selon les générations.
- Les multiprocesseurs partagent une mémoire globale, plus lente, mais en mesure de stocker beaucoup plus de données. Les cartes de dernière génération, comme la RX4090, embarquent 24Gb de RAM.
- L'architecture 9.0 introduit la notion de cluster de blocs et de mémoire partagée distribuée.

Les mémoires spécialisées

La mémoire de textures

- Les GPUs étant initialement conçus pour des applications de rendu graphique 3D, certaines mémoires dédiées sont présentes.
- La mémoire de textures est particulièrement intéressante lorsque l'on cherche à stocker des données bidimensionnelles que l'on souhaite ensuite interpoler.
- Cette mémoire, chargée par le CPU, ne peut être modifiée par un programme CUDA.

À retenir

Le GPU

- Un bloc est affecté à un multiprocesseur, les processus d'une même chaîne exécutent la même Instruction en parallèle.
- Il faut donc penser avant tout en termes de chaînes (32 processus).
- Un branchement dans un même chaîne entraîne l'inactivation temporaire de processus.

La mémoire

- Privilégier l'utilisation de la mémoire partagée, bien plus rapide que la mémoire globale.
- S'efforcer d'avoir des accès contigus pour les processus d'une même chaîne.
- Penser à utiliser la mémoire des textures si nécessaire.

Compilation

Un dialecte de C/C++

- Le GPU se programme en C/C++ avec des directives spécifiques reconnues par nvcc, le compilateur de NVIDIA.
- Placées devant un nom de variable ou de fonction, elles permettent d'en spécifier l'emplacement.
- Le fichier devant être compilé avec nvcc doit avoir l'extension `.cu`.
- CMake reconnaît CUDA comme un langage à part entière, détecte l'environnement de développement NVIDIA et génère les projets en conséquence.

Directives d'exécution

Avant la déclaration d'une fonction

Directive	Effet
__global__	La fonction est un noyau. Elle est compilée pour le GPU, mais peut être appelée depuis le CPU.
__device__	La fonction est compilée pour le GPU et ne peut être appelée que depuis le GPU.
__host__	La fonction est compilée pour le CPU et ne peut être appelée que depuis le CPU (comportement par défaut).

Directives de stockage

Avant la déclaration d'une variable

Directive	Effet
__device__	La variable est stockée dans la mémoire globale.
__constant__	La variable est stockée dans la mémoire globale des constantes.
__shared__	La variable est stockée dans la mémoire partagée d'un multiprocesseur.

Directives d'exécution

Lancement d'un noyau

- Une fonction `fun` déclarée avec la directive `__global__` est éligible à une exécution parallèle sur le GPU.
- La directive de lancement prend la forme suivante:
  `<<<gridDim,blockDim,sharedMen=0,stream=NULL>>>fun(args)`
- `gridDim` est une structure de 3 entiers, $gridDim.x$, $gridDim.y$, $gridDim.z$ déterminant la taille de la grille de blocs.
- `blocDim` est une structure de 3 entiers, $blocDim.x$, $blocDim.y$, $blocDim.z$ déterminant la taille d'un bloc de threads.
- `sharedMem` est la taille totale en octets de mémoire partagée allouée dynamiquement.
- `stream` est un pointeur vers un objet de type `stream` qui permet de gérer le parallélisme.

Dimensions

Dimensions de bloc

- Un bloc de processus est affecté à un multiprocesseur et ne peut pas excéder une certaine valeur.
- Pour des raisons de flexibilité lors du codage, une tel bloc peut être organisé en un tableau à une, deux ou trois dimensions.
- La première coordonnée est particulière : elle peut recevoir l'intégralité du bloc.

Dimensions

Dimensions de grille

- Les processus lancés sont d'abord organisés en blocs, puis les blocs en grille.
- Tout comme précédemment, les grilles possèdent trois dimensions, la première pouvant adresser tous les blocs.

