INF 560
Calcul Parallèle et Distribué
Cours 2

Eric Goubault et Sylvie Putot

Ecole Polytechnique

15 décembre 2014

Plan du cours

- Modèle de calcul PRAM
- Algorithmique (technique de saut de pointeur)
- Complexité comparée (Brent etc.)
- Une introduction à la programmation CUDA (proche de PRAM)

PRAM

Le modèle théorique "mémoire partagée" le plus répandu est la PRAM (Parallel Random Access Machine), qui est composée de :

- une suite d'instructions à exécuter plus un pointeur sur l'instruction courante,
- une suite non bornée de processeurs parallèles,
- une mémoire partagée par l'ensemble des processeurs.

La PRAM ne possédant qu'une seule mémoire et qu'un seul pointeur de programme, tous les processeurs exécutent la même opération au même moment.

Communications

Coût d'accès de n'importe quel nombre de processeurs à n'importe quel sous-ensemble de la mémoire, est d'une unité. Trois types d'hypothèses sur les accès simultanés à une même case mémoire:

- EREW (Exclusive Read Exclusive Write): seul un processeur peut lire et écrire à un moment donné sur une case donnée de la mémoire partagée. C'est un modèle proche des machines réelles (et de ce que l'on a vu à propos des threads JAVA).
- CREW (Concurrent Read Exclusive Write): plusieurs processeurs peuvent lire en même temps une même case, par contre, un seul à la fois peut y écrire.

Hypothèses (suite)

- CRCW (Concurrent Read Concurrent Write): Plusieurs processeurs peuvent lire ou écrire en même temps sur la même case de la mémoire partagée:
  - mode consistant: tous les processeurs qui écrivent en même temps sur la même case écrivent la même valeur.
  - mode arbitraire ou prioritaire: c'est la valeur d'un processeur arbitraire ou de celui de plus haut rang/priorité qui est prise en compte.
  - mode fusion: une fonction associative (définie au niveau de la mémoire), est appliquée à toutes les écritures simultanées sur une case donnée. Ce peuvent être par exemple, une fonction maximum, un ou bit à bit etc.

Technique de saut de pointeur

Calcul des préfixes

- Soit $(x_1, \ldots, x_n)$ une suite de nombres. Il s'agit de calculer la suite $(y_1, \ldots, y_n)$ définie par $y_1 = x_1$ et, pour $1 \leq k \leq n$, par

$$y_k = y_{k-1} \otimes x_k = x_1 \otimes x_2 \otimes \ldots \otimes x_k$$

- Pour résoudre ce problème on choisit une PRAM avec n processeurs.

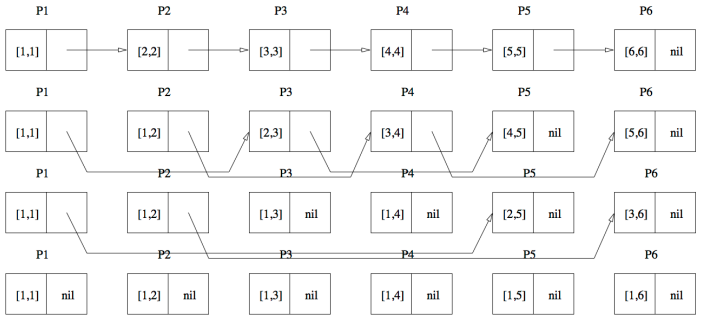
Le principe de l'algorithme est simple:

- La suite $(x_1, \ldots, x_n)$ est représentée par une liste chainée
- A chaque étape, les listes courantes sont dédoublées en des listes des objets en position paire et impaire.
- C'est le même principe que le "diviser pour régner" classique en algorithmique séquentielle.

Algorithme

Avec les notations

- $[i,j] = x_i \otimes x_{i+1} \otimes \ldots \otimes x_j$ pour $i < j$.
- `=_j` si l'instruction est exécutée par le processeur `j`.




```
for each processor i in parallel { y[i] = x[i]; }
while (exists object i s.t. next[i] not nil) {
  for each processor i in parallel {
    if (next[i] not nil) {
      y[next[i]] =_next[i] op(y[i],y[next[i]]);
      next[i] =_i next[next[i]]; } } }
```


Variante

Remarquez que si on avait écrit

```
y[i] =_i op(y[i],y[next[i]]);
```

à la place de

```
y[next[i]] =_next[i] op(y[i],y[next[i]]);
```

on aurait obtenu l'ensemble des préfixes dans l'ordre inverse, c'est-à-dire que P_1 aurait contenu le produit $[1,6]$, P_2 $[2,6]$, jusqu'à P_6 qui aurait calculé $[6,6]$.

