

Parallélisme

COURS 4 - INTRODUCTION À CUDA

ERIC GOUBAULT
 COMMISSARIAT À L'ÉNERGIE ATOMIQUE & CHAIRE ECOLE
 POLYTECHNIQUE/THALÈS
 SACLAY

LE 3 FÉVRIER 2010

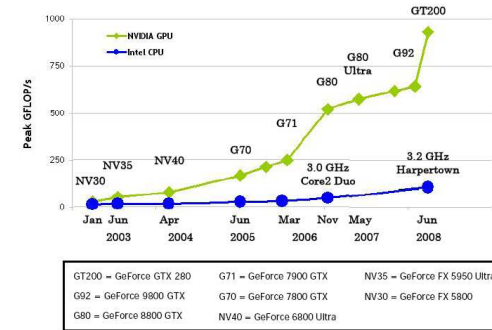
1

PLAN DU COURS

- CUDA et architecture NVIDIA
- L'abstraction logique de l'architecture proposée par CUDA
- C versus JAVA...
- API CUDA
- Un exemple: addition de matrices
- Revenons à l'architecture...pour optimiser...
- Un exemple: transposition de matrices
- Pour aller plus loin...

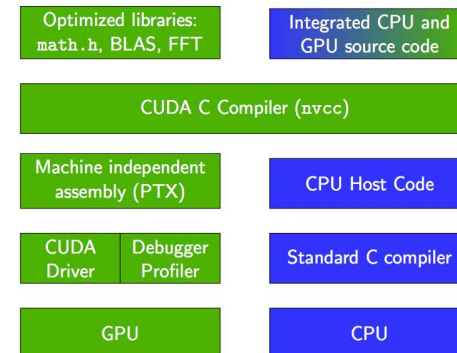
CUDA?

- “Compute Unified Device Architecture”
- Programmation massivement parallèle en C sur cartes NVIDIA
- Tirer parti de la puissance des cartes graphiques:

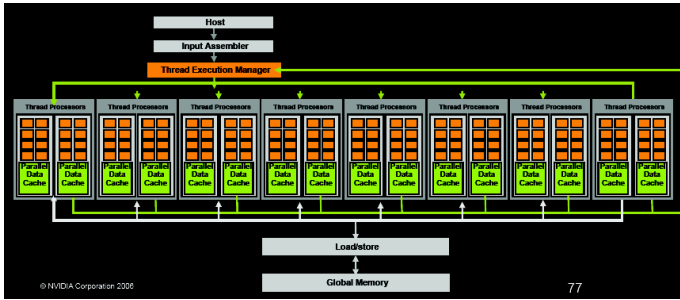


3

CUDA SDK



ARCHITECTURE PHYSIQUE

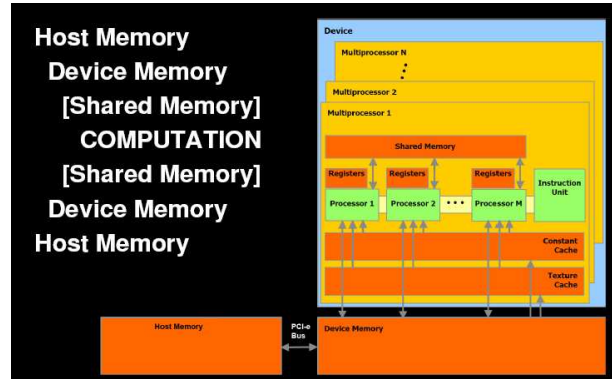


(ici 128 threads procs. pour 8 multi-procs de 16 coeurs) L'hôte peut charger des données sur le GPU, et calculer en parallèle avec le GPU

ARCHITECTURE PHYSIQUE

- Organisés en multiprocesseurs (ex. GeForce GTX 260: 24 multi-proc de 8 coeurs=192 coeurs, à 1.242GHz)
- registres 32 bits par multi-proc.
- mémoire partagées par multi-proc.
- une mémoire (“constante”) à lecture seule (ainsi qu’un cache de textures à lecture seule)

