GPGPU

General Purpose GPU programming,
ou la programmation SIMD avec CUDA

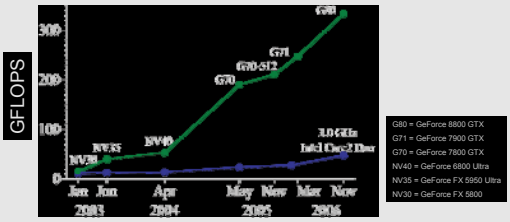
Sources de toute les figures et exemples

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL1, University of Illinois, Urbana-Champaign

Plan du cours

- Introduction
  Pourquoi programmer sur GPU ?
- Le langage CUDA
- Exemple
- Différéntes mémoires
- Exemple : le retour
- Optimisation de la mémoire partagée
- Optimisation de la mémoire globale
- Divergence

Performances GPU/CPU ces dernières années



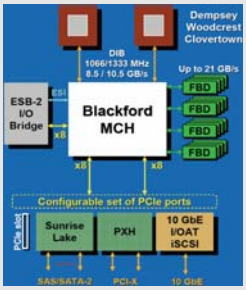
Tendance durable ?

CPU (Intel single core)

- CPU <-> RAM : 6.4 Go/s
- PCIe 16x 4Go/S (R+W)
- 4.8 GFLops

CPU (Intel bi Duo Core2)

- CPU : ~ 2 x 10 Go/s
- PCIe 16x 4Go/S (R+W)
- 4 x 4.8 GFLops

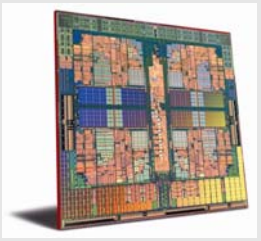


GPU : architecture SIMD

- Single Instruction / Multiple Data
- Similaire au phénomène RISC des années 90 ?

Puces

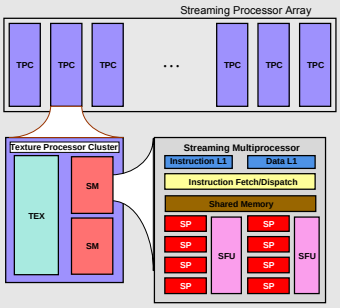
AMD K10 G80…



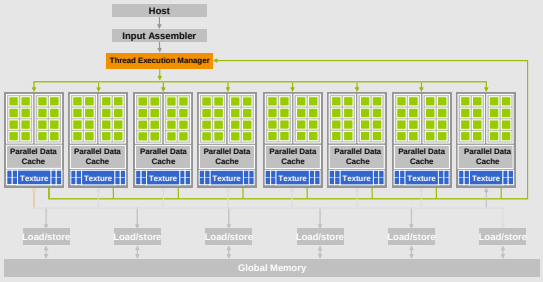
Architecture G80 1/3

SP = Stream Proc
1..512 threads en //
1 instuction =
32 threads
20+ GFlops

SFU =
Super Function Units



Architecture G80 2/3



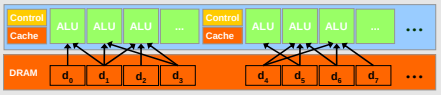
Architecture G80 3/3

- Architecture massivement parallel, 128 cores, 90W
- 367 GFlops maximum théorique
- 265 GFlops mesurés en situation réelles.
- 768 Mo DRAM,
  86 Go/S GPU ←→ RAM,
  4 Go/S CPU ←→ RAM
- Parallelisme massif, des 1000s de threads en parallele

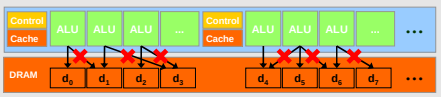
➔ 30-100 x plus rapide sur des applications régulières (Images, Volumes 3D, Matrices, etc.)

