

INF 560
Calcul Parallèle et Distribué
Cours 3

Eric Goubault

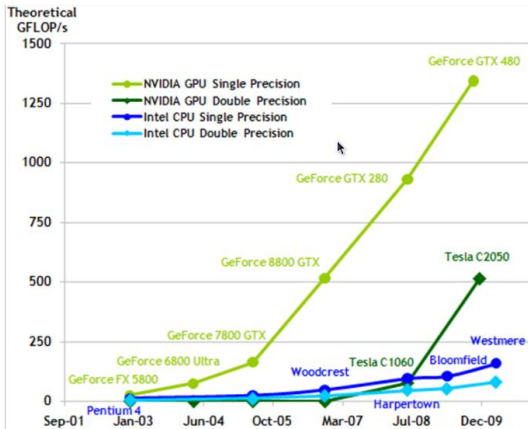
CEA, LIST & Ecole Polytechnique

27 janvier 2014

- 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

CUDA: UN MODÈLE QUASI PRAM?

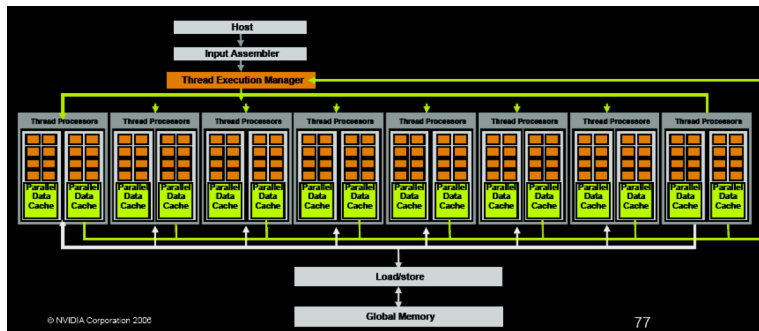
- “Compute Unified Device Architecture”
- Programmation massivement parallèle en C sur cartes NVIDIA, typiquement mémoire partagée
- Tirer parti de la puissance des cartes graphiques:



CUDA: UN MODÈLE QUASI PRAM?

- Peut être programmé pratiquement comme une PRAM CREW, à la différence près que le coût mémoire(s!) est variable et indispensable à gérer (prochain cours)
- On revient à la technique de saut de pointeur (scan, fournie dans la SDK CUDA!); avant cela quelques notions sur CUDA...

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
- Thread processor $\sim$ processus PRAM mais...

MODÈLE D'EXÉCUTION: "COPROCESSEUR PRAM"

C Program
Sequential
Execution

Serial code

Parallel kernel
Kernel0<<<>>>()

Serial code

Parallel kernel
Kernel1<<<>>>()

Host

Device

Grid 0



Host

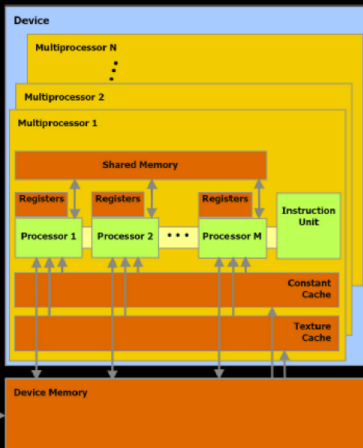
Device

Grid 1



- Organisés en multiprocesseurs (ex. GeForce GT 430 des salles 32, 36: 2 multiproc de 48 coeurs=96 coeurs, à 1.4GHz)
- registres 32 bits par multi-proc.
- **mémoire partagées rapide uniquement par multi-proc.!**
- une mémoire (“constante”) à lecture seule (ainsi qu’un cache de textures à lecture seule)

Host Memory
Device Memory
[Shared Memory]
COMPUTATION
[Shared Memory]
Device Memory
Host Memory