ARCHITECTURE PHYSIQUE



MODÈLE DE PROGRAMMATION - UN PEU DE VOCABULAIRE

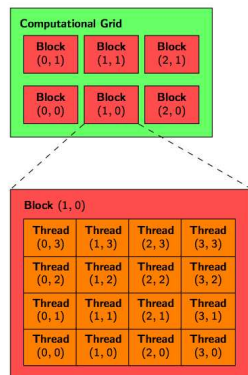
- la carte graphique=“GPU” ou “device” est utilisé comme “co-processeur” de calcul pour le processeur de la machine hôte, le PC typiquement ou “host” ou “CPU”
 - la mémoire du CPU est distincte de celle du GPU
 - mais on peut faire des recopies de l’un vers l’autre (couteux)
- une fonction calculée sur le device est appelée “kernel” (noyau)
- le kernel est dupliqué sur le GPU comme un ensemble de threads
 - cet ensemble de threads est organisé de façon logique en une “grid”
- chaque clône du kernel connaît sa position dans la grid et peut calculer la fonction définie par le kernel sur différentes données
- cette grid est mappée physiquement sur l’architecture de la carte au “runtime”

ABSTRACTION LOGIQUE - GRID

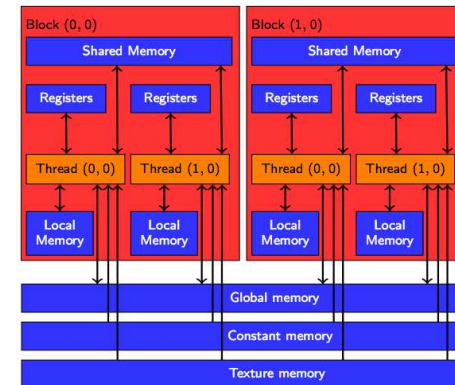
- une grid est un tableau 1D, 2D ou 3D de “thread blocks” - au maximum 65536 blocks par dimension (en pratique, 2D...)
- chaque thread block est un tableau 1D, 2D ou 3D de “threads”, chacun exécutant un clône (instance) du kernel - au maximum 512 threads par block (en général)
- chaque block a un unique **blockId**
- chaque thread a un unique **threadId** (dans un block donné)

9

GRID



MODÈLE MÉMOIRE



11

MODÈLE MÉMOIRE

Suit la hiérarchie (logique) de la grid:

- Mémoire globale (du device): la plus lente (400 à 600 cycles de latence!), accessible (lecture/écriture) à toute la grid
 - Possibilité d'optimiser cela en “amalgamant” les accès
- Mémoire partagée: rapide mais limitée (16Ko par multiprocesseur), accessible (lecture/écriture) à tout un block - qualificatif **__shared__**

MODÈLE MÉMOIRE

- Registres (16384 par multiprocesseur): rapide mais très limitée, accessible (lecture/écriture) à un thread
- Mémoire locale: lente (200 a 300 cycles!) et limitée, accessible (lecture/écriture) - gérée automatiquement lors de la compilation (quand structures ou tableaux trop gros pour être en registre)

En plus de cela (quasi pas traité ici), mémoire constante et texture: rapide, accessible (en lecture uniquement depuis le GPU, lecture/écriture depuis le CPU) à toute la grid. Mémoire constante très petite (~8 à 64 K)

13

GESTION MÉMOIRE C/JAVA

Un peu de C d'abord...:

```
float *c; /* result */
void add_matrix(float *a, float *b, int N) {
    for (int i=0; i<N; i++)
        for (int j=0; j<N; j++)
            c[i+j*N]=a[i+j*N]+b[i+j*N]; }
int main(int argc, char **argv) {
    float *x, *y;
    int N=16;
    x=(float *) malloc(N*sizeof(float));
    y=(float *) malloc(N*sizeof(float));
    c=(float *) malloc(N*sizeof(float));
    (...)
    add_matrix(a, b, N); }
```

GESTION MÉMOIRE

```
public class matrix {
    float c[]; /* result */

    public void add_matrix(
    public void main(String args) {
        float x[], y[], z[];
        int N=16;
        x=new float[N];
        y=new float[N];
        c=new float[N];
        (...)
        add_matrix(x, y, N);
    }
}
```

