

IF5-PARI : TP2 Calcul parallèle sur GPU

Traitement d'image : Raytracing

Jacquemin Thibault

Marleix Mathieu



Table des matières

Introduction.....	3
I. Réduction	4
A. Somme dans un thread	5
B. Somme partielle de chaque bloc.....	6
C. Somme finale.....	6
II. Scan	9
A. Type de scan	9
B. Principe du scan.....	11
Conclusion	15



Introduction

Le raytracing est une technique utilisée pour le rendu d'images de haute qualité.

Nous nous sommes intéressés dans ce TP à trois étapes basiques nécessaires à sa réalisation :

- La réduction (d'une somme) ;
- Le scan (ou somme des préfixes) ;
- Le radix sort (un tri sur entiers naturels) que nous n'avons pas eu le temps d'implémenter.



I. Réduction

Le but de cette partie est de réduire la complexité d'une opération longue (ici somme de n éléments) en plusieurs opérations binaires, toutes exécutables en parallèle.

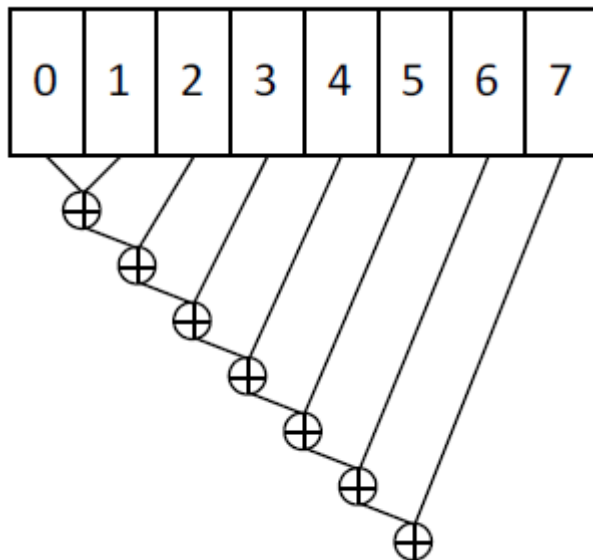
En séquentiel, l'algorithme se présente sous cette forme :

```
int sum = 0;
for( int i = 0 ; i < n ; ++i ) sum += a[i];
```

On peut voir que c'est un algorithme de complexité linéaire.

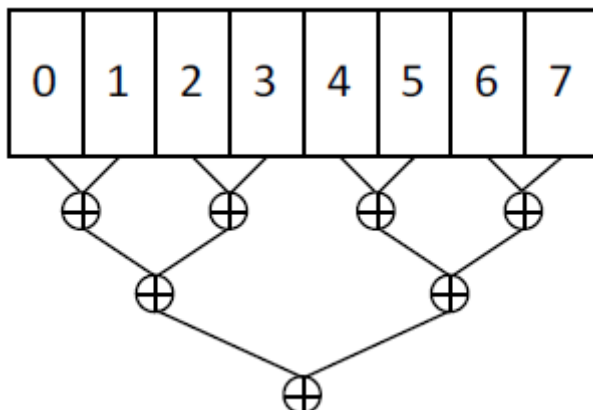
Il se déroule sur un processeur de la façon suivante :

$(a_0 + (a_1 + (a_2 + (a_3 + (a_4 + (a_5 + (a_6 + a_7)))))))$



Le fonctionnement optimal visé s'illustre de la façon suivante demandant au plus $\log_2(n)$ si l'on dispose de $n/2$ pouvant exécuter les instructions en parallèle :

$((a_0 + a_1) + (a_2 + a_3)) + ((a_4 + a_5) + (a_6 + a_7))$



Pour disposer de ce nombre de processeurs, nous allons utiliser les capacités de calcul parallèle d'une carte graphique Nvidia avec CUDA, application complètement adaptée car ce sont toujours les mêmes calculs répétitifs qui doivent être exécutés.

Les unités de calcul sur une telle carte s'organisent en blocs de threads, chaque thread étant alloué à une plage de données qu'il devra traiter complètement avant d'en passer à une autre.