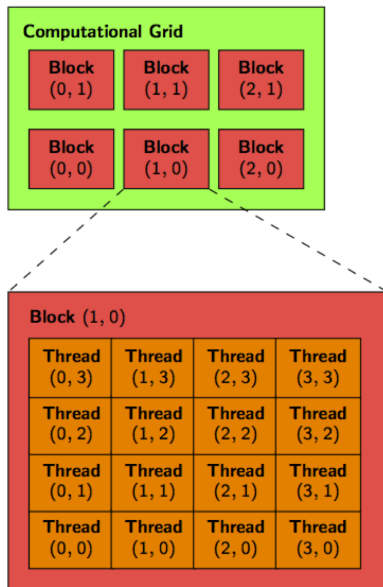
MODÈLE DE PROGRAMMATION - UN PEU DE VOCABULAIRE

- 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"

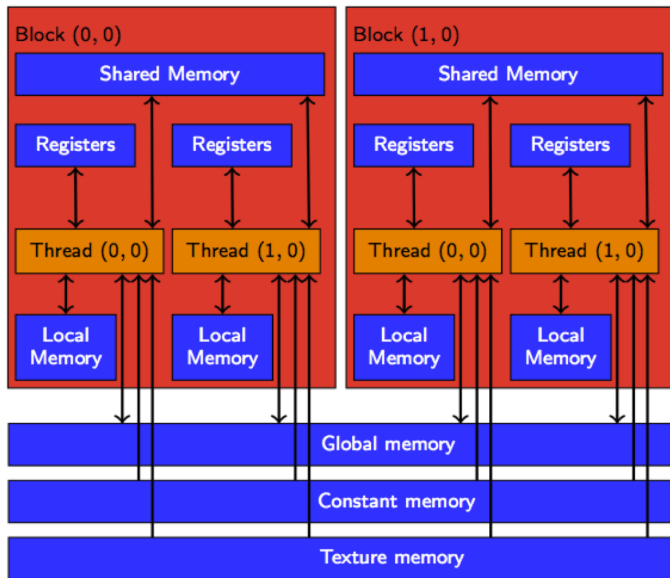
- une grid est un tableau 1D, 2D ou 3D de “thread blocks” - au maximum 65536 blocks par dimension (en pratique, 2D...)
- chaque thread block est un tableau 1D, 2D ou 3D de “threads”, chacun exécutant un clône (instance) du kernel - au maximum 512 threads par block (en général)
- chaque block a un unique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

ATTENTION

Pour toutes ces limitations (nombre de “blocks” etc.), ceci dépend précisément de l'architecture de la carte, à voir en exécutant `deviceQuery` (à importer depuis la SDK et à compiler depuis `nsight`)



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__`

- 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)

- Qualificateurs de types de fonctions:
 - `__device__ f(...)`: la fonction `f` uniquement callable/exécutable sur le GPU (device)
 - `__host__ g(...)`: la fonction `g` uniquement callable/exécutable sur le CPU (host)
 - `__global__ h(...)`: la fonction `h` exécutable sur le GPU et callable 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

- 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()` ;

- 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'autre 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;
const int MAX = 100;

__host__ void add_matrix_cpu(float* a, float *b, float *c, int N) {
    int i, j;
    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            c[i*N+j]=a[i*N+j]+b[i*N+j];
}

__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];
}
```

PREMIER EXEMPLE CUDA: SOMME DE MATRICES

```
int main() {
    int k;
    float *a = new float [N*N];
    float *b = new float [N*N];
    float *c = new float [N*N];

    StopwatchInterface *timer = 0;
    sdkCreateTimer(&timer);

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

    sdkStartTimer(&timer);
    for ( k=1; k<=MAX; k++ ) {
        add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );
        cudaThreadSynchronize();
    }
```

PREMIER EXEMPLE CUDA: SOMME DE MATRICES

```
sdkStopTimer(&timer);
printf(" Processing_time_on_GPU: %f_(ms)\n", sdkGetTimerValue(&timer)/MAX);
sdkDeleteTimer(&timer);
cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );
cudaFree( ad );
cudaFree( bd );
cudaFree( cd );

StopWatchInterface *timer2 = 0;
sdkCreateTimer(&timer2);
sdkStartTimer(&timer2);
for (k=1; k<=MAX; k++)
    add_matrix_cpu(a,b,c, N);
sdkStopTimer(&timer2);
printf(" Processing_time_on_CPU: %f_(ms)\n", sdkGetTimerValue(&timer2)/MAX);
sdkDeleteTimer(&timer2);
delete [] a;
delete [] b;
delete [] c;
return EXIT_SUCCESS;
}
```

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 (JAVA)