Achitecture mémoire pré G80 (OpenGL)

- Gather : lecture multiples
  → textures

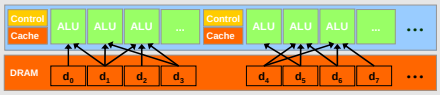


- Scatter : écritures multiples

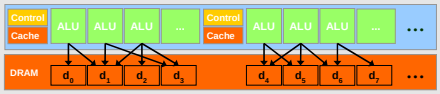


Architecture mémoire G80 (et CUDA)

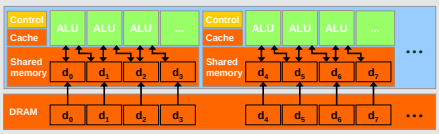
- Gather : lecture multiples
  → adressage libre (textures, vecteur 1D, array 2D)



- Scatter : écritures multiples



Architecture mémoire G80 : mémoire locale



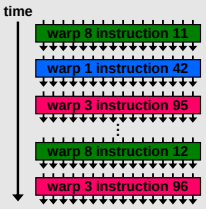
Mémoire partagé sur les multi-processeurs
- Optimisation du bus GPU ←→ RAM (*limité* à 86 Go/s)
- Bus interne à ~ 768 Go/s

Threads et *Warps*

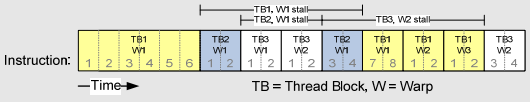
« GigaThreads »

1 Stream Processor exécute des 100s de threads en //

1 accés mémoire ~ 200..400 cycles



Multipléxage temporel de l'exécution

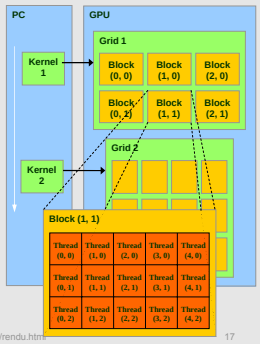


Pour cacher la latence mémoire:
- Plusieurs bloc en parallel sur un même multi-processeur
- *Time slicing* par *warp* de 32 threads

Kernels, blocs et threads

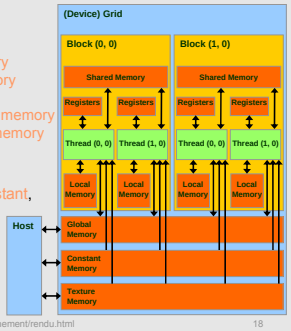
Vision abstraite:

- Kernel = procédure sur GPU
- Organisation 2D x 3D
- `blockIdx`, `threadIdx`
- `gridDim`, `blockDim`



Mémoires

- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory
- The host can R/W global, constant, and texture memories



Plan du cours

- Introduction
- Le langage CUDA
  Compute Unified Device Architecture
- Exemple
- Différéntes mémoires
- Exemple : le retour
- Optimisation de la mémoire partagée
- Optimisation de la mémoire globale
- Divergence

Langage CUDA

- Extension du langage C/C++

__device__, __host__, __global__ (fonctions)
__device__, __shared__, __constant__ (variables)

float3, dim3, … (classes)

__syncthreads()

cudaMalloc(), cudaFree(), …

cudaMemcpy(), …

kernelFunction<<<grid, bloc, shared, stream>>>(…parameters…)

Types spécifiques

Vecteurs 1..4 :
- char, uchar
- short, ushort
- int, uint
- long, ulong
- float
- double
- dim3 variante de uint3, initialisé à 1

```
float3 P = make_float3(1.0,2.0,3.0), Q(3,2,1);
P.x += Q.x;
```


Fonctions standards

- `pow`, `sqrt`, `cbrt`, `hypot`
- `exp`, `exp2`, `expm1`
- `log`, `log2`, `log10`, `log1p`
- `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`
- `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`
- `ceil`, `floor`, `trunc`, `round`
- Etc. etc.

Déclaration des fonctions

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- `__global__` defines a kernel function
  - Must return `void`
- `__device__` and `__host__` can be used together

Lancement des calculs sur le GPU

- __global__ void kernel(float p) { … }

- kernel<<<grid, threads, shared, stream>>>(1.0f);
  - dim3 grid; grid.x * grid.y
  - dim3 threads; threads.x * threads.y * threads.z
  - size_t shared; taille de la mémoire partagée
  - cudaStream_t stream; gestion de la synchrônisation.

