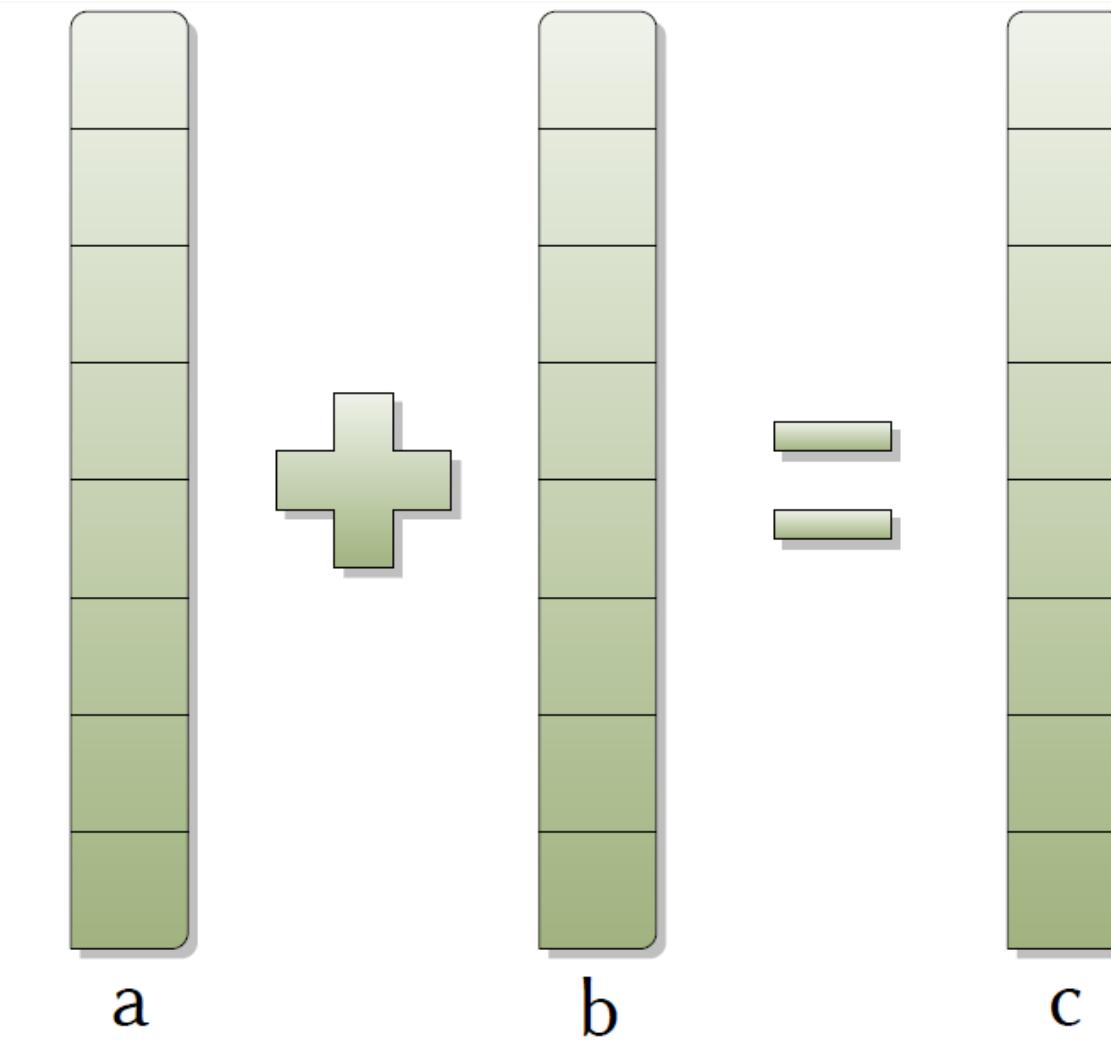




Nvidia CUDA

Addition de vecteurs



Addition de vecteurs

Un kernel qui additionne deux entiers

```
__global__ void add( int *a, int * b, int * c ) {  
    *c = *a + *b;  
}
```