15

GESTION MÉMOIRE EN C SUR LE CPU/GPU

- Pointeurs...
- Fonctions d'allocation C `malloc/free`; ainsi que `new/delete` avec `nvcc`
- tableaux 1D, 2D, 3D... arithmétique des pointeurs en C
- Fonctions spécifiques d'allocation mémoire sous CUDA: `cudaMalloc`, `cudaFree`, fonctions de copie `cudaMemcpy` mais aussi fonctions sur des tableaux `cudaMallocPitch` (2D), `cudaMalloc3D` (3D) et `cudaMemcpy2D`, `cudaMemcpy3D`

POINTEURS/ALLOCATION C

- Notion d'adresse mémoire
- Déclaration `float *x`; "pointeur" sur `x`
- `float y, *x`; puis `x=&y`; (allocation statique)
- `float *x`; puis `x=(float *) malloc(sizeof(float))`; (allocation dynamique)

17

TABLEAUX C

- Vecteur: `float x = (float *) malloc(N*sizeof(float))`;
vecteur à `N` entrées (bloc contigu de `N` mots de 32 bits)
- Matrice: `float *x = (float *) malloc(N*M*sizeof(float))`;
matrice de dimension `N*M`
 - on récupère $x_{i,j}$ par `x[i*M+j]`
- Autre méthode: `float **x = (float **) malloc(N*sizeof(float*))`; puis:

```
for (i=0; i<N; i++)
    x[i] = (float *) malloc(M*sizeof(float));
```

Le tableau est ainsi implémenté comme `N` pointeurs sur `N` vecteurs de taille `M` (chacun un bloc contigu de mémoire)
- Ainsi de suite en dimension supérieure...

Remarque: en C si `float *x`; alors `x[i]` est équivalent à `*(x+i)`

QUELQUES DIFFÉRENCES SYNTAXIQUES C/JAVA

- `System.out.println("Le resultat est: "+x); printf("Le resultat est: %d\n",x)` (si `x` est de type `int`, sinon `%f` si de type `float` etc.)
- Voir la définition de `main`
- ...

19

TABLEAUX CUDA

- `cudaMalloc(void **x,int y)`; alloue `y` octets dans la mémoire du GPU et renvoie un pointeur sur l'adresse ainsi allouée
- Exemple: allocation d'un tableau de 256 éléments flottants `float* devPtr`; `cudaMalloc((void * *) & devPtr, 256 * sizeof(float))`;

TABLEAUX CUDA

- Egalement: `cudaMallocPitch` pour un tableau 2D
- Exemple: allocation et l'utilisation d'un tableau 2D.

```
float* devPtr;  
int pitch;  
cudaMallocPitch((void**)&devPtr,&pitch,&width,&height);
```

- On accède alors aux éléments du tableau 2D correspondant par:

```
for (int r = 0 ; r < height ; ++r) {  
    float* row = (float *) ( (char *) devPtr + r * pitch);  
    for (int c = 0 ; c < width ; ++c) {  
        float element = row[c]; } } }
```

(similaire pour un tableau 3D)

21

COPIE DE TABLEAUX (CUDA)

- `cudaMemcpy(void *dst,const void *src,size_t count,enum cudaMemcpyKind kind)`
 - Copie de l'adresse `src` vers l'adresse `dst`
 - `count` octets
 - dans la direction `cudaMemcpyHostToHost`, `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost` ou `cudaMemcpyDeviceToDevice`
- `cudaMemcpy` peut renvoyer une erreur
 - `cudaSuccess`
 - `cudaErrorInvalidValue`
 - `cudaErrorInvalidDevicePointer`
 - `cudaErrorInvalidMemcpyDirection`

EGALEMENT...