Plan du cours

- Introduction
- Le langage CUDA
- Exemple
  Produit matrice x matrice
- Différéntes mémoires
- Exemple : le retour
- Optimisation de la mémoire partagée
- Optimisation de la mémoire globale
- Divergence

Un exemple : produit de matrices

- Matrices 2 D
- Simple précision 32 bits (float)
- *width* * *height* éléments

```
typedef struct {
    int width;
    int height;
    int pitch;
    float* elements;
} Matrix;
```

Générique:

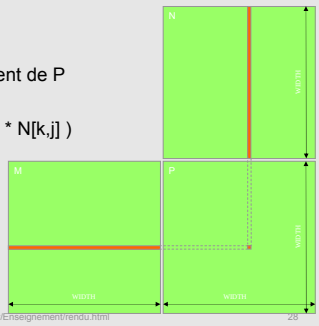
- malloc() et organisation 1D
- *pitch* allocation spéciale, et sous matrices.

Transferts CPU/GPU

```
1. Matrix Md, M;
   size_t size = Md.width * Md.height * sizeof(float);

2. cudaMalloc((void**)&Md.elements, size);

3. cudaMemcpy(Md.elements, M.elements, size,
   cudaMemcpyHostToDevice);

4. … calculs sur le GPU …

5. cudaMemcpy(M.elements, Md.elements, size,
   cudaMemcpyDeviceToHost);

6. cudaFree(Md.elements);
```


Multiplication de matrices

- P = M x N
- Une thread par élément de P
- P[i,j] = somme(M[i,k] * N[k,j])



Implémentation CPU

```
void MatrixMulOnHost(Matrix M, Matrix N, Matrix P)
{
    for (int i = 0; i < M.height; ++i)
        for (int j = 0; j < N.width; ++j) {
            float sum = 0.0;
            for (int k = 0; k < M.width; ++k) {
                float a = M.elements[i * M.pitch + k];
                float b = N.elements[k * N.pitch + j];
                sum += a * b;
            }
            P.elements[i * N.pitch + j] = sum;
        }
}
```


Implémentation GPU

```
__global__ void MatrixMulKernel(Matrix Md, Matrix Nd,
  Matrix Pd)
{
    int tx = threadIdx.x;               // 2D Thread ID
    int ty = threadIdx.y;

    float sum = 0;
    for (int k = 0; k < M.width; ++k) {
         float a = Md.elements[ty * Md.pitch + k ];
         float b = Nd.elements[k  * Nd.pitch + tx];
         sum += a * b;
    }

    Pd.elements[ty * Pd.pitch + tx] = Pvalue;
}
```

Lancement du calcul

Appel:

```
dim3 dimBlock(WIDTH, WIDTH);
MatrixMulKernel<<<1, dimBlock>>>(Md, Nd, Pd);
```

Asynchrone

I/O bloquantes : cudaMemcpy(…)

I/O non blocantes :
cudaStreamCreate() + cudaMemcpyAsync(…)

Limitations

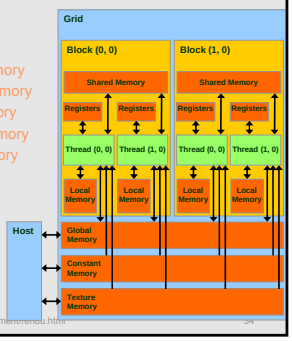
- La matrice est relue WIDTH fois
- 1 seul bloc
- Taille limite 512x512
- N'utilise que la mémoire globale.
- Limité par la bande passante !
  8 octets par instruction MADD, ou 4 octet par Flop
  ➔86/8 = 21 GFlops max (théorique),
  15 GFlops mesurés *(D. Kirk 2007/02)*

Plan du cours

- Introduction
- Le langage CUDA
- Exemple
- Différéntes mémoires
  Shared, constant, global, host, …
- Exemple : le retour
- Optimisation de la mémoire partagée
- Optimisation de la mémoire globale
- Divergence

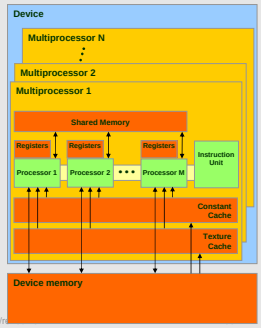
Mémoires 1/2

- Each thread can:
  - Read/write per-thread registers
  - Read/write per-thread local memory
  - Read/write per-block shared memory
  - Read/write per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory



Mémoires 2/2

- Cache ~ Registers ~ Shared
  32 bits / cycle / SP
- Caches
  Constantes, instructions (1D)
  Textures, cudaArray (2D)
  1..10s..100s cycle / 32 bits
- Sans cache
  Mémoire globale
  100s cycles / 32 bits.