Comme précédemment

`__global__` signifie que :

- `add()` s'exécutera sur le GPU
- `add()` est appelé depuis l'hôte

Addition de vecteurs

Notez que nous utilisons des pointeurs pour les variables

```
__global__ void add( int *a, int * b, int * c ) {  
    *c = *a + *b;  
}
```

- `add()` s'exécutera sur le GPU, donc `a`, `b` et `c` doivent pointer sur de la mémoire GPU
- Nous devons allouer de la mémoire sur le GPU

Addition de vecteurs

Mémoire hôte et GPU sont différentes

- Les pointeurs **device** pointent sur la mémoire GPU
 - Peuvent être passés depuis/vers du code hôte
 - Ne peuvent être déréférés dans du code hôte
- Les pointeurs **hôte** pointent sur la mémoire CPU
 - Peuvent être passés depuis/vers du code GPU
 - Ne peuvent être déréférés dans du code GPU

API

- `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
- Similaires à leurs équivalents C `malloc()`, `free()`, `memcpy()`

Addition de vecteurs

main()

```
int main (void) {  
  
    int h_a, h_b, h_c;      // copies sur l'hôte de a, b et c  
    int *d_a, *d_b, *d_c;  // copies sur le GPU de a, b et c  
    int size = sizeof(int);  
  
    // Allocation de l'espace nécessaire  
    pour les copies de a, b et c  
    cudaMalloc((void **) &d_a, size);  
    cudaMalloc((void **) &d_b, size);  
    cudaMalloc((void **) &d_c, size);  
  
    // Valeurs d'entrées  
    h_a = 7;  
    h_b = 42;
```

Addition de vecteurs

main()

```
// Copie des entrées sur le GPU
cudaMemcpy(d_a, &h_a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &h_b, size, cudaMemcpyHostToDevice);

// Lancement de add() sur le GPU
add<<< 1, 1 >>> (d_a, d_b, d_c);

// Copie du résultat en mémoire hôte
cudaMemcpy(&h_c, d_c, size, cudaMemcpyDeviceToHost);

// Nettoyage
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);

return 0;
}
```

Addition de vecteurs (blocs)

Comment lancer du code en parallèle

```
add<<< 1, 1 >>> ( d_a , d_b , d_c ) ;
```



```
add<<< N, 1 >>> ( d_a , d_b , d_c ) ;
```

Au lieu d'exécuter add() 1 fois, exécute add() N fois en parallèle.

Addition de vecteurs (blocs)

Blocs

Chaque invocation parallèle de add() sera associée à un bloc (*block*)

- L'ensemble des blocs est appelé une grille (*grid*)
- Chaque invocation peut accéder à son indice de bloc en utilisant **blockIdx.x**

```
__global__ void add( int *a, int *b, int *c ) {  
    c[ blockIdx.x ] = a[ blockIdx.x ] + b[ blockIdx.x ];  
}
```

En utilisant **blockIdx.x** pour indiquer les tableaux, chaque bloc gère un indice différent

Addition de vecteurs (blocs)

Blocs

Chaque invocation parallèle de add() sera associée à un bloc (*block*)

- L'ensemble des blocs est appelé une grille (*grid*)
- Chaque invocation peut accéder à son indice de bloc en utilisant **blockIdx.x**

```
__global__ void add(int *a, int *b, int *c){  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

Block 0

c[0] = a[0] + b[0];

Block 1

c[1] = a[1] + b[1];

Block 2

c[2] = a[2] + b[2];

Block 3

c[3] = a[3] + b[3];

Addition de vecteurs (blocs)

main()

```
#define N 512
int main (void) {

    int *h_a , *h_b , *h_c ;      // copies sur l'hôte de a, b et c
    int *d_a , *d_b , *d_c ; // copies sur le GPU de a, b et c
    int size = N*sizeof(int);