```
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);  
    }  
}
```

- 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 recopie synchrones (i.e. bloquantes) `cudaMemcpy` mais aussi fonctions sur des tableaux `cudaMallocPitch` (2D), `cudaMalloc3D` (3D) et `cudaMemcpy2D`, `cudaMemcpy3D`

- 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)

- 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
- ...

- `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) );`

- 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)

- Permet d'améliorer les performances en respectant certaines contraintes d'"alignement" (adresses mémoire multiples de 16/64 bits...)

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`

- 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):

```
[truite ~]$ export PATH=/usr/local/cuda-5.0/bin:${PATH}
[truite ~]$ export LD_LIBRARY_PATH=/usr/local/cuda-5.0/lib:$LD_LIBRARY_PATH
[truite ~]$ nvcc -I/usr/local/cuda-5.0/include/
-I/users/profs/info/goubaul1/CUDA5.0/common/inc/
-L/users/profs/info/goubaul1/CUDA5.0/common/lib/ matrix.cu
```

UTILISER LE TEMPLATE CUDA ET SON MAKEFILE

Sous /users/profs/info/goubaul1/CUDA5.0 (à recopier chez soi):

```
[truite template]$ make
/usr/local/cuda-5.0/bin/nvcc -m32 -gencode arch=compute_10,code=sm_10 -gencode
arch=compute_20,code=sm_20 -gencode arch=compute_30,code=sm_30
-gencode arch=compute_35,code=sm_35 -I/usr/local/cuda-5.0/include -I. -I..
-I../common/inc -o template.o -c template.cu
g++ -m32 -I/usr/local/cuda-5.0/include -I. -I.. -I../common/inc -o
template_cpu.o -c template_cpu.cpp
g++ -m32 -o template template.o template_cpu.o -L/usr/local/cuda-5.0/lib -lcudart
mkdir -p ../../bin/linux/release
cp template ../../bin/linux/release
[truite template]$ ls ../../bin/linux/release/template
../../bin/linux/release/template
[truite template]$ ../../bin/linux/release/template
../../bin/linux/release/template Starting...
```

GPU Device 0: "Quadro_K2000" with compute capability 3.0

Processing time: 92.646004 (ms)

UTILISER NSIGHT (ECLIPSE)

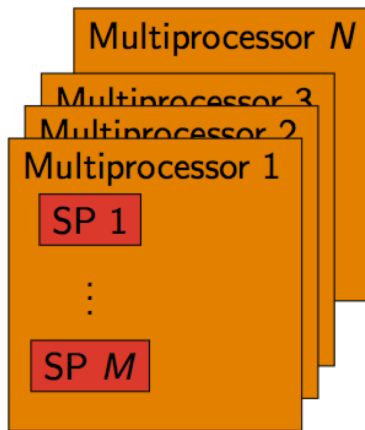
Reprendre exactement la manip sur la page TD 3!

```
[truite ~]$ nvcc -I/usr/local/cuda/include... -o matrix
[truite ~]$ ./matrix
Processing time on GPU: 0.319990 (ms)
Processing time on CPU: 3.234090 (ms)
```

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

- Code sur GPU un peu plus de 10 fois plus rapide que sur CPU.
- Attention, si vous augmentez trop `N` ou `blocksize` vous aurez un temps de l'ordre de 0.02ms (juste le temps d'appel du noyau, qui plante en fait...utiliser les fonctions `checkCudaErrors` de `helper...h` pour tester les codes de retour d'erreur!)

Mapping physique block $\rightarrow$ multiprocesseurs

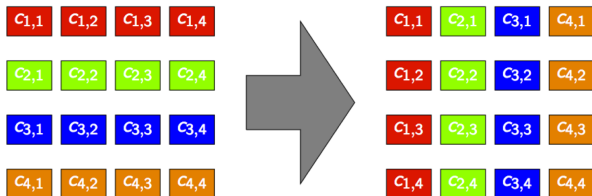


- Un block est exécuté par un seul multiprocesseur
- Chaque block est divisé en groupes de threads (“physiques”) appelés “warps”
- Un warp (en général 32 threads) est exécuté physiquement en parallèle - parallélisme “SIMT” (SIMD, mais synchrone avec seulement une divergence possible de flot de contrôle)
- Un warp est constitué de threads de threadIdx consécutifs et croissants
- L’ordonnanceur de la carte alterne entre les warps



- 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



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...