Détail technique

Une boucle parallèle du style:

```
for each processor i in parallel
  A[i] = B[i];
```

a en fait exactement la même sémantique que le code suivant:

```
for each processor i in parallel
  temp[i] = B[i];
for each processor i in parallel
  A[i] = temp[i];
```

dans lequel on commence par effectuer les lectures en parallèle, puis, dans un deuxième temps, les écritures parallèles.

Complexité

- Il y a clairement $\lfloor log(n) \rfloor$ itérations et on obtient facilement un algorithme CREW en temps logarithmique
- Il se trouve que l'on obtient la même complexité dans le cas EREW, cela en transformant simplement les affectations dans la boucle, en passant par un tableau temporaire:

```
y[next[i]]=_next[i] op(y[i],y[next[i]]);
```

devient:

```
temp[i]=_next[i] y[next[i]];
y[next[i]] =_next[i] op(y[i],temp[i]);
```


Circuit Eulérien

- On souhaite calculer à l'aide d'une machine PRAM EREW la profondeur de tous les nœuds d'un arbre binaire, c'est-à-dire leur distance à la racine.
- C'est une extension naturelle du problème de la section précédente, pour la fonction "profondeur", sur une structure de données plus complexe que les listes.
- Un algorithme séquentiel effectuerait un parcours en largeur d'abord, et la complexité dans le pire cas serait de $O(n)$ où n est le nombre de noeuds de l'arbre.

Premier algo PRAM naif

- Une première façon de paralléliser cet algorithme consiste à propager une "vague" de la racine de l'arbre vers ses feuilles.
- Cette vague atteint tous les nœuds de même profondeur au même moment (en d étapes, où d est la hauteur de l'arbre), et leur affecte la valeur d'un compteur correspondant à la profondeur actuelle.
- La complexité de ce premier algorithme est de $O(log(n))$ pour les arbres équilibrés, $O(n)$ dans le cas le pire (peigne).

Nous allons maintenant écrire un second algorithme dont la complexité sera toujours $O(log(n))$. Il sera basé sur le concept (théorie des graphes) de circuit Eulérien.

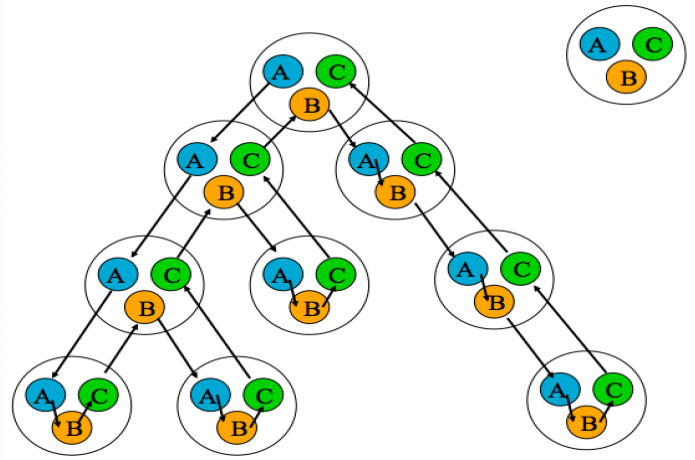
Circuit Eulérien

- Un circuit Eulerien d'un graphe orienté G est un circuit passant une et une seule fois par chaque arc de G.
- Les sommets peuvent être visités plusieurs fois lors du parcours.
- Un graphe orienté G possède un circuit Eulerien si et seulement si le degré entrant de chaque nœud v est égal à son degré sortant.

Cas d'intérêt ici: à partir d'un arbre binaire...

- Il est possible d'associer un cycle Eulerien à tout arbre binaire dans lequel on remplace les arêtes par deux arcs orientés de sens opposés,
- car alors le degré entrant est alors trivialement égal au degré sortant.