Dimensions

Obtention des valeurs maximales

- La fonction `cudaDeviceGetAttribute` permet de connaître les valeurs maximales pour les dimensions de grilles et de blocs.
- Sa signature est: `__host____device__ cudaError_t cudaDeviceGetAttribute ( int* value, cudaDeviceAttr attr, int device )`
- De nombreuses caractéristiques du GPU référencé par l'attibut `device` peuvent être obtenues.
- Pour les dimensions, `attr` prendra les valeurs: `cudaDevAttrMaxBlockDimX,...,cudaDevAttrMaxGridDimX,...`

RÉPUBLIQUE FRANÇAISE
Liberté Égalité Fraternité

ENAC

Exemple de programme

Le produit matriciel sur CPU

```
void host_matmul(int lda, int ncol, float* a,
    int ldb, float* b, float* res) {
    double s;

    for (int i = 0 ; i < lda; i++) {
        for (int j = 0 ; j < k ; j++) {
            s = 0.0;
            for (int k = 0; k < ldb; k++)
                s += a[i * lda + k] * b[k * ldb + j];
        }
        c[i * lda + j] = s;
    }
}
```

Exemple de programme

Passage sur GPU

- On remarque que l'écriture dans C peut être asynchrone.
- Les deux boucles de niveau supérieur sont remplacées par des appels parallèles.
- La boucle interne est exécutée par chaque processus.
- Un bloc reçoit une sous-matrice de C à calculer.
- Pour une écriture plus simple des calculs, on choisira d'organiser la grille et les blocs en deux dimensions.

Exemple de programme

Noyau de calcul

```
6  __global__ void matmul(int lda, int ncola, float* a, int ncolb, float* b, float* c) {
7      // get indices in the matrix
8      int i = threadIdx.x + blockIdx.x * blockDim.x;
9      int j = threadIdx.y + blockIdx.y * blockDim.y;
10     float s;
11     if (i < lda && j < ncolb) { // check validity of thread
12         s = 0.0f;
13         for (int k = 0; k < ncola; k++) // accumulate products
14             s += a[i * ncola + k] * b[k * ncolb + j];
15         c[i * ncolb + j] = s;
16     }
17 }
```

- L'élément à calculer est obtenu à partir des coordonnées de bloc (`blockIdx`), puis de processus (`threadIdx`).
- Seule la boucle interne est conservée.
- Chaque processus opère sur une ligne et une colonne de la matrice

Exemple de programme

Exécution

```
28 void device_matmul(int lda, int ncola, float* a, int ncolb, float* b, float* c) {
29     int nbx, nby;
30     // compute required numvber of blocs in each direction
31     nbx = (lda + BLOCK_DIM - 1) / BLOCK_DIM;
32     nby = (ncolb + BLOCK_DIM - 1) / BLOCK_DIM;
33     // allocate device memory
34     float* da, * db, * dc;
35     cudaMalloc(&da, lda * ncola * sizeof(float));
36     cudaMalloc(&db, ncolb * ncola * sizeof(float));
37     cudaMalloc(&dc, lda * ncolb * sizeof(float));
38     cudaMemcpy(da, a, lda * ncola * sizeof(float), cudaMemcpyHostToDevice);
39     cudaMemcpy(db, b, ncolb * ncola * sizeof(float), cudaMemcpyHostToDevice);
40     matmul<<< dim3(nbx, nby, 1), dim3(BLOCK_DIM, BLOCK_DIM, 1) >>> (lda, ncola, da, ncolb, db, dc);
41     cudaDeviceSynchronize();
42     cudaMemcpy(c, dc, lda * ncolb * sizeof(float), cudaMemcpyDeviceToHost);
43     cudaFree(da);
44     cudaFree(db);
45     cudaFree(dc);
46 }
```

- La fonction `cudaMalloc` permet d'allouer de la mémoire sur le GPU.
- Elle est libérée par `cudaFree`.
- Les données sont transférées par `cudaMemcpy`.

