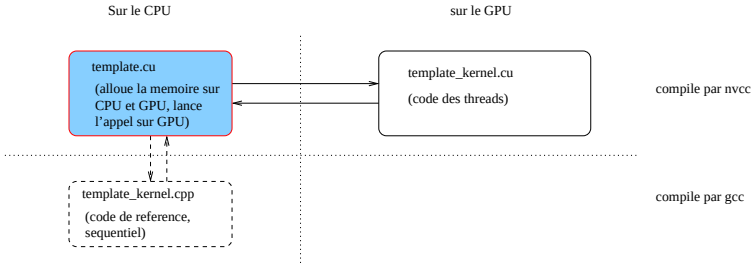
Parallélisme

Eric Goubault
Commissariat à l'Energie Atomique & Chaire Ecole Polytechnique/Thalès
Saclay

Principe de l'architecture du code CUDA

`template` fourni:



TEMPLATE.CU

```
// includes, system
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

// includes, project
#include <cutil.h>

// includes, kernels
#include <template_kernel.cu>
```


TEMPLATE.CU

Déclarations (suite)

```
void runTest( int argc, char** argv);

extern "C"
void computeGold( float* reference, float* idata,
                  const unsigned int len);
```

TEMPLATE.CU

(main)

```
int main( int argc, char** argv) {
    runTest( argc, argv);
```

Lance le programme (runTest) (qui teste la connection au GPU, en passant...)...

```
    CUT_EXIT(argc, argv); }
```

puis affiche à l'écran "Press ENTER to exit..." (macro en fait, cf. common/cutil_readme.txt)

TEMPLATE.CU

```
void runTest( int argc, char** argv) {
    CUT_DEVICE_INIT(argc, argv);
```

De même macro... qui teste la présence d'une carte NVIDIA compatible CUDA

TEMPLATE.CU

```
    unsigned int timer = 0;
    CUT_SAFE_CALL( cutCreateTimer( &timer));
    CUT_SAFE_CALL( cutStartTimer( timer));
```

Création d'un timer (pour mesurer le temps d'exécution) - utilisation de macros qui testent le code retour des fonctions

TEMPLATE.CU

```
    unsigned int num_threads = 32;
    unsigned int mem_size = sizeof( float) * num_threads;

    float* h_idata = (float*) malloc( mem_size);

    for( unsigned int i = 0; i < num_threads; ++i) {
        h_idata[i] = (float) i; }
```

Initialisation, et initialisation des données dans la mémoire CPU (h_idata)

TEMPLATE.CU

```
// allocate device memory
float* d_idata;
CUDA_SAFE_CALL( cudaMalloc( (void**) &d_idata, mem_size));
// copy host memory to device
CUDA_SAFE_CALL( cudaMemcpy( d_idata, h_idata, mem_size,
                            cudaMemcpyHostToDevice) );

// allocate device memory for result
float* d_odata;
CUDA_SAFE_CALL( cudaMalloc( (void**) &d_odata, mem_size));
```

Allocation mémoire et recopie tableau données CPU (`h_idata`) vers données en mémoire globale GPU (`d_idata`)

TEMPLATE.CU

```
dim3  grid( 1, 1, 1);
```

On définit une grille de 1 bloc (sera exécuté sur un seul multi-processeur)

```
dim3  threads( num_threads, 1, 1);
```

Sur ce multiprocesseur, `num_threads` (32) threads vont être exécutés (en général, selon la carte graphique utilisé, 8 threads à tout moment seront exécutés sur chacun des 8 coeurs du multi-processeur)

TEMPLATE.CU

```
testKernel<<< grid, threads, mem_size >>>
                                ( d_idata, d_odata);
```

On exécute la fonction `testKernel` sur le GPU, avec la répartition en grille de blocs et de threads par bloc définis précédemment (avec pour données, celles données en argument, recopiées sur le GPU)

TEMPLATE.CU

```
// check if kernel execution generated and error
CUT_CHECK_ERROR("Kernel execution failed");
```

Utilisation d'une macro pour vérifier le dernier code de retour (de `testKernel`)

```
float* h_odata = (float*) malloc( mem_size);
CUDA_SAFE_CALL( cudaMemcpy( h_odata, d_odata,
                            sizeof( float) * num_thre
                            cudaMemcpyDeviceToHost) )
```

On récupère les données résultat du calcul sur GPU (`d_odata`) par le CPU (dans `h_odata`)

TEMPLATE.CU

```
    CUT_SAFE_CALL( cutStopTimer( timer));
    printf( "Processing time: %f (ms)\n",
            cutGetTimerValue( timer));
    CUT_SAFE_CALL( cutDeleteTimer( timer));
```

Calcul du temps de calcul sur GPU (arrêt du timer)

```
    float* reference = (float*) malloc( mem_size);
    computeGold( reference, h_idata, num_threads);
```