Circuit Eulérien pour l'arbre binaire



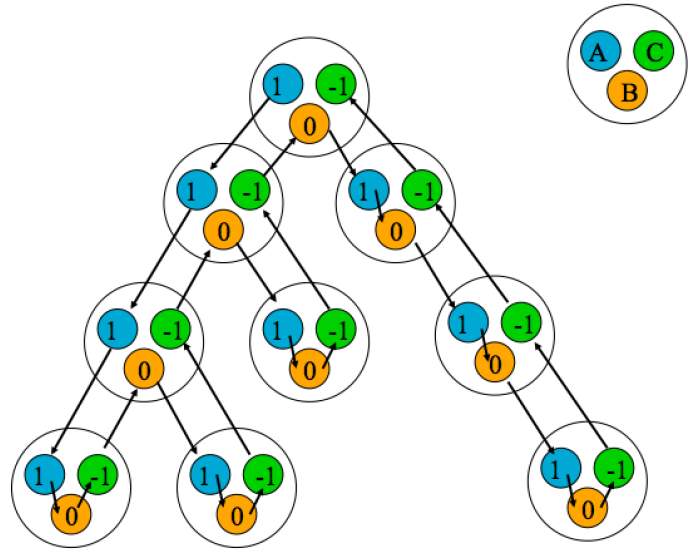
A chaque sommet de l'arbre sont associés 3 processeurs A, B, C.

Principe de l'algorithme

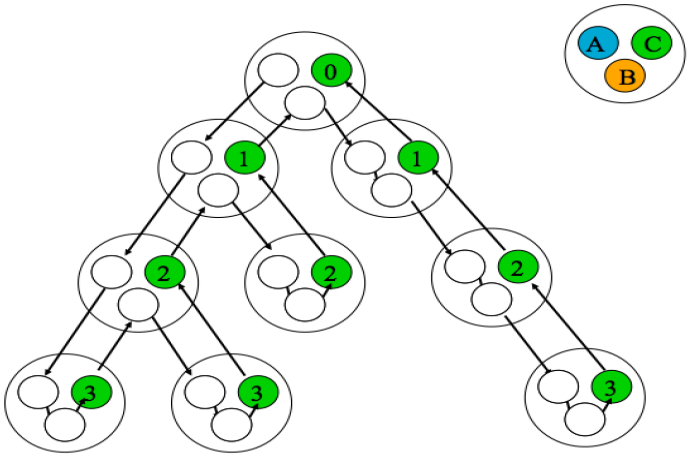
- On organise les noeuds de l'arbre dans une liste, qui est le parcours d'un circuit Eulérien, et on applique l'algorithme de somme partielle par saut de pointeur avec des bons coefficients...
- Il est en effet possible de définir un chemin reliant tous les processeurs et tel que la somme des poids rencontrés sur un chemin allant de la source à un nœud N_i soit égale à la profondeur du nœud i dans l'arbre initial.
- On obtient donc un algorithme en $O(log(n)$ sur une machine EREW.

Voir poly.

Circuit Eulérien pour calcul de profondeur: initialisation



Après calcul des préfixes



Après le calcul de préfixe avec addition des poids, le noeud C contient la profondeur du sommet.

Puissance comparée des différentes PRAM

Séparation CRCW / CREW ou EREW

Exemple 1: calcul du maximum

Calculer le maximum d'un tableau `A` à `n` éléments avec n^2 processeurs.

- Temps constant sur une machine CRCW (mode consistant: les processeurs écrivent la même valeur dans la même case)

```
for each i from 1 to n in parallel        // n processeurs
  m[i] = TRUE;
for each i, j from 1 to n in parallel     // n^2 processeurs
  if (A[i] < A[j]) m[i] = FALSE;
for each i from 1 to n in parallel        // n processeurs
  if (m[i] = TRUE) max = A[i];
```

- $log(n)$ dans le cas d'une CREW ou d'une EREW (les accès concurrents sur une valeur par n processeurs ne peuvent se faire qu'en $log(n)$ étapes).

Puissance comparée des différentes PRAM

Séparation CREW / EREW

Exemple 2: appartenance d'un nombre à un n-uplet

On a un n-uplet $(e_1, \ldots, e_n)$ de nombres tous distincts, et on cherche si un nombre donné e est l'un de ces e_i.

