Algorithmes parallèles et Cuda

Gilles Lebrun

gilles.lebrun@ensicaen.fr

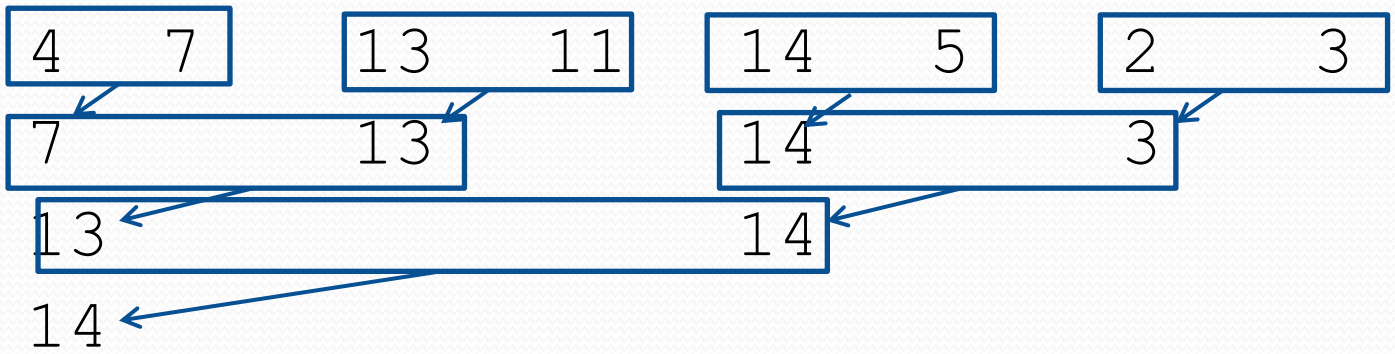
Opérations de réduction

- Exemples de réduction :
  - La recherche de la valeur maximum dans un tableau est une opération de réduction.
  - La somme des valeurs dans un tableau est également une opération de réduction
- A partir d'un ensemble de valeurs, une seule valeur doit être calculé (variable de réduction).
- L'algorithme séquentiel pour obtenir ce type de résultat n'est pas directement parallèlisable.

```
float sequentiel_max(float* vals, int nbr){
  int i; float vmax = vals[0];
  for (i = 1; i < nbr; i++)
    if (vmax < vals[i]) vmax = vals[i];
  return vmax;
}
```

Réduction parallèle (op max)

- Le principe est de réaliser une sélection du maximum en traitant deux par deux les éléments.
- A chaque étape, le nombre d'éléments à comparer est divisé par deux. Ex :



- Il y a convergence vers la solution à partir de $\log_2(n)$ étapes.

Traduction en Cuda de la réduction parallèle pour l'opérateur max

```
__global__ void maxKernel(float* vals, float* vmax, int nbr){
  int id = threadIdx.x + blockDim.x * blockIdx.x*2;
  int tid = threadIdx.x;
  for (unsigned int s = blockDim.x; s > 0; s >>= 1){
    if (tid < s){
      int id2 = id + s;
      if (id2 < nbr){
        vals[id] = max(vals[id], vals[id2]);
      }
    }
    __syncthreads();
  }
  if (tid == 0){
    vmax[blockIdx.x] = vals[id];
  }
}
```

Réflexion sur l'efficacité

- Efficacité avec 1024^2 éléments et un GPU avec 1024 cœurs.
- Nombre d'étape par SM 10
- Nombre de données traitées en // à l'étape 1 : 2048
- Nombre de blocks par étape : 512
- Nombre total de blocks à traiter : 512*10 = 5120
- Facteur d'accélération : 204
  - si 1 op CPU = 1 op GPU au niveau temps
- Overhead à ne pas négliger :
  - Transfert CPU -> GPU et GPU -> CPU
  - Cout de la synchronisation entre deux étapes
  - Fréquence CPU généralement plus rapide que fréquence GPU
  - Terminer le traitement en séquentiel des 512 maximums partielles