- `cudaMemcpyToArray(struct cudaArray * dstArray,size_t dstX,size_t dstY,const void * src,size_t count,enum cudaMemcpyKind kind)`
- `cudaMemcpy2D(void * dst,size_t dpitch,const void * src,size_t spitch,size_t width,size_t height,enum cudaMemcpyKind kind)`
- Versions asynchrones
- etc. (voir doc)

23

API SUR LE GPU

- Qualificateurs de types de fonctions:
 - `__device__ f(...)`: la fonction `f` uniquement appellable/exécuté sur le GPU (device)
 - `__host__ g(...)`: la fonction `g` uniquement appellable/exécuté sur le CPU (host)
 - `__global__ h(...)`: la fonction `h` exécutable sur le GPU et appellable depuis le CPU (tous les kernels)
- Qualificateurs de types de variables:
 - `__device__ int x;`: `x` est un entier en mémoire globale
 - `__constant__ int x=5;`: `x` est un entier en mémoire constante
 - `__shared__ int x;`: `x` est un entier en mémoire partagée

API SUR LE GPU

- Exécution du kernel `f` et mapping logique sur la grid:
 - `f<<<GridDim, BlockDim>>>(...)`
 - où `GridDim` est de type `dim3`
 - où `BlockDim` est de type `dim3`
- Chaque instance du kernel sait où il est exécuté par:
 - `blockIdx` renvoie un `uint3` indiquant dans quel block on est
 - `threadIdx` renvoie un `uint3` indiquant dans quel thread du block on est
- Dans chaque block, tous les threads exécutant le kernel peuvent se synchroniser (se mettre en attente à un point de rendez-vous) en faisant `__syncthreads()`;

25



- Si `A` est de type `dim3` ou `uint3`
- La différence entre `dim3` et `uint3` est que les composantes non-initialisées d'un `dim3` sont par défaut égales à 1
- Il existe d'autres types vecteurs...
- `A.x`, `A.y` et `A.z` renvoient des `int` donnant les 3 coordonnées
- Il est alors facile de mapper tout tableau n -dimensionnel sur une grille par une simple formule impliquant `blockIdx.x`, `blockIdx.y`, `blockIdx.z` et `threadIdx.x`, `threadIdx.y` et `threadIdx.z`

PREMIER EXEMPLE CUDA: SOMME DE MATRICES

```
const int N = 1024;
const int blocksize = 16;

__global__ void add_matrix
(float* a, float *b, float *c, int N ) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int index = i + j*N;
    if ( i < N && j < N )
        c[index] = a[index] + b[index];
}
```

27

PREMIER EXEMPLE CUDA: SOMME DE MATRICES

```
int main() {
    float *a = new float[N*N];
    float *b = new float[N*N];
    float *c = new float[N*N];
    for ( int i = 0; i < N*N; ++i ) {
        a[i] = 1.0f; b[i] = 3.5f; }
    float *ad, *bd, *cd;
    const int size = N*N*sizeof(float);
    cudaMalloc( (void**)&ad, size );
    cudaMalloc( (void**)&bd, size );
    cudaMalloc( (void**)&cd, size );
    cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
    cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );
}
```

PREMIER EXEMPLE CUDA: SOMME DE MATRICES

```
dim3 dimBlock( blockSize, blockSize );
dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );
cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );
cudaFree( ad );
cudaFree( bd );
cudaFree( cd );
delete[] a;
delete[] b;
delete[] c;
return EXIT_SUCCESS;
}
```

29

COMPILATION

- Les kernels CUDA sont écrits dans des fichiers `.cu`
- Les fonctions et main sur le CPU, dans des fichiers `.c` ou dans le même `.cu`
- Compilateur NVIDIA `nvcc` qui compile les `.c` en utilisant le compilateur C sous-jacent (`gcc` etc.) et les `.cu`

Exemple (spécifique salle TD):

```
nvcc -I/usr/local/cuda/include/
-I/usr/local/cuda/sdk/C/common/inc/
-L/usr/local/cuda/lib/
-L/users/profs/info/goubaul1/sdk/C/lib/
-lcutil matrix.cu
```

(quand on utilise les `timer`)

EXÉCUTION

```
[finlande ~]$ nvcc -I/usr/local/cuda/include...
[finlande ~]$ ./matrix
Processing time on GPU: 0.139920 (ms)
Processing time on CPU: 7.093190 (ms)
```

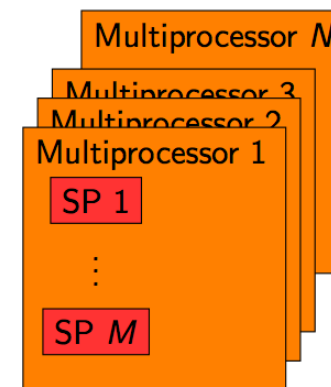
(en utilisant les fonctions de `cutil.h`)

Code sur GPU presque 51 fois plus rapide que sur CPU!

