Exemples CUDA

Patrick Martineau

Version 1.0, 2018-03-30

1. Les bases

L'environnement de développement pour CUDA doit être installé sur votre ordinateur, équipé d'une carte NVIDIA.

1.1. Vérification de la compilation par votre compilateur, exemple avec nvcc

Lien sur hello_world.cu.

```
/*
 * Copyright 1993-2010 NVIDIA Corporation.  All rights reserved.
 */
#include "./book.h"
int main( void ) {
    printf( "Hello, World!\n" );
    return 0;
}
```

```
$ nvcc hello_world.cu -o hello_world
$ ./hello_world
```

Ce programme ne fait pas appel au GPU mais uniquement au compilateur nvcc sur du code C classique. Comme nvcc s'appuye sur gcc, il n'y a pas de problème pour compiler du C.

1.2. Première utilisation du GPU

L'exemple suivant permet d'introduire la création d'un kernel. Un kernel est compilé dans le code assembleur correspond au GPU cible. Là, nvcc réalise donc un travail spécifique. Ensuite, le noyau est lancé une fois sur un coeur du GPU.

Lien sur simple_kernel.cu.

```
#include "./book.h"
__global__ void kernel( void ) {
}
int main( void ) {
    kernel<<<1,1>>>();
    printf( "Hello, World!\n" );
    return 0;
}
```

L'étape suivante consiste à ajouter le passage de paramètre de l'hôte au kernel et de récupérer le résultat du calcul réalisé par ce kernel. Ainsi, on voit apparaitre clairement 2 des 3 étapes

indispensables lors de l'utilisation d'un GPU.

- L'allocation mémoire au niveau du GPU
- Le démarrage du Kernel

Lien sur simple_kernel_params.cu.

```
#include "./book.h"
__global__ void add( int a, int b, int *c ) {
    *c = a + b;
}
int main( void ) {
    int c;
    int *dev_c;
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, sizeof(int) ) );
    add<<<1,1>>>( 2, 7, dev_c );
    HANDLE_ERROR( cudaMemcpy( &c, dev_c, sizeof(int),
                              cudaMemcpyDeviceToHost ) );
    printf( "2 + 7 = %d\n", c );
    HANDLE_ERROR( cudaFree( dev_c ) );
    return 0;
}
```

L'exemple suivant illustre l'appel d'une fonction à l'intérieur d'un noyau. On utilise alors *device*.

Lien sur simple_device_call.cu.

```
#include "./book.h"
__device__ int addem( int a, int b ) {
    return a + b;
}
__global__ void add( int a, int b, int *c ) {
    *c = addem( a, b );
}
int main( void ) {
    int c;
    int *dev_c;
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, sizeof(int) ) );
    add<<<1,1>>>( 2, 7, dev_c );
    HANDLE_ERROR( cudaMemcpy( &c, dev_c, sizeof(int),
                              cudaMemcpyDeviceToHost ) );
    printf( "2 + 7 = %d\n", c );
    HANDLE_ERROR( cudaFree( dev_c ) );
    return 0;
}
```

2. Première utilisation du parallélisme

Prenons un exemple simple : l'addition de 2 vecteurs. Le code C suivant se décompose en trois parties :

- L'initialisation des vecteurs a et b,
- Le calcul de la somme
- l'affichage du résultat

Ce qui nous intéresse est de paralléliser le calcul de la somme. Le GPU a vocation à être utilisé pour du massivement parallèle. Dans l'exemple, la taille des vecteurs est limité à 10, on souhaite donc faire les 10 additions en parallèle… La somme des vecteurs est donc isolée dans la fonction add(). Dans la version CPU, celui-ci exécute une boucle de 1 à N pour réaliser successivement l'addition pour chaque composante.

Lien sur add_loop_cpu.cu.

```
#include <stdio.h>
#define N   10
void add( int *a, int *b, int *c ) {
    int tid = 0;    // this is CPU zero, so we start at zero
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 1;   // we have one CPU, so we increment by one
    }
}
int main( void ) {
    int a[N], b[N], c[N];
    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }
    add( a, b, c );
    // display the results
    for (int i=0; i<N; i++) {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }
    return 0;
}
```

La version parallèle avec le GPU consiste donc à réduire la fonction add à la somme sur une seule composante. En plus du code 'CPU' on retrouve donc :

- l'allocation initiale de la mémoire GPU pour les 3 vecteurs a, b et c
- l'initialisation de a et b par recopie des valeurs depuis le cpu
- le démarrage de 10 threads, chacun effectuant 1 addition, dans 10 blocs différents

- la recopie du vecteur c depuis la mémoire du GPU
- la libération de l'espace mémoire utilisée dans le GPU.