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

ENAC

Exemple de programme

Vitesse d'exécution

- Deux matrices 1000x1000 sont multipliées.
- Sur la configuration de référence, on relève, pour le GPU, une performance de 55 GFlops.
- Pour le CPU, elle est de 1 Gflops.
- Peut-on améliorer la vitesse de calcul ?

Produit matriciel simple

Accès mémoire

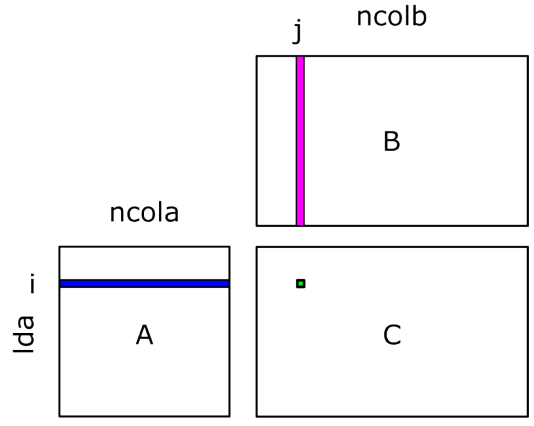

Figure: Accès linéaire.

- Seule la mémoire globale est utilisée.
- Tous les processus de même numéro de ligne (resp. colonne) accèdent aux mêmes données dans A (resp. B).

Produit matriciel par blocs

améliorer l'utilisation des données

- Le produit de deux matrices peut s'effectuer par blocs:

$$\begin{pmatrix} A_{1,1} & A_{1,2} & \dots & A_{1,n} \\ \vdots & \vdots & \vdots & \vdots \\ A_{i,1} & A_{i,2} & \dots & A_{i,n} \\ \vdots & \vdots & \vdots & \vdots \\ A_{m,1} & A_{m,2} & \dots & A_{m,n} \end{pmatrix} \begin{pmatrix} B_{1,1} & \dots & B_{1,j} & \dots & B_{1,p} \\ \vdots & \vdots & \vdots & \vdots & \\ \vdots & \vdots & \vdots & \vdots & \\ B_{n,1} & B_{n,2} & B_{n,j} & \dots & B_{n,p} \end{pmatrix}$$

$$= \begin{pmatrix} C_{1,1} & \dots & C_{1,p} \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \\ C_{m,1} & \dots & C_{m,p} \end{pmatrix}, \quad C_{i,j} = \sum_{k=1} A_{i,k} B_{k,j} \tag{1}$$

Produit matriciel par blocs

Accès mémoire

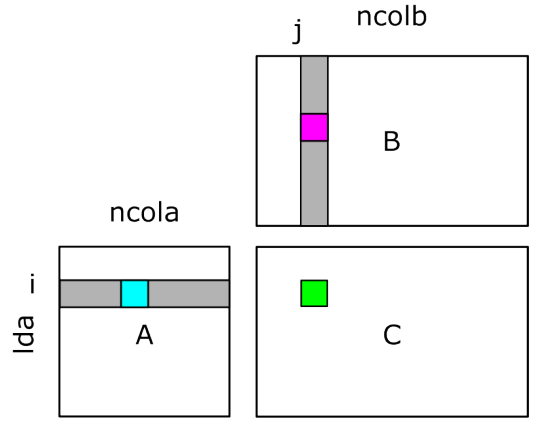

Figure: Accès par blocs.

- Les données d'un même bloc peuvent résider en mémoire partagée.
- Tous les processus d'une même chaîne peuvent effectuer un chargement simultané.