31

RUNTIME

Mapping physique block→multiprocesseurs



RUNTIME

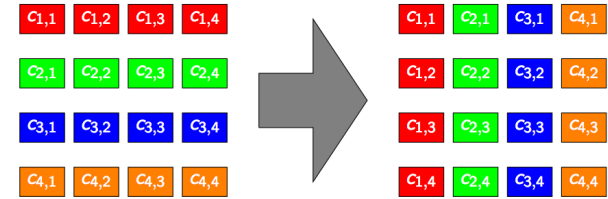
- Un block est exécuté par un seul multiprocesseur
- Chaque block est divisé en groupes de threads (“physiques”) appelés “warps”
- Un warp (en général 32 threads) est exécuté physiquement en parallèle
- Un warp est constitué de threads de `threadIdx` consécutifs et croissants
- L’ordonnanceur de la carte alterne entre les warps

33

EFFICACITÉ/SÉCURITÉ

- Pour être efficace (pour que l’ordonnanceur ait toujours quelque chose à ordonnancer), il faut essayer de définir un nombre de blocks de 2 à 100 fois égal au nombre de multiprocesseurs.
- De même, on essaie de définir plusieurs warps par multiprocesseurs (tirer parti du recouvrement potentiel calcul/accès mémoire)
- Ne pas oublier `__syncthreads()` et `cudaThreadSynchronize()` (au niveau du CPU) pour assurer les fonctions de barrière de synchronisation (resp. attente qu’un kernel soit terminé)

OPTIMISATION - EX.: TRANSPOSITION



35

OPTIMISATION - MÉMOIRE GLOBALE “AMALGAMÉE”

```
__global__ void transpose_naive
(float *out, float *in, int w, int h ) {
    unsigned int xIdx = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int yIdx = blockDim.y * blockIdx.y + threadIdx.y;
    if ( xIdx < w && yIdx < h ) {
        unsigned int idx_in = xIdx + w * yIdx;
        unsigned int idx_out = yIdx + h * xIdx;
        out[idx_out] = in[idx_in];
    }
}
```

PAS RAPIDE...

```
[finlande ~]$ nvcc -I/usr/local/cuda/include ...  
[finlande ~]$ ./transpose  
Processing time (naive) on GPU: 1.007090 (ms)  
Processing time on CPU: 8.726710 (ms)  
Accélération de presque 9 fois avec l'approche naive...
```

37

OPTIMISATION

Mémoire partagée et accès à la mémoire globale “amalgamés”:

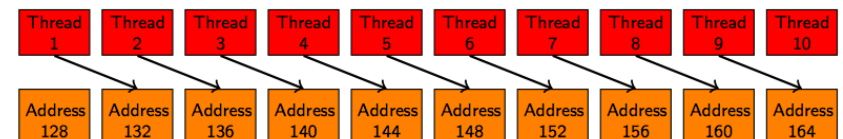
- En général, il est bien meilleur de passer par la mémoire partagée pour des calculs intensifs
- Dans ce cas, on alloue les données initiales dans la mémoire globale du GPU, on recopie les données depuis le CPU...
- Puis on recopie des bouts de ces données dans la mémoire partagée de chaque block (à la fin on recopiera les résultats de la mémoire partagée à la mémoire globale, puis au CPU)

OPTIMISATION

- Point important pour la bonne utilisation de la bande passante mémoire globale vers mémoire partagée: “amalgamation”...
- Il faut faire attention à deux choses dans l’utilisation de la mémoire partagée:
 - Bien utiliser `__syncthreads()`: permet d’attendre que tous les threads d’un même block ont bien lu ou écrit leurs données en mémoire partagée ou globale par exemple, avant de continuer un calcul...
 - Bank conflict...