Lien sur add_loop_gpu.cu.

```
#include <stdio.h>
#define N   10
__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x;    // this thread handles the data at its thread id
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
int main( void ) {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;
    // allocate the memory on the GPU
    cudaMalloc( (void**)&dev_a, N * sizeof(int) );
    cudaMalloc( (void**)&dev_b, N * sizeof(int) );
    cudaMalloc( (void**)&dev_c, N * sizeof(int) );
    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }
    // copy the arrays 'a' and 'b' to the GPU
    cudaMemcpy( dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice );
    add<<<N,1>>>( dev_a, dev_b, dev_c );
    // copy the array 'c' back from the GPU to the CPU
    cudaMemcpy( c, dev_c, N * sizeof(int),
                              cudaMemcpyDeviceToHost );
    // display the results
    for (int i=0; i<N; i++) {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }
    // free the memory allocated on the GPU
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
    return 0;
}
```

Remarque : Un point important est de bien comprendre que les 2 uunités de traitement, CPU et GPU, vont devoir collaborer pour se répartir le travail. Cette répartition est à l'initiative du CPU. Il faut aussi gérer les deux espaces mémoires et quand le GPU exécute des threads, l'espace mémoire utilisé est celui du GPU car il ne peut accéder directement à la mémoire du CPU.

Il est bon de prendre dès le début les bonnes habitudes : distinguer le nom des variables du CPU du nom des variables du GPU !

3. Mise en oeuvre efficace

L’intérêt du GPU apparait quand on utilise un grand nombre de threads en parallèle. On reprend donc l’exemple simple précédent, l’addition de deux vecteurs, et on répartit le calcul sur 128 threads. (On utilise toujours des puissances de 2) La fonction add est donc modifiée de manière à répartir les calculs des différentes composantes sur les threads. Chaque thread a un numéro différent des autres, et "saute" d’une composante à l’autre en ajoutant le nombre de threads.

Lien sur add_loop_long.cu.

```
#include <stdio.h>
#define N   (32 * 1024)
__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += gridDim.x;
    }
}
int main( void ) {
    int *a, *b, *c;
    int *dev_a, *dev_b, *dev_c;
    // allocate the memory on the CPU
    a = (int*)malloc( N * sizeof(int) );
    b = (int*)malloc( N * sizeof(int) );
    c = (int*)malloc( N * sizeof(int) );
    // allocate the memory on the GPU
    cudaMalloc( (void**)&dev_a, N * sizeof(int) );
    cudaMalloc( (void**)&dev_b, N * sizeof(int) );
    cudaMalloc( (void**)&dev_c, N * sizeof(int) );
    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = i;
        b[i] = 2 * i;
    }
    // copy the arrays 'a' and 'b' to the GPU
    cudaMemcpy( dev_a, a, N * sizeof(int),cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, b, N * sizeof(int),cudaMemcpyHostToDevice );
    add<<<128,1>>>( dev_a, dev_b, dev_c );
    // copy the array 'c' back from the GPU to the CPU
    cudaMemcpy( c, dev_c, N * sizeof(int),cudaMemcpyDeviceToHost );
    // verify that the GPU did the work we requested
    bool success = true;
    for (int i=0; i<N; i++) {
        if ((a[i] + b[i]) != c[i]) {
            printf( "Error:  %d + %d != %d\n", a[i], b[i], c[i] );
            success = false;
        }
    }
    if (success)    printf( "We did it!\n" );
    // free the memory we allocated on the GPU
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
    // free the memory we allocated on the CPU
    free( a );
    free( b );
    free( c );
    return 0;
}
```

Dans cet exemple, on retrouve les 5 étapes comme dans l'exemple précédent. On remarque que la dernière partie n'est qu'une vérification car elle consiste à recalculer la somme des composantes sur le CPU et à comparer le résultat avec le calcul GPU.

L'exemple suivant reprend le même principe mais en utilisant des threads en parallèle au sein d'un block (et non des blocks en parallèle avec chacun 1 thread).