Produit matriciel par blocs

Codage

```cpp
__global__ void block_matmul(int lda, int ncola, float *a, int ncolb, float *b, float *c)
{
    // indice de début des blocs A , B et C
    int ia = blockIdx.x * ncola * BLOCK_DIM;
    int ib = blockIdx.y * BLOCK_DIM;
    int ic = blockIdx.x * ncolb * BLOCK_DIM + blockIdx.y * BLOCK_DIM;
    // pas en A et B
    int sa = BLOCK_DIM;
    int sb = ncolb * BLOCK_DIM;
    __shared__ float blockA[BLOCK_DIM][BLOCK_DIM];
    __shared__ float blockB[BLOCK_DIM][BLOCK_DIM];
    int i = blockIdx.x * BLOCK_DIM + threadIdx.x;
    int j = blockIdx.y * BLOCK_DIM + threadIdx.y;
    float s = 0.0f;
    for (int k = 0; k < ncola; k += BLOCK_DIM, ia += sa, ib += sb)
    {
        int rem = min(BLOCK_DIM, ncola - k);
        // chargement d'un élément en mémoire partagée
        if (i < lda && threadIdx.y < rem)
            blockA[threadIdx.x][threadIdx.y] = a[ia + threadIdx.y + threadIdx.x * ncola];
        else
            blockA[threadIdx.x][threadIdx.y] = 0.0f;
        if (j < ncolb && threadIdx.x < rem)      You, 48 seconds ago • Uncommitted changes
            blockB[threadIdx.x][threadIdx.y] = b[ib + threadIdx.y + threadIdx.x * ncolb];
        else
            blockB[threadIdx.x][threadIdx.y] = 0.0f;
        __syncthreads(); // point de synchronisation pour s'assurer du chargement complet des blocs
        // calcul du produit matriciel
        for (int l = 0; l < rem; l++)
            s += blockA[threadIdx.x][l] * blockB[l][threadIdx.y];
        __syncthreads(); // point de synchronisation pour les calculs
    }
    if ((i < lda) && (j < ncolb))
        c[ic + threadIdx.y + threadIdx.x * ncolb] = s;
}
```

Figure: Accès par blocs.

- Chaque processus charge un élément en mémoire partagée.
- Les éléments invalides sont mis à 0.
- Les blocs externes sont traités spécialement.

Produit matriciel par blocs

Performances

- Avec une dimension de bloc de 8 (soit 64 processus), on atteint 350 Gflops sur des matrices de taille 10000.
- En comparaison, le CPU ne dépasse pas 2 GFlops.
- Il est possible d'optimiser encore le code, en particulier en améliorant la gestion des blocs situés en périphérie de la grille.
- Le produit matriciel est toutefois un exemple simple, le code GPU étant assez similaire à son homologue CPU.

RÉPUBLIQUE FRANÇAISE
Liberté Égalité Fraternité

ENAC

Flux de synchronisation

Opérations asynchrones

- Par défaut, les exécutions de noyaux sur le GPU sont indépendantes du déroulement du programme sur CPU, sauf pour les transferts mémoire ou les lancements de grilles de processus.
- Dans de nombreux cas, il pourrait être bénéfique de s'affranchir de cette limitation.
- Les flux d'exécution ont été introduits pour permettre à des noyaux différents ou à des opérations de transfert entre hôte et GPU de se dérouler concurremment.

Flux de synchronisation

Transferts mémoire asynchrones

- Un flux d'exécution est créé par un appel à la fonction `cudaStreamCreate(cudaStream_t *stream)`
- Il est supprimé en appelant `cudaStreamDestroy(cudaStream_t stream)`
- Un transfert mémoire associé à un flux est réalisé à l'aide de la fonction `cudaMemcpyAsync()`.
- Il est indépendant de l'ordre d'appel sur l'hôte, mais est synchronisé sur le flux.
- Les transferts de mémoire asynchrones ne peuvent s'effectuer qu'entre des blocs spécifiquement alloués sur l'hôte par `cudaMallocHost(void **ptr, size_t size)`

Flux de synchronisation

Grilles de calcul asynchrones

- Le dernier paramètre lors du lancement d'une grille de calcul est un flux d'exécution.
- Pour exécuter un noyau `kernel` dans le flux (stream) on écrira:
  `kernel<<<grid,block,dynSharedSize,stream>>>(args)`.
- Les opérations d'un flux se synchronisent sur le flux par défaut, qui est utilisé lorsque le paramètre de flux est absent. Ce comportement peut toutefois être changé.