- Temps constant sur une machine CREW sur n processeurs (comme tous les e_i sont distincts, un processeur maximum essaie d'écrire sur `res`, mais tous lisent e en même temps).

```
res = FALSE;
for each i in parallel          // n processeurs
  if (e == e[i])
    res = TRUE;
```

- $log(n)$ sur une machine EREW (il faut dupliquer e sur tous les processeurs, en $log(n)$).

Ce facteur $log(n)$ est maximal (Thm de simulation).

Comparaison des différentes PRAM

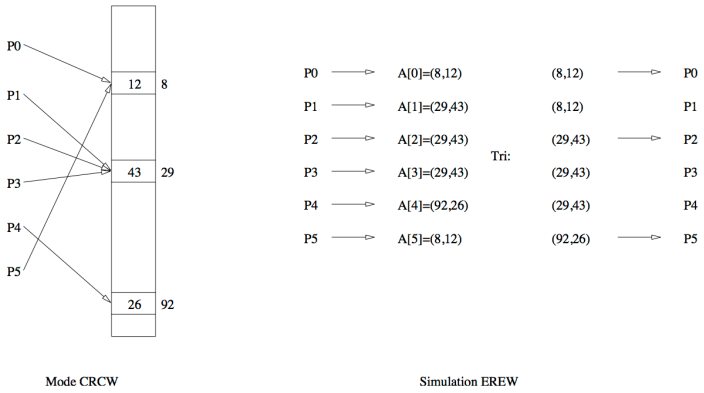
Théorème de simulation

Tout algorithme sur une machine PRAM CRCW (en mode consistant) à p processeurs ne peut pas être plus de $O(log(p))$ fois plus rapide que le meilleur algorithme PRAM EREW à p processeurs pour le même problème.

Preuve:

- Soit un algorithme CRCW à p processeurs. On va en simuler chaque pas en $O(log(p))$ pas d'un algorithme EREW
- On utilise un tableau auxilaire A de p éléments
- Quand un processeur P_i de l'algo CRCW écrit une donnée x_i à l'adresse l_i en mémoire, le P_i de l'algo EREW effectue l'écriture exclusive $A[i] = (l_i, x_i)$. On trie alors le tableau A suivant la première coordonnée en temps $O(log(p))$.
- Puis chaque P_i de l'algorithme EREW inspecte les cases adjacentes $A[i]$ et $A[i-1]$. Si $i = 0$ ou si $l_i \neq l_{i-1}$, P_i écrit la valeur x_i à l'adresse l_i.

Illustration



Simulations

Théorème (Brent)

Soit A un algorithme comportant un nombre total de m opérations et qui s'exécute en temps t sur une PRAM (avec un nombre de processeurs quelconque) Alors on peut simuler A en temps $O\left(\frac{m}{p}+t\right)$ sur une PRAM de même type avec p processeurs.

Preuve:

- A l'étape i, A effectue $m(i)$ opérations, avec $\sum_{i=1}^{t} m(i) = m$.
- On simule l'étape i avec p processeurs en temps $\lceil \frac{m(i)}{p} \rceil \leq \frac{m(i)}{p} + 1$.
- On obtient le résultat en sommant sur les étapes.

Application

Prédire les performances quand on réduit le nombre de processeurs.

Exemple: calcul du maximum, sur une PRAM EREW

- Avec un arbre binaire, calcul en temps $O(logn)$:
  - A l'étape un, on procède paire par paire avec $\lceil \frac{n}{2} \rceil$ processeurs, puis on continue avec les maxima des paires deux par deux etc. C'est à la 1ère étape qu'il faut le plus de processeurs ($O(n)$).
- On dispose maintenant de $p < n$ processeurs.
  - Par le théorème de Brent on peut simuler l'algorithme précédent en temps $O\left(\frac{n}{p}+logn\right)$, car le nombre d'opérations total est $m = n-1$.
  - Si on choisit $p = \frac{n}{logn}$, on obtient le même temps d'exécution, mais avec moins de processeurs!

Notions d'efficacité

Soit P un problème de taille n à résoudre, $T_{seq}(n)$ le temps du meilleur algo séquentiel connu pour résoudre P. Soit $T_{par}(p,n)$ le temps d'un algo qui résout P avec p processeurs.