Lien sur add_loop_blocks.cu.

```
#include <stdio.h>
#define N   10
__global__ void add( int *a, int *b, int *c ) {
    int tid = threadIdx.x;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
int main( void ) {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;
    // allocate the memory on the GPU
    cudaMalloc( (void**)&dev_a, N * sizeof(int) ) ;
    cudaMalloc( (void**)&dev_b, N * sizeof(int) );
    cudaMalloc( (void**)&dev_c, N * sizeof(int) );
    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = i;
        b[i] = 2 * i;
    }
    // copy the arrays 'a' and 'b' to the GPU
    cudaMemcpy( dev_a, a, N * sizeof(int),
                              cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, b, N * sizeof(int),
                              cudaMemcpyHostToDevice );
    add<<<1,N>>>( dev_a, dev_b, dev_c );
    // copy the array 'c' back from the GPU to the CPU
    cudaMemcpy( c, dev_c, N * sizeof(int),
                              cudaMemcpyDeviceToHost );
    // display the results
    for (int i=0; i<N; i++) {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }
    // free the memory allocated on the GPU
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
    return 0;
}
```

La meilleure solution est évidemment la dernière qui utilise un grand nombre de threads répartis entre plusieurs blocks contenant chacun plusieurs threads.

Lien sur add_loop_long_blocks.cu.

```
#include <stdio.h>
#define N   (33 * 1024)
__global__ void add( int *a, int *b, int *c ) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += blockDim.x * gridDim.x;
    }
}
int main( void ) {
    int *a, *b, *c;
    int *dev_a, *dev_b, *dev_c;
    // allocate the memory on the CPU
    a = (int*)malloc( N * sizeof(int) );
    b = (int*)malloc( N * sizeof(int) );
    c = (int*)malloc( N * sizeof(int) );
    // allocate the memory on the GPU
    cudaMalloc( (void**)&dev_a, N * sizeof(int) );
    cudaMalloc( (void**)&dev_b, N * sizeof(int) );
    cudaMalloc( (void**)&dev_c, N * sizeof(int) );
    // fill the arrays a and b on the CPU
    for (int i=0; i<N; i++) {
        a[i] = i;
        b[i] = 2 * i;
    }
    // copy the arrays a and b to the GPU
    cudaMemcpy( dev_a, a, N * sizeof(int),cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, b, N * sizeof(int),cudaMemcpyHostToDevice );
    add<<<128,128>>>( dev_a, dev_b, dev_c );
    // copy the array c back from the GPU to the CPU
    cudaMemcpy( c, dev_c, N * sizeof(int),cudaMemcpyDeviceToHost );
    // verify that the GPU did the work we requested
    bool success = true;
    for (int i=0; i<N; i++) {
        if ((a[i] + b[i]) != c[i]) {
            printf( "Error:  %d + %d != %d\n", a[i], b[i], c[i] );
            success = false;
        }
    }
    if (success)    printf( "We did it!\n" );
    // free the memory we allocated on the GPU
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
    // free the memory we allocated on the CPU
    free( a );
    free( b );
    free( c );
    return 0;
}
```

4. Utilisation adaptée des différentes zones mémoires

4.1. Mémoire partagée

La mémoire partagée est accessible par tous les threads au sein du même block. La mémoire est accessible en lecture et écriture mais elle ne peut être adresssée depuis le CPU. C'est donc au sein du block que les threads doivent initialiser la mémoire partagée et, en fin de calcul, recopier le résultat dans la zone de mémoire globale du GPU. A partir de cette mémoire globale, le CPU pourra récupérer ce résultat final et le recopier dans la mémoire du CPU.

4.1.1. Calcul réparti mais résultat global

L'exemple choisi pour mettre en évidence le travail collaboratif des threads, et le niveau de collaboration à mettre en oeuvre sur la base de l'utilisation adaptée des différentes zones mémoires est le calcul d'un histogramme. On prend un texte en entrée et on veut calculer le nombre d'occurences de chaque lettre.

La version CPU parcourt donc séquentiellement l'ensemble du texte et, pour chaque lettre identifiée, incrémente son compteur dans un tableau "histo".