Maximum global parmi les maximums des blocks

```
float* host_vmaxs = (float*)malloc(nbblock*sizeof(float));
cudaStatus = cudaMalloc((void**)&dev_res, nbblock*sizeof(float));
cudaStatus = cudaMemcpy(dev_vals, host_vals, N * sizeof(float),
                        cudaMemcpyHostToDevice);
maxKernel<<< nbblock, Nthread >>>(dev_vals, dev_res, N);
cudaStatus = cudaMemcpy(host_vmaxs, dev_res, nbblock *
sizeof(float), cudaMemcpyDeviceToHost);
float vres = sequentiel_max(host_vmaxs, nbblock);
return res;
```

- Reamarque: Une autre solution pour calculer le résultat final des différents blocks est d'exploiter une instruction atomic, mais Cuda ne propose pas de fonction *atomicMax* pour les floats (seulement pour les entiers).

Exploitation du cache (1/2)

- L'exploitation du cache d'un GPU permet une augmentation des performances de la fonction kernel

```
__global__ void maxKernel_share(float* vals, float* vmax, int nbr){
  int id = threadIdx.x + blockDim.x * blockIdx.x;
  int tid = threadIdx.x;
  float __shared__ vals_share[Nthread*2];
  // Chaque thread recopie deux données dans la mémoire cache
  if (id*2+1 < nbr){
    vals_share[tid*2] = vals[id*2];
    vals_share[tid*2+1] = vals[id*2+1];
  } else {
    vals_share[tid*2+1] = (-FLT_MAX);
    if (id*2<nbr) vals_share[tid*2] = vals[id*2];
    else vals_share[tid*2] = (-FLT_MAX);
  }
  __syncthreads(); // Attendre la totalité de la copie dans le cache
```

Exploitation du cache (1/2)

```
// les étapes de réduction sont réalisées en mémoire partagée
for (unsigned int s = blockDim.x ; s > 0; s >>= 1){
   if (tid < s){
     int tid2 = tid + s;
     vals_share[tid] = max(vals_share[tid],
                           vals_share[tid2]);
   }
   __syncthreads();
 }
 // Puis le résultat du maximum est recopié en mémoire globale
 if (tid == 0){
   vmax[blockIdx.x] = vals[id];
 }
}
```