Mémoires 3/3

Memory	Location	Cached	Access	Who
Local	Off-chip ?????	No	Read/write	One thread
Shared	On-chip	N/A - resident	Read/write	All threads in a block
Global	Off-chip	No	Read/write	All threads + host
Constant	Off-chip	Yes	Read	All threads + host
Texture	Off-chip	Yes	Read	All threads + host

- `cudaMalloc()` == global memory
- variables → register (+ local) memory
- `__shared__` + appel → shared memory
- `__constant__`

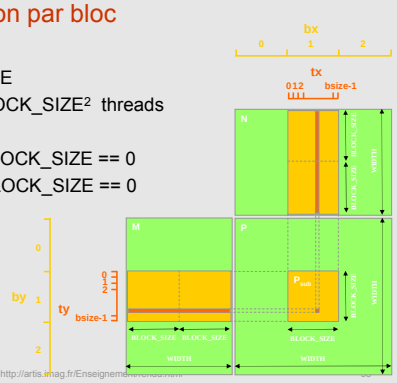
Plan du cours

- Introduction
- Le langage CUDA
- Exemple
- Différéntes mémoires
- Exemple : le retour
  Cache (à la main) en mémoire partagée
- Optimisation de la mémoire partagée
- Optimisation de la mémoire globale
- Divergence

Multiplication par bloc

- BLOCK_SIZE
- 1 bloc = BLOCK_SIZE² threads

- Width % BLOCK_SIZE == 0
- Height % BLOCK_SIZE == 0



Tunning

- 1 bloc = multiple de 32 threads (tout les warps sont complets)
- 1 bloc > 192 threads (optimisation)
- 1 bloc < 512 threads (limite hardware)
- 1 multiproc ≤ 8 blocs en parallel et ≤ 768 threads
- 1 kernel < 65536 x 65536 x 1 blocs
- Au moins 64 blocs pour « remplir » le GPU, plus c'est mieu…

➔ `BLOCK_SIZE = 16`

1 bloc =
- Lit deux matrices 16x16 floats
- 256 threads font 16 MADDs

➔ Max théorique 172 GFlops *(du point de vue de la mémoire globale)*

1 bloc = 2Ko et 16Ko par SM ➔ 8 blocs par MultiProc
1 bloc = 256 threads ➔ 3 blocs par MultiProc

Utilisation de la mémoire partagée

```
__global__ void blocMatrixMultiply(Matrix Md, Matrix Nd, Matrix Pd)
{
    int bx = blockIdx.x;                                    // Block index
    int by = blockIdx.y;
    int tx = threadIdx.x;                                   // Thread index
    int ty = threadIdx.y;

    float sum = 0.0f;
    for(int m=0; m<Md.width/BLOCK_SIZE; ++m) {
          Matrix Msub = GetSubMatrix(Md, m, by);            // Sous-matrices globales
          Matrix Nsub = GetSubMatrix(Nd, bx, m);

          __shared__ float Ms[BLOCK_SIZE][BLOCK_SIZE];      // Copie partagées
          __shared__ float Ns[BLOCK_SIZE][BLOCK_SIZE];

          Ms[ty][tx] = Msub.elements[tx + Msub.pitch*ty];
          Ns[ty][tx] = Nsub.elements[tx + Nsub.pitch*ty];

          __syncthreads();

          for (int k = 0; k < Md.width/BLOCK_SIZE; ++k)
                   sum += Ms[ty][k] * Ns[k][tx];

          __syncthreads();
    }

    Matrix Psub = GetSubMatrix(P, bx, by);
    Psub.elements[ tx + ty * Psub.pitch] = sum;
}
```