Le facteur d'*accélération* est défini comme

$$S_p = \frac{T_{seq}(n)}{T_{par}(p,n)}$$

L'*efficacité* comme

$$e_p = \frac{T_{seq}(n)}{pT_{par}(p,n)}$$

Enfin, le *travail* de l'algorithme est

$$W_p = pT_{par}(p,n)$$

Résultat général

Proposition

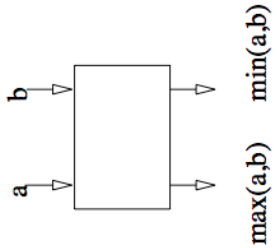
Soit A un algorithme qui s'exécute en temps t sur une PRAM avec p processeurs. Alors on peut simuler A sur une PRAM de même type avec $p' \leq p$ processeurs, en temps $O\left(\frac{tp}{p'}\right)$.

Preuve: avec $p' \leq p$ processeurs, on simule chaque étape de A en temps proportionnel à $\lceil \frac{p}{p'} \rceil$. On obtient un temps total de $O\left(\frac{tp}{p'}\right)$.

On dit qu'un algorithme PRAM est *efficace* quand son travail est de l'ordre de la complexité séquentielle (il est forcément au moins du même ordre).

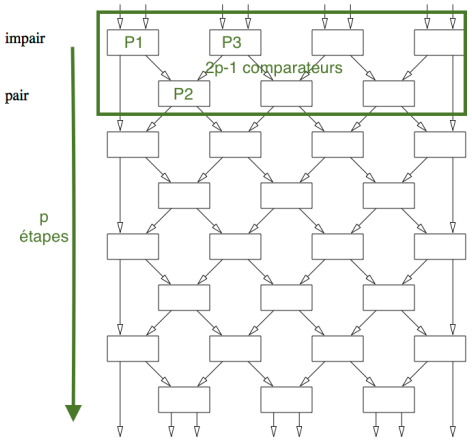
Tris et réseaux de tris

Un réseau de tri est une machine constituée uniquement d'une brique très simple, le comparateur: "circuit" qui prend deux entrées, ici, a et b, et qui renvoie deux sorties: la sortie "haute" est $min(a, b)$, la sortie "basse" est $max(a, b)$.



- Un réseau va être formé d'une succession de lignes de comparateurs
- Une classe particulière de machines PRAM

Tri: réseau de transposition pair-impair



Pour $n = 2p$ éléments à trier, total de $p(2p-1) = \frac{n(n-1)}{2}$ comparateurs dans le réseau. Le tri s'effectue en temps n, le travail est de $O(n^3)$: sous-optimal.

Version réaliste: repliement sur réseau linéaire de processeurs

- Supposons que nous ayons un réseau linéaire de processeurs dans lequel les processeurs ne peuvent communiquer qu'avec leurs voisins de gauche et de droite
- L'idée est de replier le réseau sur un réseau linéaire ou chaque processeur communique alternativement avec le voisin de gauche ou de droite (suivant si étape paire ou impaire)
- Sauf pour les deux extrémités, mais cela a peu d'importance ici, et on aurait pu tout à fait considérer un réseau en anneau comme on en verra plus tard.