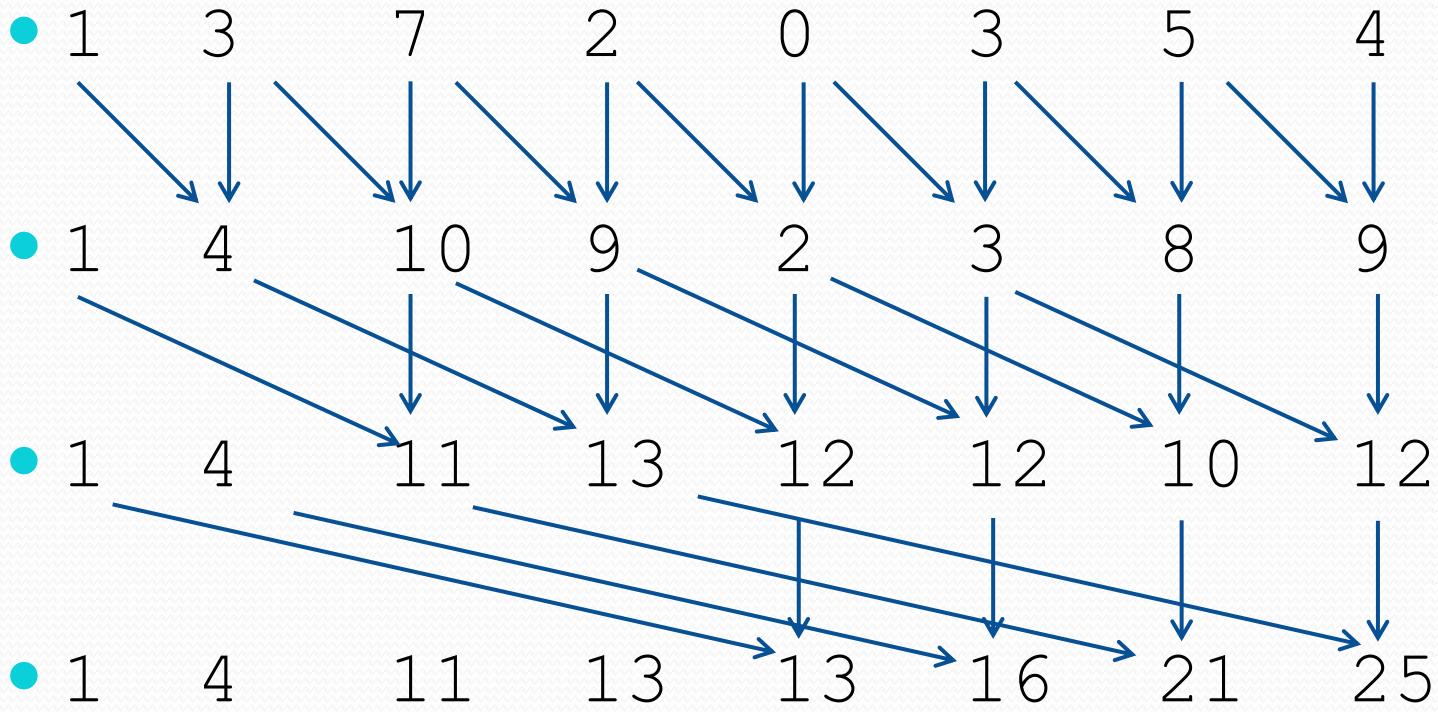
L'algorithme scan

- La version séquentielle (inclusive) de scan est la suivante :

```
void sequentiel_scan_add(float* vin, float* vout, int nbr){
  vout[0] = vint[0];
  for (i = 1; i < nbr; i++){
    vout[i] = vout[i-1] + vin[i];
  }
}
```

- `vin = {1,3,7,2,0,3,5,4} ->`
  `vout = {1,4,11,13,13,16,21,25}`
- L'algorithme séquentiel n'est pas parallèlisable directement
- Remarque : vout[n] = $\sum_{i=0}^{n}$ vin[i]
- Version exclusive : vout[n] = $\sum_{i=0}^{n}$ vin[i − 1] avec vin[-1]=0

Scan parallèle (Hillis Steele - 1986)



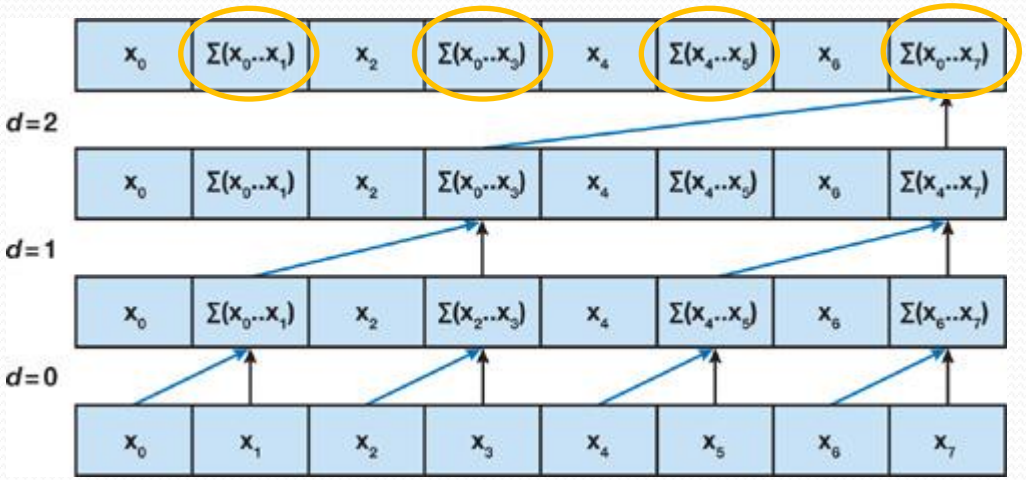
- $Log_2(n)$ étapes et $nlog_2(n)$ opérations