Lien sur hist_cpu.cu.

```
#include <stdio.h>
#define SIZE    (100*1024*1024)
int main( void ) {
    unsigned char *buffer =
                     (unsigned char*)big_random_block( SIZE );
    // capture the start time
    clock_t         start, stop;
    start = clock();
    unsigned int    histo[256];
    for (int i=0; i<256; i++)
        histo[i] = 0;
    for (int i=0; i<SIZE; i++)
        histo[buffer[i]]++;
    stop = clock();
    float   elapsedTime = (float)(stop - start) /
                          (float)CLOCKS_PER_SEC * 1000.0f;
    printf( "Time to generate:  %3.1f ms\n", elapsedTime );
    long histoCount = 0;
    for (int i=0; i<256; i++) {
        histoCount += histo[i];
    }
    printf( "Histogram Sum:  %ld\n", histoCount );
    free( buffer );
    return 0;
}
```

La version initiale adaptée pour le GPU consiste à identifier le noyau à paralléliser. On identifie rapidement que le texte en entrée peut être découpé en morceaux traités indépendamment par chaque thread. La mise en oeuvre est réalisée par "saut" ce qui permet de s'adapter implicitement à la taille de l'entrée et au nombre de threads. Par contre, comme le résultat est global, il faut que les compteurs "histo" soient partagés. Pour obtenir un résultat cohérent, on assure l'accès en exclusion mutuelle à "histo".

Lien sur hist_gpu_gmem_atomics.cu.

```
#include <sdio.h>
#define SIZE    (100*1024*1024)
__global__ void histo_kernel( unsigned char *buffer,
                              long size,
                              unsigned int *histo ) {
    // calculate the starting index and the offset to the next
    // block that each thread will be processing
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;
    while (i < size) {
        atomicAdd( &histo[buffer[i]], 1 );
        i += stride;
    }
}
```

```
int main( void ) {
    unsigned char *buffer =
                    (unsigned char*)big_random_block( SIZE );
    // capture the start time
    // starting the timer here so that we include the cost of
    // all of the operations on the GPU.
    cudaEvent_t     start, stop;
    cudaEventCreate( &start );
    cudaEventCreate( &stop );
    cudaEventRecord( start, 0 );
    // allocate memory on the GPU for the file's data
    unsigned char *dev_buffer;
    unsigned int *dev_histo;
    cudaMalloc( (void**)&dev_buffer, SIZE );
    cudaMemcpy( dev_buffer, buffer, SIZE, cudaMemcpyHostToDevice );
    cudaMalloc( (void**)&dev_histo,
                            256 * sizeof( int ) );
    cudaMemset( dev_histo, 0,
                            256 * sizeof( int ) );
    // kernel launch - 2x the number of mps gave best timing
    cudaDeviceProp  prop;
    cudaGetDeviceProperties( &prop, 0 );
    int blocks = prop.multiProcessorCount;
    histo_kernel<<<blocks*2,256>>>( dev_buffer, SIZE, dev_histo );
    unsigned int    histo[256];
    cudaMemcpy( histo, dev_histo,
                            256 * sizeof( int ),
                            cudaMemcpyDeviceToHost );
    // get stop time, and display the timing results
    cudaEventRecord( stop, 0 );
    cudaEventSynchronize( stop );
    float   elapsedTime;
    cudaEventElapsedTime( &elapsedTime,
                                      start, stop );
    printf( "Time to generate:  %3.1f ms\n", elapsedTime );
    long histoCount = 0;
    for (int i=0; i<256; i++) {
        histoCount += histo[i];
    }
    printf( "Histogram Sum:  %ld\n", histoCount );
    // verify that we have the same counts via CPU
    for (int i=0; i<SIZE; i++)
        histo[buffer[i]]--;
    for (int i=0; i<256; i++) {
        if (histo[i] != 0)
            printf( "Failure at %d!  Off by %d\n", i, histo[i] );
    }
    cudaEventDestroy( start );
    cudaEventDestroy( stop );
    cudaFree( dev_histo );
    cudaFree( dev_buffer );
```

```
    free( buffer );
    return 0;
}
```

Cette mise en oeuvre montre des limites dues aux performances d'accès à la mémoire globale lorsque de nombreux threads veulent y accéder simultanément. En effet, même si tous les threads n'accèdent pas simultanément à la même case mémoire, tous utilisent le même bus interne à la carte GPU.