Version réaliste sur réseau linéaire

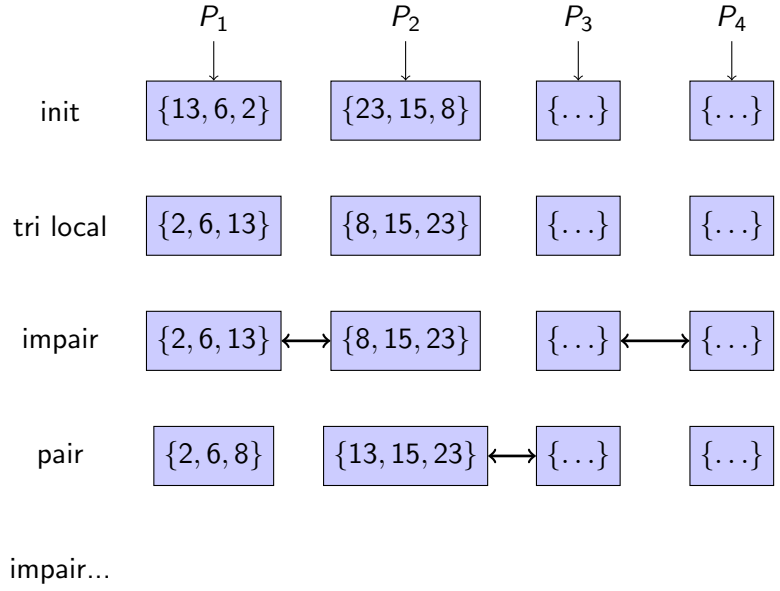
Initialisation

- Supposons n données à trier et p processeurs, n divisible par p.
- On met les données par paquets de $\frac{n}{p}$ sur chaque processeur.
- Chaque suite est triée séquentiellement en temps $O(\frac{n}{p} log \frac{n}{p})$.

Etapes de fusion-échanges

- Ensuite l'algorithme de tri fonctionne en p étapes d'échanges alternés, selon le principe du réseau de tri pair-impair, mais en échangeant des suites de taille $\frac{n}{p}$.
- Quand deux processeurs voisins communiquent, leurs deux suites de taille $\frac{n}{p}$ sont fusionnées (temps $O(\frac{n}{p})$), le processeur de gauche conserve la première moitié, celui de droite la deuxième moitié.
- Temps de calcul $O\left(\frac{n}{p} log \frac{n}{p} + n\right)$, travail $O(n(p + log \frac{n}{p}))$.
- L'algorithme est optimal pour $p \leq logn$.

Illustration tri sur réseau linéaire



Les threads JAVA, modèle PRAM?

- Les threads JAVA peuvent être utilisés pour expérimenter les algorithmes PRAM CREW
- Revenons sur la somme par saut de pointeur: on crée n (ici $n = 8$) threads, le thread p calcule $\sum_{i=0}^{i=p-1}$ `t[i]`.
- A l'étape i des 3=$log_2(n)$ étapes, calcul en parallèle sur $k \geq 2^{i-1}$ de $t[k][i] = t[k - 2^{i-1}][i - 1] + t[k][i - 1]$.
- Mais attention il faut synchroniser explicitement les threads

```
public class SommePartielle extends Thread {
    int pos,i;
    int t[][];
    SommePartielle(int position,int tab[][]) { pos = position; t=tab;  }
    public void run() {
      int i,j;
      for (i=1;i<=3;i++) {
        j = pos-(int)Math.pow(2,i-1);
        while (t[pos][i-1]==0) {} ; // attendre resultat etape i-1
        if (j>=0) {
          while (t[j][i-1]==0) {} ; // attendre resultat etape i-1
          t[pos][i] = t[pos][i-1]+t[j][i-1] ;
          }
        else { t[pos][i] = t[pos][i-1] ; }
 } } }
```


Saut de pointeurs avec des threads JAVA

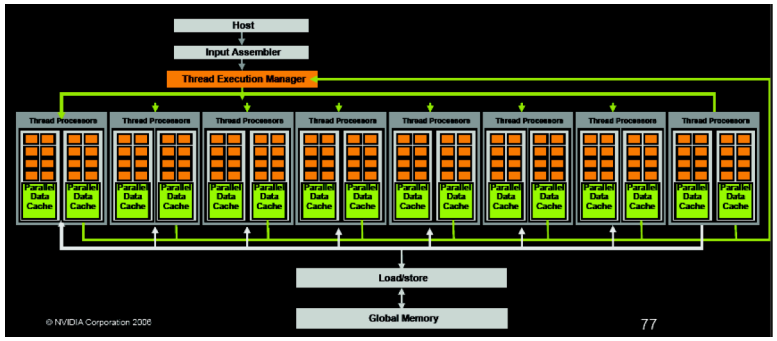
Initialisation du tableau d'origine de façon aléatoire, puis appel du calcul parallèle de la réduction par saut de pointeur

```
import java.util.* ;

public class Exo3 {
    public static void main(String[] args) {
        int[][] tableau = new int[8][4];
        int i,j;
        Random r = new Random();
        for (i=0;i<8;i++) {
            tableau[i][0] =  r.nextInt(8)+1 ;
            for (j=1;j<4;j++) {
                tableau[i][j]=0; }; };
        for (i=0;i<8;i++) {
            new SommePartielle(i,tableau).start(); };
        for (i=0;i<8;i++) {
           while (tableau[i][3]==0) {}; };

        for (i=0;i<4;i++) {
            System.out.print("\n");
            for (j=0;j<8;j++) {
                System.out.print(tableau[j][i]+"␣");
            };
        };
        System.out.print("\n");
    }
}
```

CUDA: un modèle quasi PRAM?