```
[truite ~]$ nvcc -I/usr/local/cuda/include ...  
[truite ~]$ ./transpose  
Processing time (naive) on GPU: 0.451870 (ms)  
Processing time on CPU: 10.354730 (ms)
```

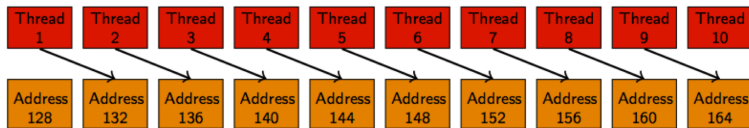
Accélération de 23 fois avec l'approche naive...

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)

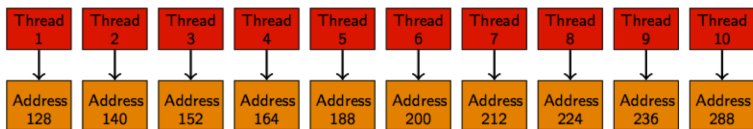
- Point important pour la bonne utilisation de la bande passante mémoire globale vers mémoire partagée: “amalgamation”...
- Il faut faire attention à deux choses dans l'utilisation de la mémoire partagée:
 - Bien utiliser `__syncthreads()`: permet d'attendre que tous les threads d'un même block ont bien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

EXPLICATION: ACCÈS MÉMOIRE NON-AMALGAMÉ



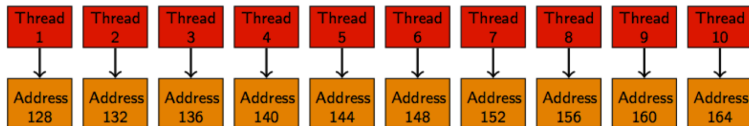
Accès non aligné modulo 16

EXPLICATION: ACCÈS MÉMOIRE NON-AMALGAMÉE



Adresses non “connexes” dans un bloc

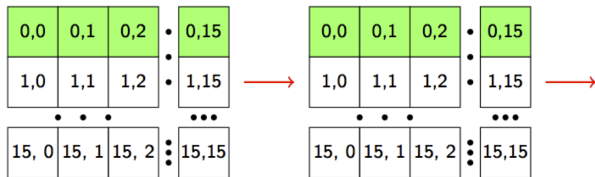
CE QU'IL FAUT FAIRE: ACCÈS MÉMOIRE AMALGAMÉE



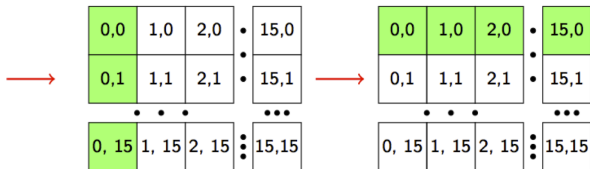
On doit accéder à la mémoire par des accès 8, 16, 32/64 128 bits consécutifs (dans l'ordre des threads pour avant 1.2 - ce n'est pas notre cas ici), dans un bloc mémoire de 32, 64 ou 128 octets; adresse de départ alignée modulo 16

- La matrice est partitionnée en sous-blocs carrés
- Un bloc carré est associé à un block (b_x, b_y) :
 - Charger le bloc (b_x, b_y) 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

```
--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;

    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];}
}
```

```
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 );
```

COMPILATION ET EXÉCUTION

```
[finlande ~]$ nvcc -I/usr/local/cuda/include ...  
[finlande ~]$ ./transpose  
Processing time (naïve) on GPU: 0.451870 (ms)  
Processing time (coalesced) on GPU: 0.324970 (ms)  
Processing time on CPU: 10.354730 (ms)
```

Accélération de 23 fois avec l'approche naïve...de 32 fois avec l'approche améliorée