Flux de synchronisation

Un exemple: le produit scalaire

- Le produit scalaire entre deux vecteurs peut être efficacement implémenté sur le GPU.
- Dans cet exemple, on choisit de réaliser un calcul par blocs, mais d'effectuer la réduction sur le CPU.
- On calcule des produits scalaires de façon asynchrone, dans des processus (CPU) différents.

Noyau pour le produit scalaire

Code

```
__global__ void dot_product(float *a, float *b, float *c, int  n) {
    __shared__ float r[BLOCK_DIM];
    int i = blockIdx.x * BLOCK_DIM + threadIdx.x;
    int step = BLOCK_DIM >> 1;
    // chargement des données
    if(i < n) {
        r[threadIdx.x] = a[i] * b[i];
    } else
        r[threadIdx.x] = 0.0f;
    __syncthreads();
    // calcul de la somme
    while(step >= 1) {
        if(threadIdx.x < step)
            r[threadIdx.x] += r[threadIdx.x + step];
        __syncthreads();
        step >>= 1;
    }
    c[blockIdx.x] = r[0];
}
```

Figure: Produits partiels sur GPU

- La première partie du calcul est classique.
- La réduction s'effectue par sommation d'éléments deux à deux.

Lancement asynchrone

Code

```
void dot_async(int n, int ntasks) {
    float *a, *b;
    std::future<float> *dot = new std::future<float>[ntasks];
    cudaStream_t *streams = new cudaStream_t[ntasks];
    a = new float[n];
    b = new float[n];
    for(int i = 0 ; i < n ; i++) {
        a[i] = 1.0f/(float)(i+1);
        b[i] = (float)(i+1);
    }

    float r = 0.0f;
    for(int i = 0 ; i < n ; i++)
            r += a[i] * b[i];
    for(int tsk = 0 ; tsk < ntasks ; tsk++) {
        cudaStreamCreate(streams+tsk);
        dot[tsk] = std::async(std::launch::async, device_dot, a, b, n, streams[tsk]);
    }
    cudaDeviceSynchronize();
    for(int tsk = 0 ; tsk < ntasks ; tsk++) {
        dot[tsk].wait();
        cudaStreamDestroy(streams[tsk]);
        std::cout  << tsk << '\t' << r << '\t' << dot[tsk].get() <<std::endl;
    }

    delete[] streams;
    delete[] dot;
    delete[] a;
    delete[] b;
}
```

Figure: API asynchrone.

- Les transferts en mémoire et les appels de grille sont attachés à un flux.

Processus de calcul

Code

```
void dot_async(int n, int ntasks) {
    float *a, *b;
    std::future<float> *dot = new std::future<float>[ntasks];
    cudaStream_t *streams = new cudaStream_t[ntasks];
    a = new float[n];
    b = new float[n];
    for(int i = 0 ; i < n ; i++) {
        a[i] = 1.0f/(float)(i+1);
        b[i] = (float)(i+1);
    }

    float r = 0.0f;
    for(int i = 0 ; i < n ; i++)
            r += a[i] * b[i];
    for(int tsk = 0 ; tsk < ntasks ; tsk++) {
        cudaStreamCreate(streams+tsk);
        dot[tsk] = std::async(std::launch::async, device_dot, a, b, n, streams[tsk]);
    }
    cudaDeviceSynchronize();
    for(int tsk = 0 ; tsk < ntasks ; tsk++) {
        dot[tsk].wait();
        cudaStreamDestroy(streams[tsk]);
        std::cout  << tsk << '\t' << r << '\t' << dot[tsk].get() <<std::endl;
    }

    delete[] streams;
    delete[] dot;
    delete[] a;
    delete[] b;
}
```

Figure: Exécution asynchrone sur le CPU.