Pour accélérer le fonctionnement de cette mise en oeuvre parallèle, il faut permettre aux threads de ne pas être **trop** ralentis par l'accès à la mémoire. On peut envisager que chaque thread ait un tableau histo dans lequel il peut librement incrémenter ses compteurs mais cela utiliserait beaucoup de mémoire locale et conduirait à un gros travail de synthèse à la terminaison de tous les threads. La solution intermédiaire présentée ici, consiste à créer une zone partagée "shared" accessible en lecture / écriture à l'ensemble des threads d'un même block.

Comme il y a moins de threads potentiellement intéressés par ce tableau, il y a moins de concurrence et comme le tableau est local au block, il est alloué dans une zone proche des threads. Globalement, l'accès est donc beaucoup plus rapide. Par contre, il ne faut pas oublier de consolider le résultat à la fin du block. Cette consolidation doit se faire en exclusion mutuelle.

On voit apparaitre un équilibre à trouver entre le nombre de block (plus il est important et plus on sépare les traitements mais plus on aura de travail lors de la consolidation) et le nombre des threads (plus ils sont nombreux et plus on parallélise mais plus il y a de concurrence sur le tableau partagé).

Remarque : le choix aui est fait ici est de créer autant de threads qu'il y a de symboles à compter (et donc de cases dans le tableau histogramme). Cela permet d'utiliser l'astuce suivante, chaque thread s'occupe de l'initialisation et de la consolidation d'une case du tableau. Chacune de ces opérations étant réalisée en parallèle sur les 256 threads.

Lien sur hist_gpu_shmem_atomics.cu.

```
#include <stdio.h>
#define SIZE    (100*1024*1024)
__global__ void histo_kernel( unsigned char *buffer,
                              long size,
                              unsigned int *histo ) {
    // chaque thread initialise une case du tableau
    __shared__  unsigned int temp[256];
    temp[threadIdx.x] = 0;
    __syncthreads();
    // calculate the starting index and the offset to the next
    // block that each thread will be processing
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;
    while (i < size) {
        atomicAdd( &temp[buffer[i]], 1 );
        i += stride;
```

```
    }
    //A la fin des threads, chacun consolide dans la mémoire globale une case du
tableau
    __syncthreads();
    atomicAdd( &(histo[threadIdx.x]), temp[threadIdx.x] );
}
int main( void ) {
    unsigned char *buffer =
                    (unsigned char*)big_random_block( SIZE );
    cudaEvent_t     start, stop;
    cudaEventCreate( &start );
    cudaEventCreate( &stop );
    cudaEventRecord( start, 0 );
    // allocate memory on the GPU for the file's data
    unsigned char *dev_buffer;
    unsigned int *dev_histo;
    cudaMalloc( (void**)&dev_buffer, SIZE );
    cudaMemcpy( dev_buffer, buffer, SIZE,
                              cudaMemcpyHostToDevice );
    cudaMalloc( (void**)&dev_histo,
                              256 * sizeof( int ) );
    cudaMemset( dev_histo, 0,
                              256 * sizeof( int ) );
    // kernel launch - 2x the number of mps gave best timing
    cudaDeviceProp  prop;
    cudaGetDeviceProperties( &prop, 0 );
    int blocks = prop.multiProcessorCount;
    histo_kernel<<<blocks*2,256>>>( dev_buffer,
                                    SIZE, dev_histo );
    unsigned int    histo[256];
    cudaMemcpy( histo, dev_histo,
                              256 * sizeof( int ),
                              cudaMemcpyDeviceToHost );
    // get stop time, and display the timing results
    cudaEventRecord( stop, 0 );
    cudaEventSynchronize( stop );
    float   elapsedTime;
    cudaEventElapsedTime( &elapsedTime,
                                        start, stop );
    printf( "Time to generate:  %3.1f ms\n", elapsedTime );
    long histoCount = 0;
    for (int i=0; i<256; i++) {
        histoCount += histo[i];
    }
    printf( "Histogram Sum:  %ld\n", histoCount );
    // verify that we have the same counts via CPU
    for (int i=0; i<SIZE; i++)
        histo[buffer[i]]--;
    for (int i=0; i<256; i++) {
        if (histo[i] != 0)
            printf( "Failure at %d!\n", i );
```

```
    }
    cudaEventDestroy( start );
    cudaEventDestroy( stop );
    cudaFree( dev_histo );
    cudaFree( dev_buffer );
    free( buffer );
    return 0;
}
```

4.1.2. Produit scalaire : calcul parallèle et réduction

L'exemple suivant reprend l'intérêt de mettre en place une zone de mémoire partagée pour accélérer les traitements mais aussi affine le calcul d'une réduction puisqu'on veut un scalaire et non un tableau au final.