Lancement du calcul

```
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid(N.width  / dimBlock.x,
             M.height / dimBlock.y);
Size_t shared = 2 * BLOCK_SIZE * BLOCK_SIZE
  * sizeof(float);

blocMatrixMultiply<<<dimGrid, dimBlock,
  shared>>>(Md, Nd, Pd);
```

This code should run at about 45 GFLOPS

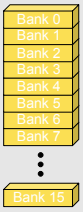
Plan du cours

- Introduction
- Le langage CUDA
- Exemple
- Différéntes mémoires
- Exemple : le retour
- Optimisation de la mémoire partagée
- Optimisation de la mémoire globale
- Divergence

Conflits sur la mémoire partagée

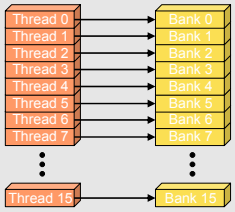
Optimisation de la mémoire partagée
➔Utilisation de 16 banc de mémoire en parallel

- Débit = 32 bits par cycle, et par banc mémoire
- N° banc = (adresse 32 bits) % 16
- Accés multiples == conflits == sérialisation
- Pas de conflits entre (demi) warps différénts

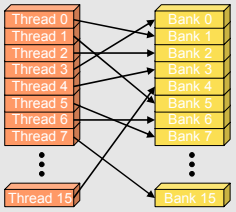


Adressage sans conflits

Pas de conflit
Adressage linéaire
pas == 1

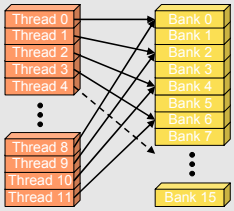


Pas de conflit
Permutation quelconque

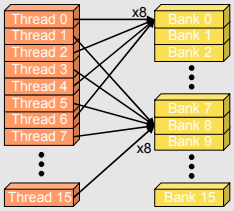


Conflits adresse dans la mémoire partagée

Conflit double accés
Adressage linéaire
pas == 2

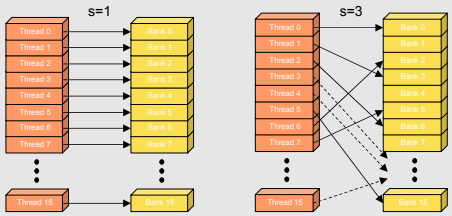


Conflit octuple accés
Adressage linéaire
pas == 8



Cas d'un adressage linéaire

```
__shared__ float shared[256];
float foo = shared[baseIndex + s * threadIdx.x];
```

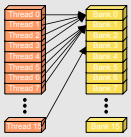


Pas de conflit quand *s est premier avec le nombre de bancs*
Donc pas un multiple de 2, 4, 8 ou 16.

Adressage des types de base en mémoire partagée

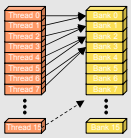
- 8 bits

```
__shared__ char shared[];
foo = shared[baseIndex + threadIdx.x];
```



- 16 bits

```
__shared__ short shared[];
foo = shared[baseIndex + threadIdx.x];
```



Adressage des structures en mémoire partagée

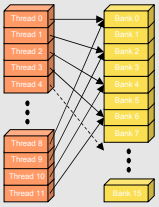
Ok → 3 cycles

```
struct vec3 { float x, y, z; };
__shared__ struct vec3 P3[64];
struct vec3 v = P3[baseIndex + tid];
```

Conflit double accés → 4 cycles

```
struct vec2 {
   float f;
   int c;
};
__shared__ struct vec2 Q2[64];
struct vec2 m = Q2[baseIndex + tid];
```