    // Allocation de l'espace nécessaire
    pour les copies de a, b et c
    cudaMalloc((void **) &d_a , size);
    cudaMalloc((void **) &d_b , size);
    cudaMalloc((void **) &d_c , size);

    // Allocation de l'espace pour les copies en mémoire hôte
    // et initialisation
    h_a = (int *) malloc(size); random_ints(a , N);
    h_b = (int *) malloc(size); random_ints(b , N);
    h_c = (int *) malloc(size);
```

Addition de vecteurs (blocs)

```
main()
```

```
// Copie des entrées sur le GPU
cudaMemcpy(d_a, &h_a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &h_b, size, cudaMemcpyHostToDevice);

// Lancement de add() sur le GPU avec N blocs
add<<< N, 1 >>> (d_a, d_b, d_c);

// Copie du résultat en mémoire hôte
cudaMemcpy(&h_c, d_c, size, cudaMemcpyDeviceToHost);

// Nettoyage
free(a); free(b); free(c);
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);

return 0;
}
```

Addition de vecteurs (Threads)

Threads

Un bloc peut être découpé en *threads*

- Changeons add() pour utiliser des *threads* parallèles au lieu de blocs parallèles

```
__global__ void add( int *a, int * b, int * c ) {  
    c[ threadIdx.x ] = a[ threadIdx.x ] + b[ threadIdx.x ];  
}
```

- Nous utilisons **threadIdx.x** au lieu de **blockIdx.x**
- Il faut faire quelques changements dans main()

Addition de vecteurs (Threads)

main()

```
#define N 512
int main (void) {

    int *h_a, *h_b, *h_c;      // copies sur l'hôte de a, b et c
    int *d_a, *d_b, *d_c; // copies sur le GPU de a, b et c
    int size = N*sizeof(int);

    // Allocation de l'espace nécessaire
    pour les copies de a, b et c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Allocation de l'espace pour les copies en mémoire hôte
    // et initialisation
    h_a = (int *) malloc(size); random_ints(a, N);
    h_b = (int *) malloc(size); random_ints(b, N);
    h_c = (int *) malloc(size);
```

Addition de vecteurs (Threads)

main()

```
// Copie des entrées sur le GPU
cudaMemcpy(d_a, &h_a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &h_b, size, cudaMemcpyHostToDevice);

// Lancement de add() sur le GPU avec N threads
add<<< 1, N >>> (d_a, d_b, d_c);

// Copie du résultat en mémoire hôte
cudaMemcpy(&h_c, d_c, size, cudaMemcpyDeviceToHost);

// Nettoyage
free(a); free(b); free(c);
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);