```
is148098:Cours03 Eric$ ./transpose2  
Processing time (naïve) on GPU: 1.978900 (ms)  
Processing time (coalesced) on GPU: 0.943500 (ms)  
Processing time on CPU: 11.288430 (ms)
```

Amélioration de plus de deux fois sur mon Mac (par rapport à la version naïve)...dépend de l'architecture du nombre de threads, de blocks déclarés etc.

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

// includes CUDA
...
// includes , project
...

////////////////////////////////////
// declaration , forward
void runTest(int argc , char **argv);

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

```
/////////////////////////////////////////////////////////////////
//! Simple test kernel for device functionality
//! @param g_idata input data in global memory
//! @param g_odata output data in global memory
/////////////////////////////////////////////////////////////////
__global__ void
testKernel(float *g_idata, float *g_odata)
{
    // shared memory
    // the size is determined by the host application
    extern __shared__ float sdata[];

    // access thread id
    const unsigned int tid = threadIdx.x;
    // access number of threads in this block
    const unsigned int num_threads = blockDim.x;

    // read in input data from global memory
    sdata[tid] = g_idata[tid];
    __syncthreads();

    // perform some computations
    sdata[tid] = (float) num_threads * sdata[tid];
    __syncthreads();

    // write data to global memory
    g_odata[tid] = sdata[tid];
}
```

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

/////////////////////////////////////////////////////////////////
//! Run a simple test for CUDA
/////////////////////////////////////////////////////////////////
void
runTest(int argc, char **argv)
{
    bool bTestResult = true;

    printf("%s_Starting...\n\n", argv[0]);

    // use command-line specified CUDA device, otherwise use device with highest Gflops/
    int devID = findCudaDevice(argc, (const char **)argv);

    StopWatchInterface *timer = 0;
    sdkCreateTimer(&timer);
    sdkStartTimer(&timer);

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

    // allocate host memory
    float *h_idata = (float *) malloc(mem_size);
}
```

TEMPLATE.CU

```
// initialize the memory
for (unsigned int i = 0; i < num_threads; ++i)
{   h_idata[i] = (float) i;   }
// allocate device memory
float *d_idata;
checkCudaErrors(cudaMalloc((void **) &d_idata , mem_size));
// copy host memory to device
checkCudaErrors(cudaMemcpy(d_idata , h_idata , mem_size ,
                           cudaMemcpyHostToDevice));

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

// setup execution parameters
dim3  grid(1, 1, 1);
dim3  threads(num_threads, 1, 1);
// execute the kernel
testKernel<<< grid , threads , mem_size >>>(d_idata , d_odata);
// check if kernel execution generated an error
getLastCudaError("Kernel_execution_failed");

// allocate mem for the result on host side
float *h_odata = (float *) malloc(mem_size);
// copy result from device to host
checkCudaErrors(cudaMemcpy(h_odata , d_odata , sizeof(float) * num_threads ,
                           cudaMemcpyDeviceToHost));

sdkStopTimer(&timer);
printf("Processing_time:_%f_(ms)\n" , sdkGetTimerValue(&timer));
sdkDeleteTimer(&timer);
```

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

// check result
if (checkCmdLineFlag(argc, (const char **) argv, "regression"))
{
    // write file for regression test
    sdkWriteFile("./data/regression.dat", h_odata, num_threads, 0.0f, false);
}
else
{
    // custom output handling when no regression test running
    // in this case check if the result is equivalent to the expected solution
    bTestResult = compareData(reference, h_odata, num_threads, 0.0f, 0.0f);
}