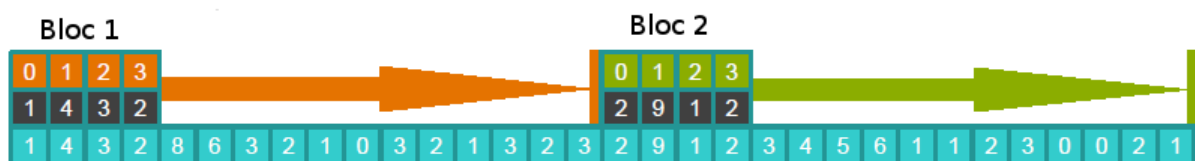
Les données sont d'abord chargées dans la mémoire que se partagent les threads d'un même bloc, les calculs y sont traités par bloc, et enfin globalisés par un seul bloc en général.

Il y a trois grandes lignes pour réaliser une réduction :

- Chaque thread réalise l'opération sur sa plage de donnée ;
- Puis cette même opération est réalisée sur les résultats partiels des threads d'un même bloc ;
- Enfin, un bloc synthétise les résultats partiels de tous les blocs pour obtenir le résultat final.

A. Somme dans un thread

Un bloc de thread se positionne au début de la plage de données qui lui est attribué. Les threads enregistrent alors la valeur correspondant à leur indice en mémoire partagée. Le bloc de thread avance alors par 'bonds successifs', les threads sommant à chaque fois les nouvelles valeurs tombant à leur indice en mémoire partagée.



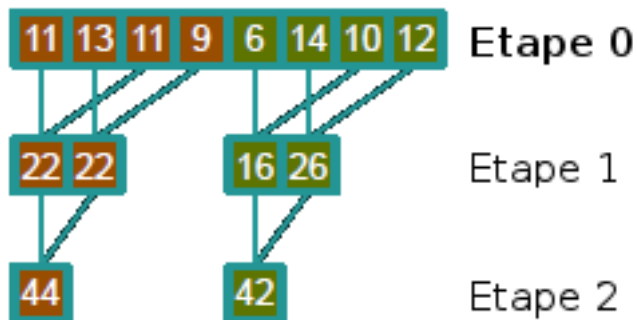
Il faut ensuite attendre que chaque bloc ait fini de se déplacer en mettant en place une barrière de synchronisation.



B. Somme partielle de chaque bloc

Les sommes partielles de chaque thread sont stockées dans la mémoire partagée du bloc.

C'est ici que se fait l'étape de réduction à proprement parlé. A la première étape, la première moitié des threads va exécuter une opération binaire et stocker le résultat dans la cellule correspondant à son indice. Puis à chaque itération, cette moitié sera divisée par 2 jusqu'à ce qu'il ne reste plus qu'une seule case contenant la somme partielle du bloc à inscrire en mémoire globale.



C. Somme finale

Le principe pour cette somme est le même que celui des deux étapes précédentes, à ceci près que c'est un seul bloc qui travaillera, sur les sommes partielles de chaque bloc stockées en mémoire globale.

Voici le code à exécuter sur la carte graphique :

```

////////////////////////////////////
////////////////////////////////////

// GPU REDUCTION

////////////////////////////////////
////////////////////////////////////

__global__ void reduce_kernel( int n, const int *in_buffer, int
*out_buffer, const int2 *block_ranges )
{
    // Allocate shared memory inside the block.
    extern __shared__ int s_mem[];

    // The range of data to work with.
    int2 range = block_ranges[blockIdx.x];

    // Compute the sum of my elements.
    int my_sum = 0;

    // TODO: fill-in that section of the code
    for(int i=range.x; i<range.y; i+=blockDim.x) {
        my_sum += in_buffer[threadIdx.x+i];
    }
}