return 0;
}
```

Addition de vecteurs (Blocs + Threads)

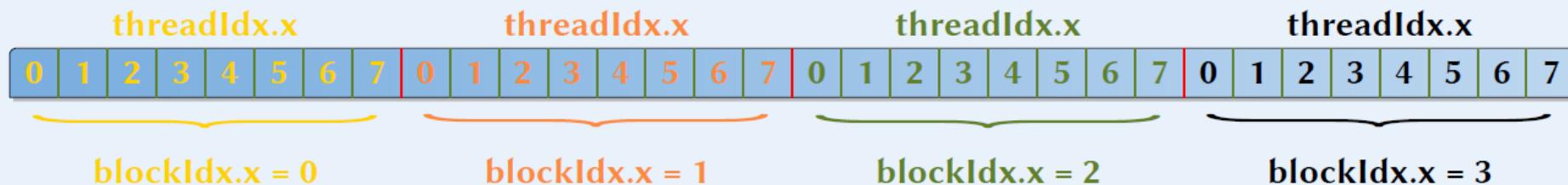
Combinons blocs et *threads*

- Nous avons vu :
 - Plusieurs blocs de 1 *thread* chacun
 - 1 seul bloc de plusieurs *threads*
- Adaptons notre code pour utiliser à la fois blocs et *threads*
- Pourquoi ? (nous y reviendrons plus tard)
- D'abord, discutons indexation de données

Addition de vecteurs (Blocs + Threads)

Indices de tableaux avec blocs et *threads*

- Plus aussi simple que d'utiliser `blockIdx.x` ou `threadIdx.x`
- Considérons les indices d'un tableaux avec un élément par *thread* et 8 *threads* par bloc

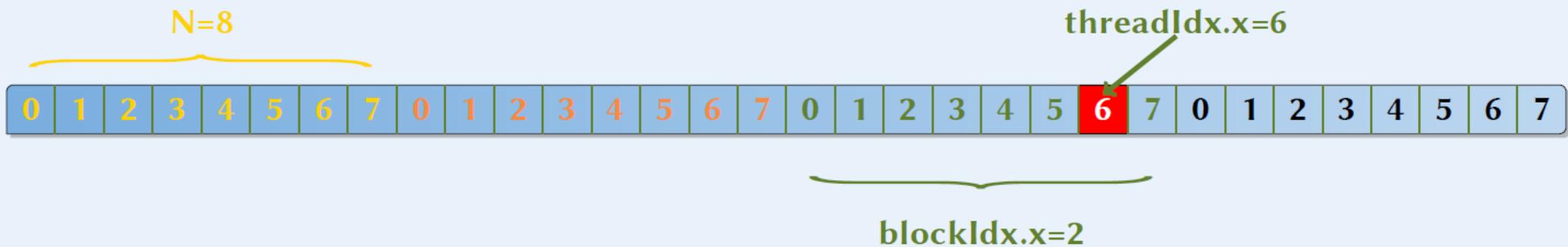


- Avec N *threads* par bloc, un indice unique est donné par :

```
int index = blockIdx.x * N + threadIdx.x
```

Addition de vecteurs (Blocs + Threads)

Quel thread travaille sur l'élément rouge ?



```
int index = N * blockIdx.x + threadIdx.x  
= 8 * 2 + 6  
= 22;
```

Addition de vecteurs (Blocs + Threads)

Utilisez la variable `blockDim.x` pour les *threads* par bloc

```
int index = blockIdx.x * blockDim.x + threadIdx.x;
```

Version combinée de `add()` avec blocs et *threads* parallèles

```
__global__ void add(int *a, int *b, int *c){  
    int index = blockIdx.x * blockDim.x + threadIdx.x;  
    c[index] = a[index] + b[index];  
}
```

Quelles modifications faut il apporter à `main()` ?

Addition de vecteurs (Blocs + Threads)

main()

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512

int main (void) {

    int *h_a, *h_b, *h_c;      // copies sur l'hôte de a, b et c
    int *d_a, *d_b, *d_c; // copies sur le GPU de a, b et c
    int size = N*sizeof(int);

    // Allocation de l'espace nécessaire
    pour les copies de a, b et c
    cudaMalloc((void**) &d_a, size);
    cudaMalloc((void**) &d_b, size);
    cudaMalloc((void**) &d_c, size);

    // Allocation de l'espace pour les copies en mémoire hôte
    // et initialisation
    h_a = (int*) malloc(size); random_ints(a, N);
    h_b = (int*) malloc(size); random_ints(b, N);
    h_c = (int*) malloc(size);
```

Addition de vecteurs (Blocs + Threads)

main()

```
// Copie des entrées sur le GPU
cudaMemcpy(d_a, &h_a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &h_b, size, cudaMemcpyHostToDevice);

// Lancement de add() sur le GPU avec N threads
add<<< N/THREADS_PER_BLOCK, THREADS_PER_BLOCK >>> (d_a, d_b, d_c);

// Copie du résultat en mémoire hôte
cudaMemcpy(&h_c, d_c, size, cudaMemcpyDeviceToHost);

// Nettoyage
free(a); free(b); free(c);
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);

return 0;
}
```

Fin