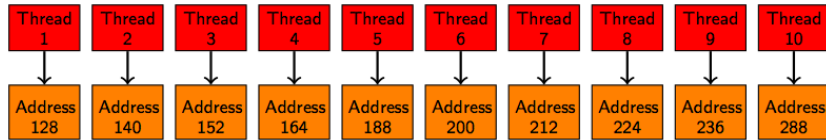
39

EXPLICATION: ACCÈS MÉMOIRE NON-AMALGAMÉ



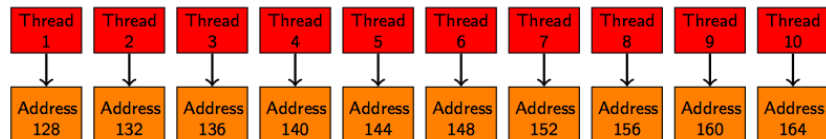
Accès non aligné modulo 16

EXPLICATION: ACCÈS MÉMOIRE NON-AMALGAMÉE



Adresses non “connexes” dans un bloc

CE QU'IL FAUT FAIRE: ACCÈS MÉMOIRE AMALGAMÉE

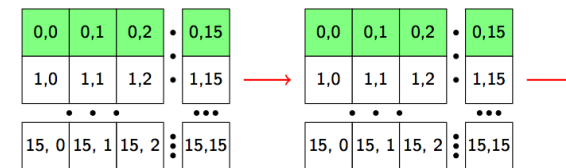


On doit accéder à la mémoire par des accès 8, 16, 32/64 128 bits consécutifs (dans l'ordre des threads pour avant 1.2 - ce n'est pas notre cas ici), dans un bloc mémoire de 32, 64 ou 128 octets; adresse de départ alignée modulo 16

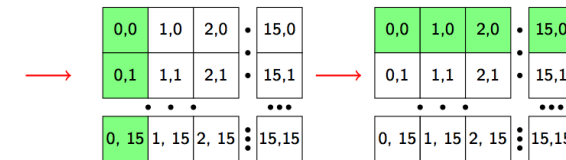
VERSION AMALGAMÉE

- La matrice est partitionnée en sous-blocs carrés
- Un bloc carré est associé à un block (bx,by):
 - Charger le bloc (bx,by) de la mémoire globale vers la mémoire partagée
 - Faire la transposition en mémoire partagée (pas de problème d'amalgamation, juste les “bank conflicts...”) en parallèle sur tous les thread processor
- Ecrire le résultat dans la mémoire globale, par blocs contigus

CE QU'IL FAUT FAIRE POUR LA TRANSPOSITION:



Lecture de la mém. globale ; Ecriture en mémoire partagée



Lire les adresses transposées en SMEM ; Ecrire dans la mém. globale

VERSION AMALGAMÉE

DANS LE MAIN

```
__global__ void transpose
(float *out, float *in, int width, int height ) {
    __shared__ float block[BLOCK_DIM*BLOCK_DIM];
    unsigned int xBlock = blockDim.x * blockIdx.x;
    unsigned int yBlock = blockDim.y * blockIdx.y;
    unsigned int xIndex = xBlock + threadIdx.x;
    unsigned int yIndex = yBlock + threadIdx.y;
    unsigned int index_out, index_transpose;
```

```
N = 1024
blocksize = BLOCK_DIM = 16;

dim3 dimBlock( blocksize, blocksize );
dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );

transpose<<<dimGrid, dimBlock>>>( ad, bd, N, N );
```

45

47

VERSION AMALGAMÉE

```
if ( xIndex < width && yIndex < height ) {
    unsigned int index_in=width*yIndex+ xIndex;
    unsigned int index_block=threadIdx.y*BLOCK_DIM+threadIdx.x;
    block[index_block]=in[index_in];
    index_transpose=threadIdx.x*BLOCK_DIM+threadIdx.y;
    index_out=height*(xBlock+threadIdx.y)+yBlock+threadIdx.x;
}
__syncthreads();
if ( xIndex < width && yIndex < height ) {
    out[index_out]=block[index_transpose];}
}
```

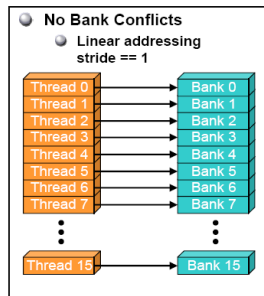
COMPILATION ET EXÉCUTION

```
[finlande ~]$ nvcc -I/usr/local/cuda/include ...
[finlande ~]$ ./transpose
Processing time (naive) on GPU: 1.007090 (ms)
Processing time (coalesced) on GPU: 0.292140 (ms)
Processing time on CPU: 8.726710 (ms)
```

Accélération de presque 9 fois avec l'approche naive...de 30 fois avec l'approche améliorée!