Lien sur dot.cu.

```
#include <stdio.h>
#define imin(a,b) (a<b?a:b)
const int N = 33 * 1024;
const int threadsPerBlock = 256;
const int blocksPerGrid =
            imin( 32, (N+threadsPerBlock-1) / threadsPerBlock );
__global__ void dot( float *a, float *b, float *c ) {
    __shared__ float cache[threadsPerBlock];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int cacheIndex = threadIdx.x;
    float   temp = 0;
    while (tid < N) {
        temp += a[tid] * b[tid];
        tid += blockDim.x * gridDim.x;
    }
    // set the cache values
    cache[cacheIndex] = temp;
    // synchronize threads in this block
    __syncthreads();
    // for reductions, threadsPerBlock must be a power of 2
    // because of the following code
    int i = blockDim.x/2;
    while (i != 0) {
        if (cacheIndex < i)
            cache[cacheIndex] += cache[cacheIndex + i];
        __syncthreads();
        i /= 2;
    }
    if (cacheIndex == 0)
        c[blockIdx.x] = cache[0];
}
int main( void ) {
    float   *a, *b, c, *partial_c;
```

```
    float   *dev_a, *dev_b, *dev_partial_c;
    // allocate memory on the cpu side
    a = (float*)malloc( N*sizeof(float) );
    b = (float*)malloc( N*sizeof(float) );
    partial_c = (float*)malloc( blocksPerGrid*sizeof(float) );
    // allocate the memory on the GPU
    cudaMalloc( (void**)&dev_a, N*sizeof(float) );
    cudaMalloc( (void**)&dev_b, N*sizeof(float) );
    cudaMalloc( (void**)&dev_partial_c, blocksPerGrid*sizeof(float) );
    // fill in the host memory with data
    for (int i=0; i<N; i++) {
        a[i] = i;
        b[i] = i*2;
    }
    // copy the arrays 'a' and 'b' to the GPU
    cudaMemcpy( dev_a, a, N*sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, b, N*sizeof(float), cudaMemcpyHostToDevice );
    dot<<<blocksPerGrid,threadsPerBlock>>>( dev_a, dev_b,
                                            dev_partial_c );
    // copy the array 'c' back from the GPU to the CPU
    cudaMemcpy( partial_c, dev_partial_c,
                            blocksPerGrid*sizeof(float),
                            cudaMemcpyDeviceToHost );
    // finish up on the CPU side
    c = 0;
    for (int i=0; i<blocksPerGrid; i++) {
        c += partial_c[i];
    }
    #define sum_squares(x)  (x*(x+1)*(2*x+1)/6)
    printf( "Does GPU value %.6g = %.6g?\n", c,
             2 * sum_squares( (float)(N - 1) ) );
    // free memory on the gpu side
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_partial_c );
    // free memory on the cpu side
    free( a );
    free( b );
    free( partial_c );
}
```

Normalement, sur la base de ces exemples, vous devez avoir compris :

- ☑ l'intérêt de la zone de mémoire partagée
- ☑ comment tirer partie de cette zone de mémoire à accès rapide mais locale au block
- ☑ Comment mettre en oeuvre une réduction (c'est plus compliqué qu'avec OpenMP)

4.2. Mémoire constante

Il existe une autre zone mémoire intéressante, la mémoire constante. De taille limitée, elle est accessible en lecture uniquement. Son accès est plus rapide parce qu'elle profite implicitement des zones de cache. Dès qu'un thread y accède, les autres threads du block n'auront pas besoin d'attendre un accès à la mémoire globale, la donnée se trouve accessible (et correcte parce que non modifiable) dans la zone de cache locale.

L'exemple suiavnt montre un calcul d'image basé sur le lancement de rayon sur des sphères. Les caractéristiques des sphères sont positionnées dans la mémoire constante et tous les threads y accèdent rapidement. Les deux versions ci-dessous proposent la version sans mémoire constante et celle utilisant la mémoire constante.