// cleanup memory
free(h_idata);
free(h_odata);
free(reference);
checkCudaErrors(cudaFree(d_idata));
checkCudaErrors(cudaFree(d_odata));

cudaDeviceReset();
exit(bTestResult ? EXIT_SUCCESS : EXIT_FAILURE);
}
```

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

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//! Compute reference data set
//! Each element is multiplied with the number of threads / array length
//! @param reference  reference data, computed but preallocated
//! @param idata      input data as provided to device
//! @param len        number of elements in reference / idata
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
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 (de plus les nouvelles cartes sont bien moins performantes que celles de l'année dernière)
- Essayez de comprendre les raisons:
 - bank conflict (voir prochain cours)
 - 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 le "occupancy calculator" (feuille excel - cf. nvidia.com/cuda) et éventuellement le profiler sous nsight

CUBIN ET "OCCUPANCY CALCULATOR"

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

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

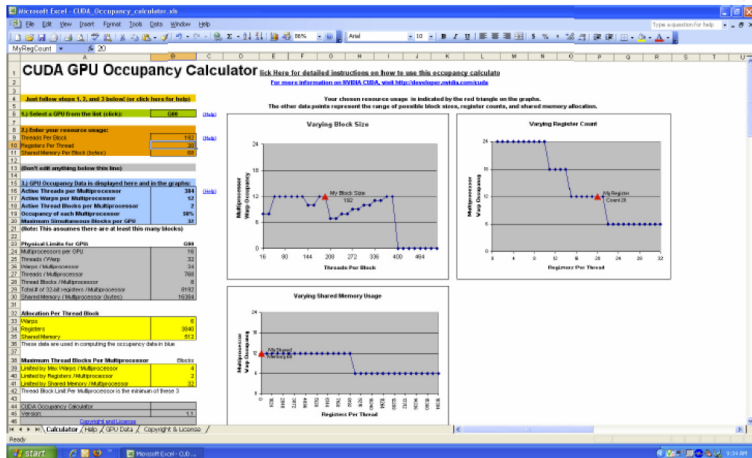
per thread local memory

per thread block shared memory

per thread registers

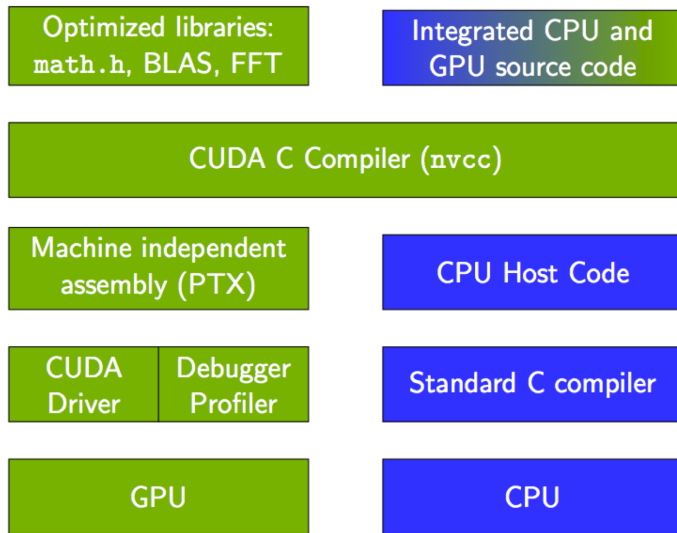
CUBIN ET "OCCUPANCY CALCULATOR"

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

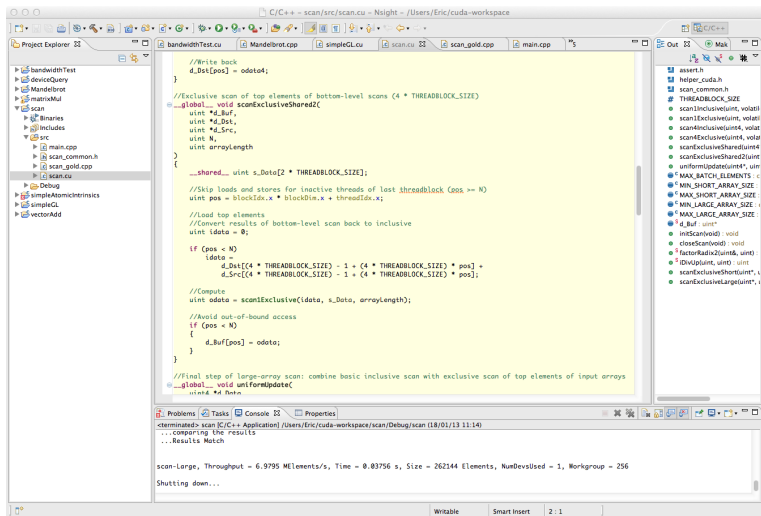


- Choisir un nombre de threads grand (multiple du nombre de threads par warp: 32...) pour cacher la latence d'accès à la mémoire
 - Typiquement 128 à 256 (min: 64, max: 512 en général)
 - Mais plus il y a des threads dans un block...plus cela peut être lent quand on fait `__syncthreads()`...
- Choisir un nombre de blocks important (au moins double du nombre de multiprocesseurs - : ≥ 10 typiquement)

CUDA SDK



LA VERSION DE LA SDK 5.0



The screenshot shows a C/C++ IDE with the following components:

- Project Explorer:** Shows a project structure with folders like `bandwidthTest`, `deviceQuery`, `Mandelbrot`, `matrixMul`, `scan`, `src`, `Debug`, `simpleAtomicIntrinsics`, `simpleGL`, and `vectorAdd`. The `scan` folder is expanded, showing `main.cpp`, `scan_common.h`, `scan_gold.cpp`, and `scan.cu`.
- Code Editor:** Displays the `scan.cu` file. The code implements a scan operation using `scanExclusiveShared2` and `scanExclusiveShared2uim4`. It includes comments for writing back, exclusive scan, global void scan, shared void scan, skip loads and stores, load top elements, compute, avoid out-of-bound access, and final step of large-array scan.
- Output Window:** Shows the execution results of the `scan` application. The output indicates that the scan operation was successful and the results match the expected output.