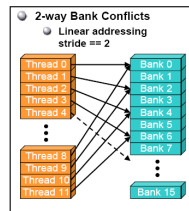
POUR ALLER PLUS LOIN: “BANK CONFLICT”?

- La mémoire d’une machine parallèle à mémoire partagée est généralement partitionnée en “bank” sur lesquelles un seul **read** ou un seul **write** (de mots 32 bits ici) est effectué à tout instant (2 cycles en général)
- Cf. PRAM EREW...



49

“BANK CONFLICT”?



- Cas de conflit:
- Perte de performance: sérialisation des accès mémoire...
- Cas particulier: tous les threads d’un warp lisent la même adresse en même temps...pas de conflit!
- Ex. les cartes G80:
 - ont 16 banks, faits de mots 32 bits contigus
 - Donc # bank=adresse % 16
 - attention à l’alignement des données! (`__ALIGN(X)`)

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
}
```

51

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 librairie CUT (cf. `bank_checker.h`)

COMPILATION ET EXÉCUTION

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

```
...
```

Un appel à `cutCheckBankAccess` imprime les conflits (bibliothèque `cutil`)

53

POUR ALLER PLUS LOIN...

- Les flux; tableaux CUDA généraux
- Les fonctions atomiques `atomicAdd`, `atomicSub`, `atomicInc` et `atomicMin`, `atomicMax`
- Les problèmes de précision numériques, les unités `double/float`
- jCUDA; openCL; RMI+CUDA...
- CUDA profiler, cuda-gdb, Nexus...

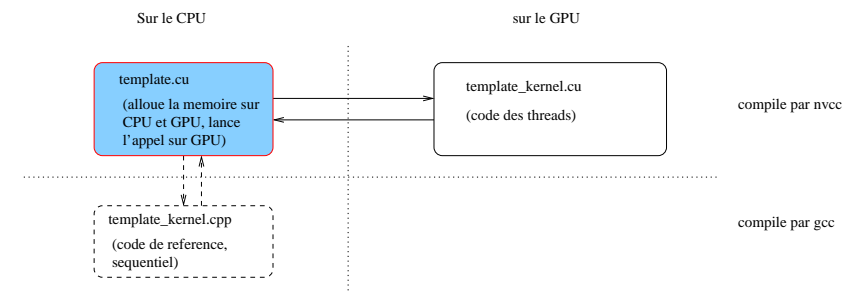
EN PRATIQUE EN TD

- On dispose de cartes GeForce GTX 260 avec 192 coeurs pour les TDs (salles 30 [pays], 35 [os], 36 [voitures])
 - on peut en théorie atteindre 600 GFlops crête!
 - 896 Mo de mémoire
- Ceux qui ont des machines pour jeux vidéos... par exemple GeForce série 8 peuvent télécharger CUDA:
 - <http://www.nvidia.com/cuda> accessible sous windows, vista, linux etc.
 - installer le driver CUDA, le compilateur `nvcc` et le sdk (avec de multiples exemples)
- On peut utiliser pour démarrer le `template` de base...

55

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>
```

57

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)
```

59

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

61

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**)

63

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- sur d'autres...32...etc.)

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 blocks et de threads par block définis précédemment (avec pour données, celles données en argument, copiées sur le GPU)

65

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_threads,
    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

67

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_t
    printf( "Test %s\n", (1 == res) ? "PASSED" : "FAILED");
```