- Chaque grille de calcul est appelée dans un processus CPU différent.

Parallélisme dynamique

Grilles filles

- Une grille de calcul peut en exécuter une autre.
- Un seul processus est responsable du lancement de la grille fille.
- Une grille mère ne peut se terminer avant ses enfants.
- En dehors du flux par défaut, on peut utiliser les flux prédéfinis suivants :
  - `cudaStreamFireAndForget`. La grille fille n'est pas synchronisée à sa mère (sauf en fin d'exécution), ni à d'autres enfants.
  - `cudaStreamTailLaunch`. La grille fille est lancée à la fin de l'exécution de la mère. Les enfants se synchronisent entre eux dans l'ordre d'appel.

Parallélisme dynamique

Réduction parallèle

```
// b est un buffer en memoire globale
__global__ void reduction(float *a, float *b, int n) {
    __shared__ float r[BLOCK_DIM];
    int i = blockIdx.x * BLOCK_DIM + threadIdx.x;
    int nb = (n + BLOCK_DIM -1) / BLOCK_DIM;
    int step = BLOCK_DIM >> 1;
    if (i < n)
        r[threadIdx.x] = a[i];
    else
        r[threadIdx.x] = 0.0f;
    __syncthreads();
    while(step >= 1) {
        if(threadIdx.x < step)
          r[threadIdx.x] += r[threadIdx.x + step];
        __syncthreads();
        step >>= 1;
    }
    if(threadIdx.x == 0)
        b[blockIdx.x] = r[0];
    __syncthreads();
    if(i == 0 && nb > 1)
            reduction<<<nb, BLOCK_DIM,0,cudaStreamTailLaunch>>>(b,b,nb);
}
```

Figure: Création de grilles filles.

- Le flux `cudaStreamTailLaunch` garantit l'ordre d'exécution.
- Il faut des options de compilation spéciales (code relogeable).

Accès mémoire unifié

Un même pointeur CPU/GPU

- La mémoire unifiée est allouée grâce à la fonction `cudaMallocManaged`.
- Elle est libérée en appelant `cudaFree`
- Le pointeur obtenu est utilisable sur l'hôte **et** sur le GPU.
- Cela permet d'éliminer la plupart du temps les appels à `cudaMemcpy`.
- Le décorateur `__managed__` peut également être utilisé devant un nom de variable globale.

Mémoire unifiée

Noyau

```
#define BLOCK_DIM (256)
#define SIZE (10000)

__device__ __managed__ float c[SIZE];

__global__ void add(float *a, float *b, int n) {
    int i = blockIdx.x * BLOCK_DIM + threadIdx.x;
    if(i < n)
        c[i] = a[i] + b[i];
}
```

Figure: Utilisation de la mémoire unifiée.

- On notera la déclaration de la variable globale de retour.

Mémoire unifiée

Programme principal

```
int main(int argc, char *argv[]) {
    float *a,*b;
    int ns  = (SIZE + BLOCK_DIM -1) / BLOCK_DIM;
    cudaMallocManaged(&a, SIZE * sizeof(float));
    cudaMallocManaged(&b, SIZE * sizeof(float));
    for(int i = 0 ; i < SIZE ; i++) {
        a[i] = 1.0f;
        b[i] = (float)i;
    }
    add<<<ns, BLOCK_DIM>>>(a,b,SIZE);
    cudaDeviceSynchronize();
    float err = 0.0f;
    for(int i = 0 ; i < SIZE ; i++) {
        err += fabsf(b[i]+1.0f-c[i]);
    }
    std::cout << "Erreur : " << err << std::endl;
    cudaFree(b);
    cudaFree(a);
}
```

Figure: Allocation sur CPU.

- L'utilisation de la mémoire unifiée permet de se dispenser d'opérations de copie.
- Le décorateur `__managed__` déclare la variable sur le CPU et le GPU.