Calcul de la solution séquentielle

TEMPLATE.CU

```
if(cutCheckCmdLineFlag( argc, (const char**) argv,
                        "regression")) {
  CUT_SAFE_CALL(cutWriteFilef("./data/regression.dat",
                              h_odata, num_threads, 0.0)); }
else {
  CUTBoolean res = cutComparef( reference, h_odata, num_threads);
  printf( "Test %s\n", (1 == res) ? "PASSED" : "FAILED");    }
```

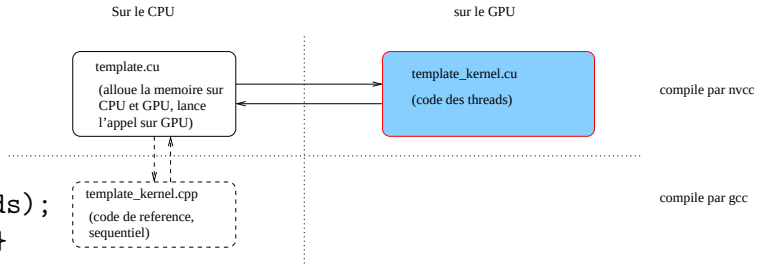
On vérifie que les résultats parallèles et séquentiels sont les mêmes

TEMPLATE.CU

```
    free( h_idata);
    free( h_odata);
    free( reference);
    CUDA_SAFE_CALL(cudaFree(d_idata));
    CUDA_SAFE_CALL(cudaFree(d_odata));
}
```

On libère la mémoire, sur le CPU, et sur le GPU

Principe de l'architecture du code CUDA [suite]



TEMPLATE_KERNEL.CU

```
#include <stdio.h>
#define SDATA( index)      CUT_BANK_CHECKER(sdata, index)
```

Utilisation d'une macro permettant de déterminer en mode émulation (`nvcc -deviceemu ...`) s'il y a conflit potentiel d'accès en mémoire partagée globale (modèle EREW...).

Peut être une source importante de baisse de performance.

Exemple de "bank conflict"

Code standard:

```
__global__ void test(int *gpA)
{
   __shared__ int sa[16];
   sa[0]=3; // bank conflict if blocksize > 1
   gpA[0]=sa[0]; // bank conflict again
}
```


Code instrumenté

```
__global__ void test(int *gpA)
{
__shared__ int sa[16];
CUT_BANK_CHECKER(sa,0)=3; // bank conflict if blocksize >
gpA[0]=CUT_BANK_CHECKER(sa,0); // bank conflict again
}
```

Macro fournie par la libraire CUT (cf. `bank_checker.h`)

Compilation et exécution

```
> nvcc -deviceemu...
...
```

Un appel à `cutCheckBankAccess` imprime les conflits

Suite du code `template_kernel.cu`

```
__global__ void
testKernel( float* g_idata, float* g_odata) {
```

`__global__`: fonction appelable depuis le CPU ou le GPU (et exécutée sur le GPU)

```
  extern  __shared__  float sdata[];
```

En mémoire partagée, taille déterminée par CPU

Suite du code `template_kernel.cu`

```
const unsigned int tid = threadIdx.x;
const unsigned int num_threads = blockDim.x;
SDATA(tid) = g_idata[tid];
```

Grille d'un seul bloc, et organisation uni-dimensionnelle des threads, d'où `tid=threadIdx.x`

```
__syncthreads();
```

Synchronisation de tous les threads (bloquant)

Suite du code `template_kernel.cu`

```
  SDATA(tid) = (float) num_threads * SDATA( tid);
```

Calcul (exemple)

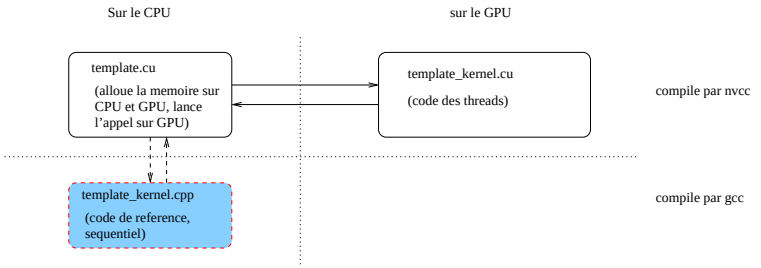
```
  __syncthreads();
```

Synchronisation de tous les threads (bloquant)

```
  g_odata[tid] = SDATA(tid);
}
```

Ecriture en mémoire globale

Principe de l'architecture du code CUDA [suite]



TEMPLATE_GOLD.CPP

(code séquentiel de référence)

```
extern "C"
void computeGold( float* reference, float* idata,
                  const unsigned int len);
void computeGold( float* reference, float* idata,
                  const unsigned int len)
{   const float f_len = static_cast<float>( len);
    for( unsigned int i = 0; i < len; ++i) {
        reference[i] = idata[i] * f_len;   } }
```