On vérifie que les résultats parallèles et séquentiels sont les mêmes

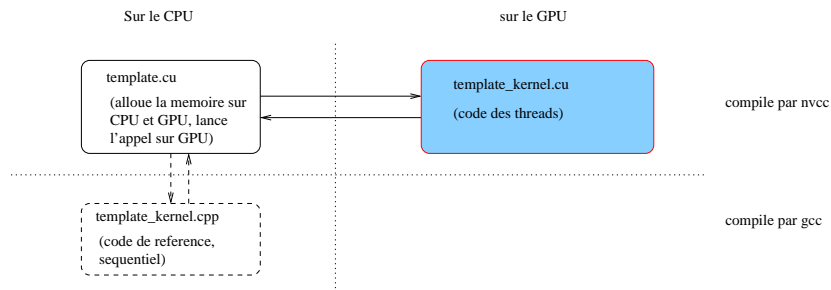
TEMPLATE.CU

```
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

69

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.

71

SUITE DU CODE TEMPLATE_KERNEL.CU

```
__global__ void
testKernel( float* g_idata, float* g_odata) {
__global__: fonction appellable depuis le CPU ou le GPU (et exécutée sur le GPU)
    extern __shared__ float sdata[];
    En mémoire partagée (sur chaque multi-processeur), 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 block, et organisation uni-dimensionnelle des threads, d'où `tid=threadIdx.x`

```
__syncthreads();
```

Synchronisation de tous les threads (bloquant) d'un block.

SUITE DU CODE TEMPLATE_KERNEL.CU

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

Calcul (exemple)

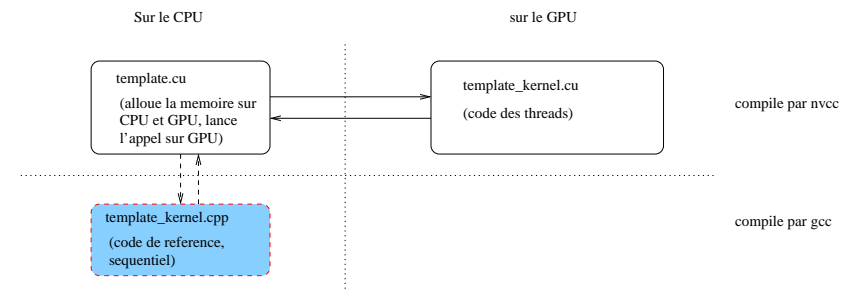
```
__syncthreads();
```

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
 - pour les experts: problèmes d'alignement des données (cf. présentation "optimisation CUDA" sur page web)
- Utilisez la librairie CUT, le "occupancy calculator" (feuille excel - cf. page nvidia.com/cuda) et éventuellement un profiler...

77

CUBIN ET "OCCUPANCY CALCULATOR"

Compiler avec `nvcc -cubin`, ouvrir le fichier `...cubin`:

```

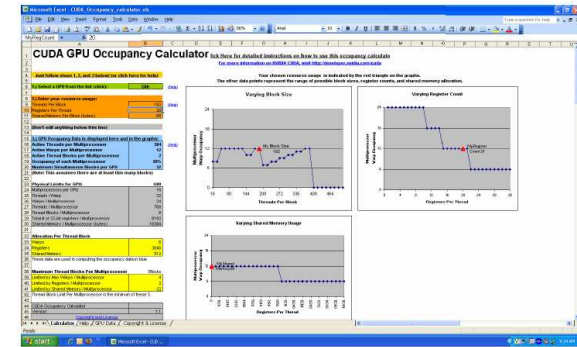
architecture {sm_10}
abiversion {0}
modname {cubin}
code {
    name = BlackScholesGPU
    lmem = 0
    smem = 68
    reg = 20
    bar = 0
    bincode {
        0xa0004205 0x04200780 0x40024c09 0x00200780
    }
}
    
```

Annotations:

- `lmem = 0`: per thread local memory
- `smem = 68`: per thread block shared memory
- `reg = 20`: per thread registers

CUBIN ET "OCCUPANCY CALCULATOR"

Feuille excel: permet de déterminer au mieux le nombre de threads et block:



79

PERFORMANCES...SUITE

- Choisir un nombre de threads grand (multiple du nombre de threads par warp: 32...) pour cacher la latence d'accès à la mémoire
 - Typiquement 128 à 256 (min: 64, max: 512 en général)
 - Mais plus il y a des threads dans un block...plus cela peut être lent quand on fait `__syncthreads()`...
- Choisir un nombre de blocks important (au moins double du nombre de multiprocesseurs - T10: ≥ 60 typiquement)