Version Cuda de scan (Hillis, Steele)

```
// Version simplifiée ne prenant en compte qu'un block
 __global__ void scan_add_kernel(float *vin, float *vout, int n)
  extern __shared__ float temp[];
  int thid = threadIdx.x;
  temp[thid] = vin[thid];
  int pout = 0, pin = 1;
  __syncthreads();
  for (int offset = 1; offset < n; offset *= 2)
  {
    pout = 1 - pout; // swap double buffer indices
    pin = 1 - pout;
    if (thid >= offset)
      temp[pout*n+thid] += temp[pin*n+thid - offset];
    else
      temp[pout*n+thid] = temp[pin*n+thid];
    __syncthreads();
  }
  vout[thid] = temp[thid1];
```

Scan parallèle (Blelloch) (1/2)

- Calcul en deux parties :

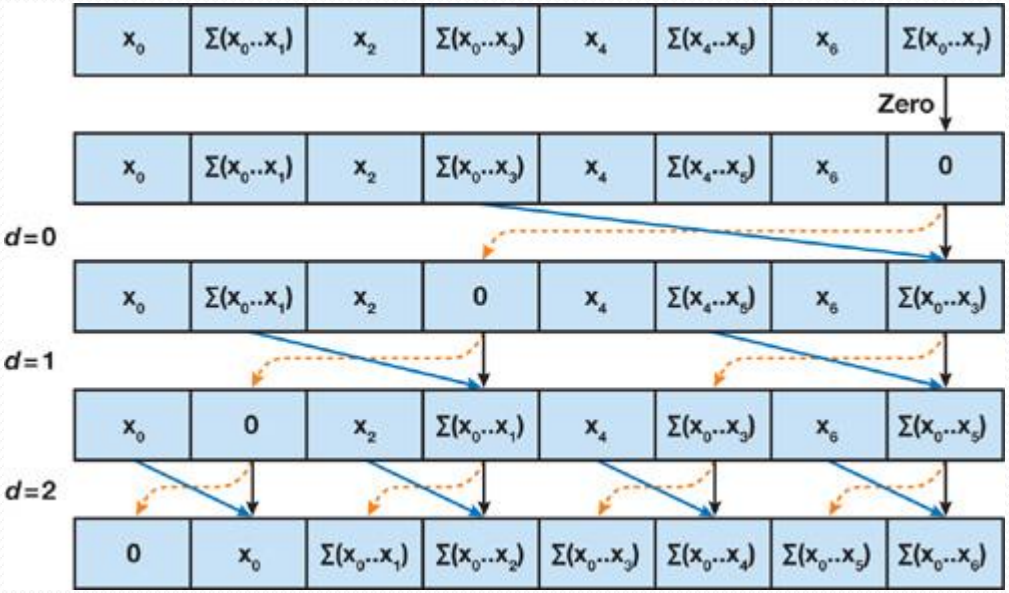


Code Cuda (1/2)

```
__global__ void  void scan_add_kernel(float *vin, float *vout, int nbr)
{
  extern __shared__ float temp[];
  int thid = threadIdx.x;
  int offset = 1;
  temp[2*thid] = vin[2*thid];
  temp[2*thid+1] = vin[2*thid+1];
  for (int d = nbr>>1; d > 0; d >>= 1){
    __syncthreads();
    if (thid < d)
    {
      int ai = offset(2*thid+1)-1;
      int bi = offset(2*thid+2)-1;
      temp[bi] += temp[ai];
    }
    offset *=2;
}
if (thid == 0) temp[nbr-1] = 0;
```

Scan parallèle (Blelloch) (2/2)

- Remarque : c'est la version exclusive qui est calculée



Code Cuda (2/2)

```
  for (int d = 1; d < n; d *= 2)
  {
    offset >>= 1;
    __syncthreads();
    if (thid < d){
      int ai = offset*(2*thid+1)-1;
      int bi = offset*(2*thid+2)-1;
     float t = temp[ai];
     temp[ai] = temp[bi];
     temp[bi] += t;
    }
  }
  __syncthreads();
  vout[2*thid] = temp[2*thid];
  vout[2*thid+1] = temp[2*thid+1];
}
```