Remarques... sur les performances

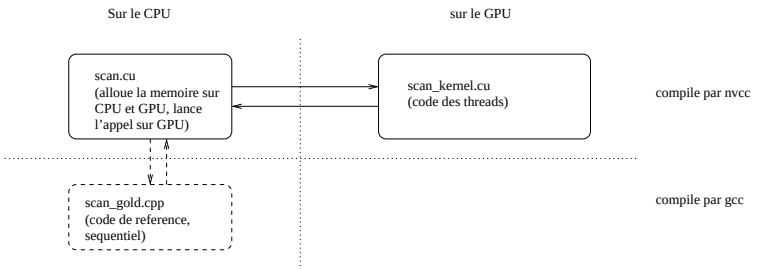
- Ne vous laissez pas décourager par de piètres performances pour une première version de vos programmes
- Essayez de comprendre les raisons:
  - bank conflict
  - transferts de données trop importants entre CPU et GPU pour un calcul trop court
  - trop de passage par la mémoire globale du GPU, et pas assez par la mémoire partagée au niveau des multi-processeurs
- Utilisez la librairie CUT, le "occupancy calculator" (feuille excel - cf. page `nvidia.com/cuda`) et éventuellement un profiler...

Application: scan (naif) CUDA

Limitée à des tableaux de 512 éléments (nombre de threads maxi sur un multi-processeur) - cf. "projects/scan" pour la doc et les codes

```
for d := 1 to log2n do
    forall k in parallel do
        if k ≥ 2^d then
            x[out][k] := x[in][k − 2^(d-1)] + x[in][k]
        else
            x[out][k] := x[in][k]
    swap(in, out)
```


Principe de l'architecture du code CUDA



Implémentation CUDA (naive)

(scan_kernel.cu)

```
__global__ void scan_naive(float *g_odata, float *g_idata, int n)
{
// Dynamically allocated shared memory for scan kernels
extern  __shared__  float temp[];
int thid = threadIdx.x;
int pout = 0;
int pin = 1;
// Cache the computational window in shared memory
temp[pout*n + thid] = (thid > 0) ? g_idata[thid-1] : 0;
```


Implémentation CUDA

```
for (int offset = 1; offset < n; offset *= 2)
{
  pout = 1 - pout;
  pin  = 1 - pout;
  __syncthreads();
  temp[pout*n+thid] = temp[pin*n+thid];
  if (thid >= offset)
    temp[pout*n+thid] += temp[pin*n+thid - offset];
  }
  __syncthreads();
  g_odata[thid] = temp[pout*n+thid];
}
```


Implémentation CUDA

(scan.cu)

```
int main( int argc, char** argv)
{    runTest( argc, argv);
     CUT_EXIT(argc, argv); }
void runTest( int argc, char** argv)
{
CUT_DEVICE_INIT(argc, argv);...
// initialize the input data on the host to be integer va
// between 0 and 1000
for( unsigned int i = 0; i < num_elements; ++i)
  h_data[i] = floorf(1000*(rand()/(float)RAND_MAX));
// compute reference solution
float* reference = (float*) malloc( mem_size);
computeGold( reference, h_data, num_elements);
```


Implémentation CUDA

```
// allocate device memory input and output arrays
float* d_idata;
float* d_odata[3];
CUDA_SAFE_CALL( cudaMalloc( (void**) &d_idata, mem_size))
CUDA_SAFE_CALL( cudaMalloc( (void**) &(d_odata[0]), mem_s
CUDA_SAFE_CALL( cudaMalloc( (void**) &(d_odata[1]), mem_s
CUDA_SAFE_CALL( cudaMalloc( (void**) &(d_odata[2]), mem_s
// copy host memory to device input array
CUDA_SAFE_CALL( cudaMemcpy( d_idata, h_data, mem_size, cu
```

Implémentation CUDA

```
// setup execution parameters
// Note that these scans only support a single thread-block worth of data,
// but we invoke them here on many blocks so that we can accurately compare
// performance
#ifndef __DEVICE_EMULATION__
    dim3  grid(256, 1, 1);
#else
    dim3  grid(1, 1, 1); // only one run block in device emu mode or it will be too slow
#endif
    dim3  threads(num_threads*2, 1, 1);

// make sure there are no CUDA errors before we start
CUT_CHECK_ERROR("Kernel execution failed");
```


Implémentation CUDA

```
unsigned int numIterations = 100;
for (unsigned int i = 0; i < numIterations; ++i)
  {
  scan_naive<<< grid, threads, 2 * shared_mem_size >>>
            (d_odata[0], d_idata, num_elements);
  }
cudaThreadSynchronize();
...
```