```



```

// Copy my sum in shared memory.
s_mem[threadIdx.x] = my_sum;

// Make sure all the threads have copied their value in shared memory.
__syncthreads();

// Compute the sum inside the block.

// TODO: fill-in that section of the code
for (int reduceSize = blockDim.x>>1; reduceSize>=1; reduceSize>>=1) {
    if(threadIdx.x < reduceSize) {
        s_mem[threadIdx.x] += s_mem[reduceSize + threadIdx.x];
        __syncthreads();
    }
}

// The first thread of the block stores its result.
if( threadIdx.x == 0 )
    out_buffer[blockIdx.x] = s_mem[0];
}

int reduce_on_gpu( int n, const int *a_device )
{
    // Compute the size of the grid.
    const int BLOCK_DIM    = 256;
    const int grid_dim      = std::min( BLOCK_DIM, (n + BLOCK_DIM-1) /
BLOCK_DIM );
    const int num_threads = BLOCK_DIM * grid_dim;

    // Compute the number of elements per block.
    const int elements_per_block = BLOCK_DIM * ((n + num_threads - 1) /
num_threads);

    // Allocate memory for temporary buffers.
    int *partial_sums = NULL;
    int2 *block_ranges = NULL;

    CUDA_SAFE_CALL( cudaMalloc( (void **) &partial_sums, BLOCK_DIM *
sizeof(int) ) );
    CUDA_SAFE_CALL( cudaMalloc( (void **) &block_ranges, grid_dim *
sizeof(int2) ) );

    // Compute the ranges for the blocks.
    int sum = 0;
    int2 *block_ranges_on_host = new int2[grid_dim];
    for( int block_idx = 0 ; block_idx < grid_dim ; ++block_idx )
    {
        block_ranges_on_host[block_idx].x = sum;
        block_ranges_on_host[block_idx].y = std::min( sum +=
elements_per_block, n );
    }
    CUDA_SAFE_CALL( cudaMemcpy( block_ranges, block_ranges_on_host, grid_dim
* sizeof(int2), cudaMemcpyHostToDevice ) );
    delete[] block_ranges_on_host;

    // First round: Compute a partial sum for all blocks.
    reduce_kernel<<<grid_dim, BLOCK_DIM, BLOCK_DIM*sizeof(int)>>>( n,
a_device, partial_sums, block_ranges );
    CUDA_SAFE_CALL( cudaGetLastError() );

```



```
// Set the ranges for the second kernel call.
int2 block_range = make_int2( 0, grid_dim );
CUDA_SAFE_CALL( cudaMemcpy( block_ranges, &block_range, sizeof(int2),
cudaMemcpyHostToDevice ) );

// Second round: Compute the final sum by summing the partial results of
all blocks.
reduce_kernel<<<1, BLOCK_DIM, BLOCK_DIM*sizeof(int)>>>( grid_dim,
partial_sums, partial_sums, block_ranges );
CUDA_SAFE_CALL( cudaGetLastError() );

// Read the result from device memory.
int result;
CUDA_SAFE_CALL( cudaMemcpy( &result, partial_sums, sizeof(int),
cudaMemcpyDeviceToHost ) );

// Free temporary memory.
CUDA_SAFE_CALL( cudaFree( block_ranges ) );
CUDA_SAFE_CALL( cudaFree( partial_sums ) );

return result;
}
```

Pour améliorer l'algorithme, il existe des divisions à l'intérieur de chaque bloc dans lesquels les threads peuvent communiquer entre eux, appelées warp.

L'idée pour obtenir un code plus rapide est de :

- Premièrement, générer les sommes partielles pour chaque thread ;
- Deuxièmement, générer les sommes partielles pour chaque warp en s'appuyant sur la propriété de communication inter-threads pour perdre moins de temps en accès mémoire ;
- Troisièmement, générer les sommes partielles pour chaque bloc où l'on réalise encore un gain en accès mémoire vu qu'il y a moins de warp que de threads ;
- Quatrièmement, générer la somme totale.



II. Scan

Le scan (ou somme parallèle des préfixes) est une étape utile en programmation parallèle sur laquelle repose beaucoup d'algorithmes comme le quick sort ou le radix sort par exemple.

Il permet d'obtenir un tableau des effectifs cumulés croissants :

$[a_0, (a_0+a_1), \dots (a_0+a_1+\dots+a_{N-1})]$.

Il peut être inclusif – c'est-à-dire tenir compte de la valeur i dans la somme à la i -ème position – comme dans l'exemple ci-dessus, ou exclusif (et donc ne pas tenir compte de la valeur i ...).

L'algorithme séquentiel d'un scan s'écrit sous la forme suivante :

```
int sum = 0;
for( int i = 0 ; i < n ; ++i ) {
    b[i] = sum;
    sum += a[i];
}
```

C'est un algorithme de complexité linéaire ici présent dans sa forme exclusive. Pour un scan inclusif, il suffit d'inverser la ligne d'assignation dans $b[i]$ et celle de l'ajout à sum .

A. Type de scan

L'idée du scan parallèle est d'effectuer un scan par bloc, stocker les sommes partielles et exécuter un scan dessus pour ensuite additionner ces nouvelles sommes aux données appartenant à ce bloc.