```
//Write back
d_Dst[pos] = odata4;
}

//Exclusive scan of top elements of bottom-level scans (4 * THREADBLOCK_SIZE)
__global__ void scanExclusiveShared2(
    uint *d_Buf,
    uint *d_Dst,
    uint *d_Src,
    uint N,
    uint arrayLength
)
{
    __shared__ uint s_Data[2 * THREADBLOCK_SIZE];

    //Skip loads and stores for inactive threads of last threadblock (pos >= N)
    uint pos = blockIdx.x * blockDim.x + threadIdx.x;

    //Load top elements
    //Convert results of bottom-level scan back to inclusive
    uint idata = 0;

    if (pos < N)
    {
        idata =
            d_Dst[(4 * THREADBLOCK_SIZE) - 1 + (4 * THREADBLOCK_SIZE) * pos] +
            d_Src[(4 * THREADBLOCK_SIZE) - 1 + (4 * THREADBLOCK_SIZE) * pos];

        //Compute
        uint odata = scanExclusive(idata, s_Data, arrayLength);

        //Avoid out-of-bound access
        if (pos < N)
        {
            d_Buf[pos] = odata;
        }
    }

    //Final step of large-array scan: combine basic inclusive scan with exclusive scan of top elements of input arrays
    __global__ void uniformUpdate(
        uint *d_Buf
    )
{
    //...
}
```

Problems Tasks Console Properties

<terminated> scan [C/C++ Application] /Users/Eric/cuda-workspace/scan/Debug/scan (18/01/13 11:14)

...comparing the results

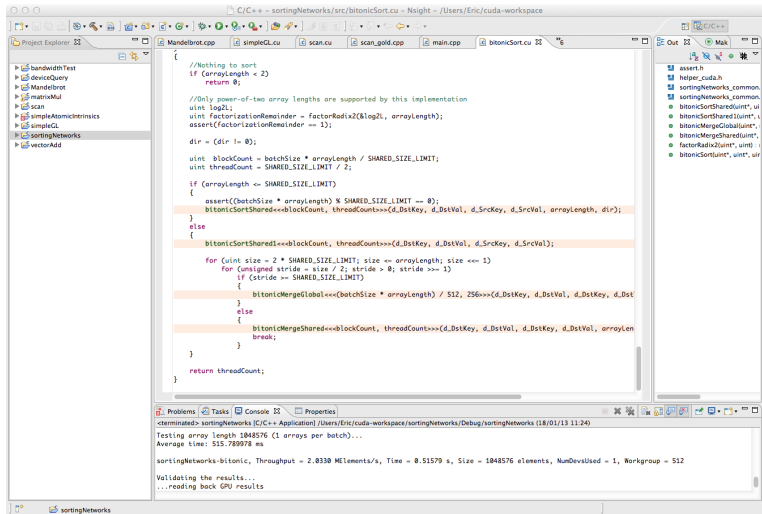
...Results Match

scan-Large, Throughput = 6.9795 MElements/s, Time = 0.83756 s, Size = 262144 Elements, NumDevUsed = 1, Workgroup = 256

Shutting down...

Writable Smart Insert 2:1

REMARQUE: EXEMPLE DE RÉSEAU DE TRI EN CUDA