Lien sur ray_noconst.cu.

```
#include "cuda.h"
#include "../common/book.h"
#include "../commonMac/cpu_bitmap.h"
#define DIM 1024
#define rnd( x ) (x * rand() / RAND_MAX)
#define INF     2e10f
struct Sphere {
    float   r,b,g;
    float   radius;
    float   x,y,z;
    __device__ float hit( float ox, float oy, float *n ) {
        float dx = ox - x;
        float dy = oy - y;
        if (dx*dx + dy*dy < radius*radius) {
            float dz = sqrtf( radius*radius - dx*dx - dy*dy );
            *n = dz / sqrtf( radius * radius );
            return dz + z;
        }
        return -INF;
    }
};
#define SPHERES 20
__global__ void kernel( Sphere *s, unsigned char *ptr ) {
    // map from threadIdx/BlockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;
    float   ox = (x - DIM/2);
    float   oy = (y - DIM/2);
    float   r=0, g=0, b=0;
    float   maxz = -INF;
    for(int i=0; i<SPHERES; i++) {
        float   n;
        float   t = s[i].hit( ox, oy, &n );
        if (t > maxz) {
```

```
            float fscale = n;
            r = s[i].r * fscale;
            g = s[i].g * fscale;
            b = s[i].b * fscale;
            maxz = t;
        }
    }
    ptr[offset*4 + 0] = (int)(r * 255);
    ptr[offset*4 + 1] = (int)(g * 255);
    ptr[offset*4 + 2] = (int)(b * 255);
    ptr[offset*4 + 3] = 255;
}
//
// globals needed by the update routine
struct DataBlock {
    unsigned char   *dev_bitmap;
    Sphere          *s;
};
//
int main( void ) {
    DataBlock   data;
    // capture the start time
    cudaEvent_t     start, stop;
    cudaEventCreate( &start );
    cudaEventCreate( &stop );
    cudaEventRecord( start, 0 );
//
    CPUBitmap bitmap( DIM, DIM, &data );
    unsigned char   *dev_bitmap;
    Sphere          *s;
//
    // allocate memory on the GPU for the output bitmap
    cudaMalloc( (void**)&dev_bitmap,
                               bitmap.image_size() );
    // allocate memory for the Sphere dataset
    cudaMalloc( (void**)&s,
                               sizeof(Sphere) * SPHERES );
    // allocate temp memory, initialize it, copy to
    // memory on the GPU, then free our temp memory
    Sphere *temp_s = (Sphere*)malloc( sizeof(Sphere) * SPHERES );
    for (int i=0; i<SPHERES; i++) {
        temp_s[i].r = rnd( 1.0f );
        temp_s[i].g = rnd( 1.0f );
        temp_s[i].b = rnd( 1.0f );
        temp_s[i].x = rnd( 1000.0f ) - 500;
        temp_s[i].y = rnd( 1000.0f ) - 500;
        temp_s[i].z = rnd( 1000.0f ) - 500;
        temp_s[i].radius = rnd( 100.0f ) + 20;
    }
    cudaMemcpy( s, temp_s,
                                 sizeof(Sphere) * SPHERES,
```

```
                                cudaMemcpyHostToDevice );
    free( temp_s );
    // generate a bitmap from our sphere data
    dim3    grids(DIM/16,DIM/16);
    dim3    threads(16,16);
    kernel<<<grids,threads>>>( s, dev_bitmap );
    // copy our bitmap back from the GPU for display
    cudaMemcpy( bitmap.get_ptr(), dev_bitmap,
                              bitmap.image_size(),
                              cudaMemcpyDeviceToHost );
    // get stop time, and display the timing results
    cudaEventRecord( stop, 0 );
    cudaEventSynchronize( stop );
    float   elapsedTime;
    cudaEventElapsedTime( &elapsedTime,
                                        start, stop );
    printf( "Time to generate:  %3.1f ms\n", elapsedTime );
    cudaEventDestroy( start );
    cudaEventDestroy( stop );
    cudaFree( dev_bitmap );
    cudaFree( s );
    // display
    bitmap.display_and_exit();
}
```