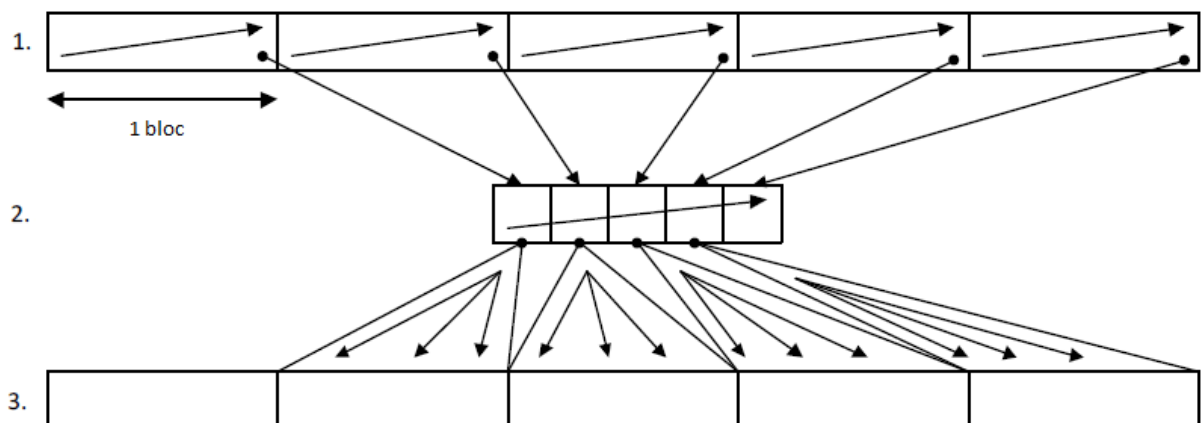
La méthodologie pour y parvenir se déroule en trois grandes étapes, néanmoins il existe plusieurs variations dans l'implémentation d'un scan parallèle comme :

- Le scan puis distribution :

La première étape lance des scans par bloc en parallèle, desquelles on peut en retirer les sommes partielles de chaque bloc qu'on stocke ;

La deuxième étape consiste en un scan sur les sommes partielles ;

La troisième étape ajoute les valeurs issues du scan précédent aux valeurs appartenant au bon bloc en sortie.

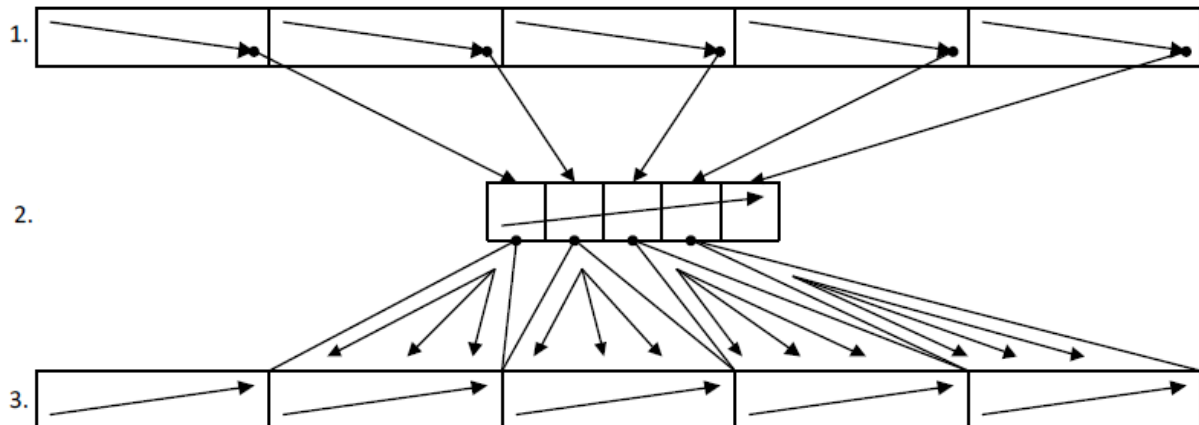


- La réduction puis scan :

La première étape exécute une réduction pour générer les sommes partielles de chaque bloc qu'on stocke ;

La deuxième étape consiste en un scan sur les sommes partielles ;

La troisième étape lance des scans par bloc en parallèle et ajoute à la volée les valeurs issues du scan précédent aux valeurs appartenant au bon bloc en sortie.