```
// Nothing to sort
if (arrayLength < 2)
    return 0;

// Only power-of-two array lengths are supported by this implementation
uint log2L;
uint factorizationRemainder = factorRadix2(&log2L, arrayLength);
assert(factorizationRemainder == 1);

dir = (dir != 0);

uint blockCount = batchSize * arrayLength / SHARED_SIZE_LIMIT;
uint threadCount = SHARED_SIZE_LIMIT / 2;

if (arrayLength <= SHARED_SIZE_LIMIT)
{
    assert((batchSize * arrayLength) % SHARED_SIZE_LIMIT == 0);
    bitonicSortShared<<<blockCount, threadCount>>>(d_DstKey, d_DstVal, d_SrcKey, d_SrcVal, arrayLength, dir);
}
else
{
    bitonicSortShared1<<<blockCount, threadCount>>>(d_DstKey, d_DstVal, d_SrcKey, d_SrcVal);

    for (uint size = 2 * SHARED_SIZE_LIMIT; size <= arrayLength; size <= 1)
        for (unsigned stride = size / 2; stride > 0; stride >= 1)
            if (stride >= SHARED_SIZE_LIMIT)
            {
                bitonicMergeGlobal<<<(batchSize * arrayLength) / 512, 256>>>(d_DstKey, d_DstVal, d_DstKey, d_DstVal);
            }
            else
            {
                bitonicMergeShared<<<blockCount, threadCount>>>(d_DstKey, d_DstVal, d_DstKey, d_DstVal, arrayLen);
            }
        }
    }

    return threadCount;
}
```

Problems Tasks Console Properties

<terminated> sortingNetworks [C/C++ Application] /Users/Eric/cuda-workspace/sortingNetworks/Debug/sortingNetworks (18/01/13 11:24)

Testing array length 1048576 (1 arrays per batch)...

Average time: 515.789978 ms

sortingNetworks-bitonic, Throughput = 2.8330 MElements/s, Time = 0.51579 s, Size = 1048576 elements, NumDevsUsed = 1, Workgroup = 512

Validating the results...

...reading back GPU results

POUR ALLER PLUS LOIN (OPENCL)

- Multi-plateforme, beaucoup plus verbeux
- Le code GPU est compilé à l'exécution du code host.
- Vocabulaire un peu différent...:

CUDA	OpenCL
Thread	Work item
Block	Work group
Grid	NDRange
Shared memory	Local memory
Registers	Private memory

EXEMPLE DE PROGRAMME OPENCL

```
cl_program program[1];
cl_kernel kernel[2];
cl_command_queue cmd_queue;
cl_context context;
cl_device_id cpu = NULL, device = NULL;
cl_int err = 0;
size_t returned_size = 0;
size_t buffer_size;
cl_mem a_mem, b_mem, ans_mem;

// Find the CPU CL device, as a fallback
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1, &cpu, NULL);

// Now create a context to perform our calculation with the
// specified device
context = clCreateContext(0, 1, &device, NULL, NULL, &err);
// And also a command queue for the context
cmd_queue = clCreateCommandQueue(context, device, 0, NULL);

// Load the program source from disk
const char * filename = "example.cl";
char *program_source = load_program_source(filename);
program[0] = clCreateProgramWithSource(context, 1, (const char*)&program_source,
NULL, &err);
// Now create the kernel "objects" that we want to use in the example file
kernel[0] = clCreateKernel(program[0], "add", &err);
...
// example.cl
__kernel void
add(__global float *a, __global float *b, __global float *answer)
{
    int gid = get_global_id(0);
    answer[gid] = a[gid] + b[gid];
}
```