Lien sur ray.cu.

```
#include "cuda.h"
#include "../common/book.h"
#include "../commonMac/cpu_bitmap.h"
#define DIM 1024
#define rnd( x ) (x * rand() / RAND_MAX)
#define INF     2e10f
struct Sphere {
    float   r,b,g;
    float   radius;
    float   x,y,z;
    __device__ float hit( float ox, float oy, float *n ) {
        float dx = ox - x;
        float dy = oy - y;
        if (dx*dx + dy*dy < radius*radius) {
            float dz = sqrtf( radius*radius - dx*dx - dy*dy );
            *n = dz / sqrtf( radius * radius );
            return dz + z;
        }
        return -INF;
    }
};
#define SPHERES 20
__constant__ Sphere s[SPHERES];
```

```
__global__ void kernel( unsigned char *ptr ) {
    // map from threadIdx/BlockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;
    float   ox = (x - DIM/2);
    float   oy = (y - DIM/2);
    float   r=0, g=0, b=0;
    float   maxz = -INF;
    for(int i=0; i<SPHERES; i++) {
        float   n;
        float   t = s[i].hit( ox, oy, &n );
        if (t > maxz) {
            float fscale = n;
            r = s[i].r * fscale;
            g = s[i].g * fscale;
            b = s[i].b * fscale;
            maxz = t;
        }
    }
    ptr[offset*4 + 0] = (int)(r * 255);
    ptr[offset*4 + 1] = (int)(g * 255);
    ptr[offset*4 + 2] = (int)(b * 255);
    ptr[offset*4 + 3] = 255;
}
// globals needed by the update routine
struct DataBlock {
    unsigned char   *dev_bitmap;
};
//
int main( void ) {
    DataBlock   data;
    // capture the start time
    cudaEvent_t     start, stop;
    cudaEventCreate( &start );
    cudaEventCreate( &stop );
    cudaEventRecord( start, 0 );
    CPUBitmap bitmap( DIM, DIM, &data );
    unsigned char   *dev_bitmap;
    // allocate memory on the GPU for the output bitmap
    cudaMalloc( (void**)&dev_bitmap,
                               bitmap.image_size() );
    // allocate temp memory, initialize it, copy to constant
    // memory on the GPU, then free our temp memory
    Sphere *temp_s = (Sphere*)malloc( sizeof(Sphere) * SPHERES );
    for (int i=0; i<SPHERES; i++) {
        temp_s[i].r = rnd( 1.0f );
        temp_s[i].g = rnd( 1.0f );
        temp_s[i].b = rnd( 1.0f );
        temp_s[i].x = rnd( 1000.0f ) - 500;
        temp_s[i].y = rnd( 1000.0f ) - 500;
```

```
        temp_s[i].z = rnd( 1000.0f ) - 500;
        temp_s[i].radius = rnd( 100.0f ) + 20;
    }
    cudaMemcpyToSymbol( s, temp_s,
                                sizeof(Sphere) * SPHERES);
    free( temp_s );
    // generate a bitmap from our sphere data
    dim3    grids(DIM/16,DIM/16);
    dim3    threads(16,16);
    kernel<<<grids,threads>>>( dev_bitmap );
    // copy our bitmap back from the GPU for display
    cudaMemcpy( bitmap.get_ptr(), dev_bitmap,
                              bitmap.image_size(),
                              cudaMemcpyDeviceToHost );
    // get stop time, and display the timing results
    cudaEventRecord( stop, 0 );
    cudaEventSynchronize( stop );
    float   elapsedTime;
    cudaEventElapsedTime( &elapsedTime,
                                       start, stop );
    printf( "Time to generate:  %3.1f ms\n", elapsedTime );
    cudaEventDestroy( start );
    cudaEventDestroy( stop );
    cudaFree( dev_bitmap )
    // display
    bitmap.display_and_exit();
}
```

5. Conclusion

Un complément à regarder est la manière de masquer les tranferts avec le GPU en utilisant les "streams". Du point de vue de l'optimisation globale, on peut avoir intérêt à découper un flux de donner en deux (ou plus) et ainsi cacher une partie des transferts vers le GPU (ou depuis le GPU) par les traitements ralisés sur l'autre moitié du flux.

- basic_double_stream_correct.cu

Le dernier exemple important montre comment tirer partie de plusieurs GPU présents sur la machine et donc comment associer les demandes de traitements à un GPU en particulier. On utilise cudaSetDevice() pour préciser le GPU concerné par les instructions suivantes. Les autres éléments permettent de séparer les données pour chaque stream et associe un thread à chaque GPU pour associer les synchronisations liés à un GPU avec un seul thread.

- multidevice.cu