C'est à cette deuxième variation que nous implémenterons. Comme la première étape a déjà été présentée, nous allons nous intéresser directement au fonctionnement du scan. Il faut retenir qu'avec ces réductions par bloc sera construit un tableau contenant toutes les sommes partielles de chaque bloc (appelé ici spline) sur lequel il faudra exécuter un scan.



B. Principe du scan

Tout d'abord, il faut charger autant de cases qu'il y a de threads dans un bloc, c'est-à-dire DIM_BLOCK cases

On veut obtenir en i-ème case la somme des i (ou i-1) premiers éléments. Il existe donc un thread qui fait au maximum DIM_BLOCK sommes en parcourant DIM_BLOCK cases. Or, le code exécuté est le même quel que soit le thread. Il faut donc que les autres threads qui tenteront également de parcourir DIM_BLOCK cases additionnent des zéros à la place de nombres qui n'existent pas.

Pour se faire, on va donc allouer un tableau de DIM_BLOCK cases remplis de zéros et s'assurer que les données que l'on veut charger se situent à côté en mémoire partagée, ce qui nous donnera un tableau 2*DIM_BLOCK ressemblant à ceci :

0	1	2	3	4	5	6	7
0	0	0	0	2	3	1	4

Exemple pour DIM_BLOCK = 4

Chaque thread est maintenant en mesure d'additionner les DIM_BLOCK cases se trouvant la case qui lui correspond en décalant à chaque étape d'une case vers la droite.

A la fin de cette exécution, le résultat du scan de chaque case sera contenu soit dans la case elle-même pour un scan inclusive, soit dans la case précédente.

NB : la somme partielle du bloc est récupérable dans la dernière case et l'on peut donc construire le spline avec des scans par blocs.

Voici le code à exécuter sur la carte graphique :

```

////////////////////////////////////
////////////////////////////////////

// GPU SCAN

////////////////////////////////////
////////////////////////////////////

__global__ void reduce_kernel( int n, const int *a, int *spine, const int2
*block_ranges )
{
    extern __shared__ int s_mem[]; // Size is blockDim.x

    // The range of data to work with.
    int2 range = block_ranges[blockIdx.x];

    // Compute the sum of my elements.
    int my_sum = 0, i;

    // TODO: fill-in that section of the code
    for(i=range.x; i<range.y; i+=blockDim.x) {
        my_sum += a[threadIdx.x+i];
    }
}

```



```
// Copy my sum in shared memory.
s_mem[threadIdx.x] = my_sum;

// Make sure all the threads have copied their value in shared memory.
__syncthreads();

// Compute the sum inside the block.

// TODO: fill-in that section of the code
for (int reduceSize = blockDim.x>>1; reduceSize>=1; reduceSize>>=1) {
    if(threadIdx.x < reduceSize) {
        s_mem[threadIdx.x] += s_mem[reduceSize + threadIdx.x];
    }
}

// The first thread of the block stores its result.
if( threadIdx.x == 0 )
    spine[blockIdx.x] = s_mem[0];
}

__device__ __forceinline__ int2 scan_block( int *s_mem, int item )
{

    // Where to store my element.
    int my_idx = threadIdx.x;

    // Copy item to shared memory and make sure the block_dim first values
    are 0.

    // TODO: fill-in that section of the code
    s_mem[my_idx] = 0;
    s_mem[my_idx += blockDim.x] = item;

    // Make sure all threads have already written to shared memory.
    __syncthreads();

    // Run the scan operation.

    // TODO: fill-in that section of the code
    for(int scanSize=1; scanSize <= blockDim.x; scanSize<<=1) {
        s_mem[my_idx] += s_mem[scanSize + my_idx];
        __syncthreads();
    }
    // My results.
    int2 results = make_int2( s_mem[my_idx-1], s_mem[2*blockDim.x-1] );

    // Make sure all threads have read their results before pushing forward.
    __syncthreads();

    // Return my partial sum and the total.
    return results;
}
```



```

__global__ void scan_spine_kernel( int n, int *spine )
{
    extern __shared__ int s_mem[]; // Size is 2*blockDim.x

    // Load my item.
    int item = threadIdx.x < n ? spine[threadIdx.x] : 0;

    // Compute the scan on the spine.

    // TODO: fill-in that section of the code
    int2 rank_total = scan_block(s_mem, item);

    // Store the partial sums.
    if( threadIdx.x < n )
        spine[threadIdx.x] = rank_total.x;
}

__global__ void scan_kernel( int n, const int *in_buffer, int *out_buffer,
const int2 *block_ranges, const int *spine )
{
    extern __shared__ int s_mem[]; // Size is 2*blockDim.x

    // The range of data to work with.
    int2 range = block_ranges[blockIdx.x];

    // My local sum.
    int my_sum = spine[blockIdx.x];

    // Compute the scan per blocks.

    // TODO: fill-in that section of the code
    for(int i=range.x; i<range.y; i+=blockDim.x)
    {
        int2 rank_total = scan_block(s_mem, in_buffer[threadIdx.x+i]);
        out_buffer[threadIdx.x+i+1] = my_sum + rank_total.y;
    }
}

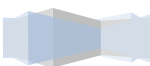
int scan_on_gpu( int n, const int *in_device, int *out_device, int
block_dim )
{
    // Compute the size of the grid.
    const int grid_dim = std::min( block_dim, (n + block_dim-1) /
block_dim );
    const int num_threads = block_dim * grid_dim;

    // Compute the number of elements per block.
    const int elements_per_block = block_dim * ((n + num_threads - 1) /
num_threads);

    // Allocate memory for temporary buffers.
    int *spine = NULL;
    int2 *block_ranges = NULL;

    CUDA_SAFE_CALL( cudaMalloc( (void **) &spine, block_dim *
sizeof(int) ) );
    CUDA_SAFE_CALL( cudaMalloc( (void **) &block_ranges, grid_dim *
sizeof(int2) ) );