Egalement : `int2, float2, float4, …`

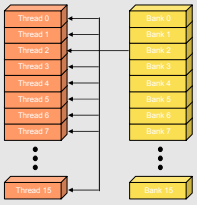


Diffusion depuis un même banc

1 cycle:
```
__shared__ float shared[256];
float foo = shared[constant];
```

2 cycles (64 bits):
```
struct vec2 m = Q2[constant];
```

3 cycles (96 bits):
```
struct vec3 v = P3[constant];
```

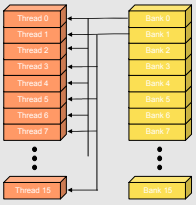


`constant` **== la même valeur pour toutes les threads d'un warp**

Diffusion multiples

2 cycles:
```
__shared__ float shared[256];
float foo = shared[tid % 2];
```

Idem dans les cas mixtes…



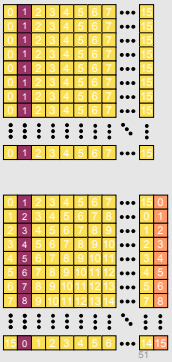
Cas des adressages 2D

- Matrice partagée 16x16 floats

```
int tx = threadIdx.x;
int ty = threadIdx.y;
 __shared__ float Ms[16][16];
 Ms[ty][tx] = …
```

- Matrice partagée 16x17 floats

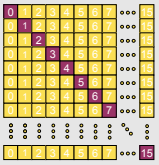
```
__shared__ float Ms[16][17];
```



Cas des adressages 2D

- Matrice partagée 16x16 floats + décalage

```
int ty = threadIdx.y;
int tx = (threadIdx.x + ty) % 16;

Ms[ty][tx] = …
```

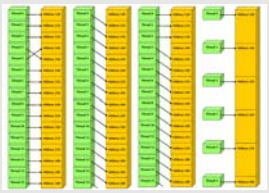


Plan du cours

- Introduction
- Le langage CUDA
- Exemple
- Différéntes mémoires
- Exemple : le retour
- Optimisation de la mémoire partagée
- Optimisation de la mémoire globale
- Divergence

Coalescence des accés globaux

- Bus mémoire en mode « *burst* » sur un demi-warp
- *Coalescence* == lecture d'un bloc consécitif
  Optimisé pour des accés 32, 64, ou 128 bits.
  Contraintes de taille + d'allignement



Coalescence en CUDA

```
float x = globalFloat[tid];    OUI

float2 v = globalVec2[tid];    OUI
float3 w = globalVec3[tid];    NON

float x = globalVec2[tid].x;   NON
v.x = globalX[tid];            OUI
v.y = globalY[tid];

struct __align__(16) {         2 x 128 bits << 5 x 32 bits
  float a;
  float b;
  float c;
  float d;
  float e;
};
```

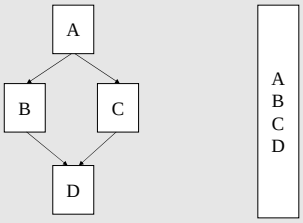

Plan du cours

- Introduction
- Le langage CUDA
- Exemple
- Différéntes mémoires
- Exemple : le retour
- Optimisation de la mémoire partagée
- Optimisation de la mémoire globale
- Divergence

Contrôle de flux et SIMD 1/3

Multi-processeur == SIMD

Divergence == Les 16 *Stream processors* ne font pas la même chose…



Contrôle de flux et SIMD 2/3

CUDA

```
:
:
if (x == 10)
   c = c + 1;
:
:
```

Assembleur PTX

```
      :
      :
      LDR r5 <- X
      p1 <- r5 eq 10
<p1> LDR  r1 <- C
<p1> ADD r1, r1, 1
<p1> STR  r1 -> C
      :
      :
```


Contrôle de flux et SIMD 3/3

```
      :
      :
     p1,p2 <- r5 eq 10
<p1> inst 1 from B
<p1> inst 2 from B
<p1>  :
      :
<p2> inst 1 from C
<p2> inst 2 from C
<p2>  :
      :
```

```
      :
      :
     p1,p2 <- r5 eq 10
<p1> inst 1 from B
<p2> inst 1 from C

<p1> inst 2 from B
<p2> inst 2 from C

<p1>  :
<p2>  :
```


Références

- http://developer.nvidia.com/object/cuda.htm
  CUDA 1.1
  Toolkit = compilateur nvcc
  SDK = 18 exemples
  Linux/Windows XP et 32/64 bits
- http://courses.ece.uiuc.edu/ece498/al1
  > Syllabus