```



```

// Compute the ranges for the blocks.
int sum = 0;
int2 *block_ranges_on_host = new int2[grid_dim];
for( int block_idx = 0 ; block_idx < grid_dim ; ++block_idx )
{
    block_ranges_on_host[block_idx].x = sum;
    block_ranges_on_host[block_idx].y = std::min( sum +=
elements_per_block, n );
}
CUDA_SAFE_CALL( cudaMemcpy( block_ranges, block_ranges_on_host, grid_dim
* sizeof(int2), cudaMemcpyHostToDevice ) );
delete[] block_ranges_on_host;

// First round: Compute a partial sum for all blocks.
reduce_kernel<<<grid_dim, block_dim, block_dim*sizeof(int)>>>( n,
in_device, spine, block_ranges );
CUDA_SAFE_CALL( cudaGetLastError() );

// Second round: Scan the spine.
scan_spine_kernel<<<1, block_dim, 2*block_dim*sizeof(int)>>>( grid_dim,
spine );
CUDA_SAFE_CALL( cudaGetLastError() );

// Second round: Scan the complete array.
scan_kernel<<<grid_dim, block_dim, 2*block_dim*sizeof(int)>>>( n,
in_device, out_device, block_ranges, spine );
CUDA_SAFE_CALL( cudaGetLastError() );

// Free temporary memory.
CUDA_SAFE_CALL( cudaFree( block_ranges ) );
CUDA_SAFE_CALL( cudaFree( spine ) );

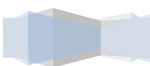
return 0;
}

```

On peut constater que :

- la première étape *reduce_kernel()* correspond au même kernel présenté en première partie ;
- la fonction *scan_block()* reprend le scan expliqué juste au-dessus (construction du double tableau + scan), c'est la fonction centrale ;
- la fonction *scan_spine_kernel()* correspond à la deuxième étape ;
- la fonction *scan_kernel()* correspond à la troisième étape.

Comme pour la réduction, il est possible d'optimiser le scan en utilisant les warps.



Conclusion

Dans le cas d'une réduction, l'utilisation du calcul sur GPU, a permis de diminuer la complexité de l'algorithme, représentant un gain de performance énorme.

Dans le cas du scan, même si la complexité n'a pas diminué, le temps d'exécution, quant à lui, l'a fait.

Quand les calculs sont aussi répétitifs que du traitement pixel par pixel (ou par exemple un bruteforce), utiliser le calcul parallèle sur carte graphique est une bonne option pour augmenter ses performances. Cependant, la limite reste le fait qu'il n'y a qu'une seule instruction identique pour chaque processeur.