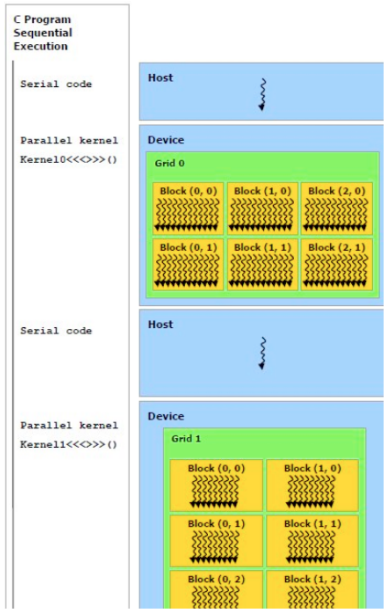
- "Compute Unified Device Architecture"
- Programmation massivement parallèle en C sur cartes NVIDIA
- Tirer parti de la puissance des cartes graphiques:
- Peut être programmé pratiquement comme une PRAM CREW, à la différence près que le coût mémoire(s!) est variable et indispensable à gérer (prochain cours)
- On va revenir à la technique de saut de pointeur (`scan`, fournie dans la SDK CUDA!); avant cela quelques notions sur CUDA...

Architecture physique



- L'hôte peut charger des données sur le GPU, et calculer en parallèle avec le GPU
- Thread processor $\sim$ processus PRAM mais...

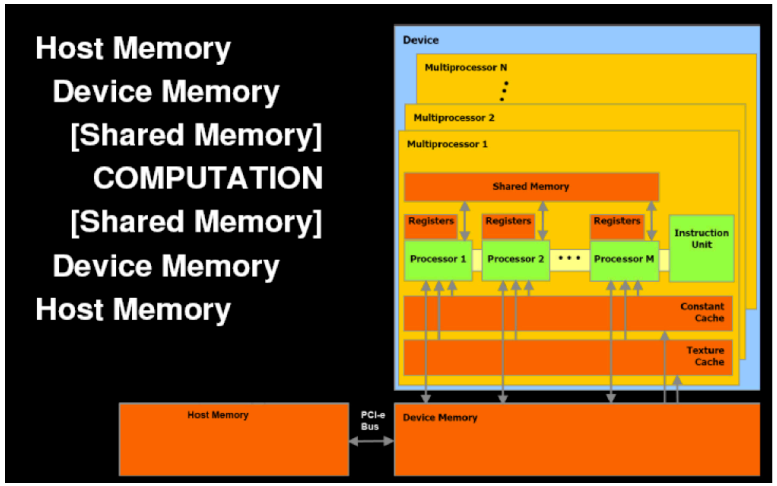
Modèle d'exécution: "coprocesseur PRAM"



Architecture physique

- Organisés en multiprocesseurs (ex. Quadro K2000 de la salle 32: 2 multiproc de 192 coeurs=384 coeurs)
- Registres 32 bits par multi-proc.
- **Mémoire partagée rapide uniquement 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"

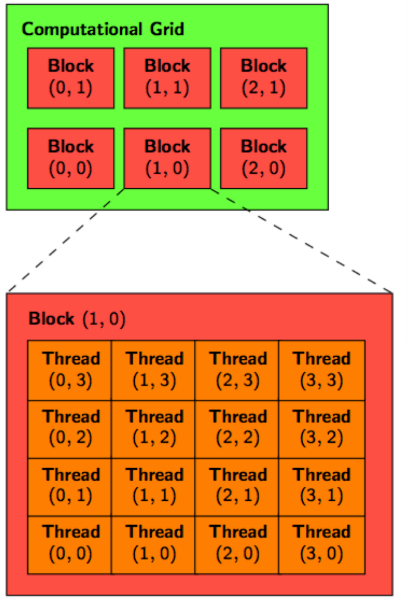
Abstraction logique - grid

- Une grid est un tableau 1D, 2D ou 3D de "thread blocks" - au maximum 65536 blocks par dimension (en pratique, souvent 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 est divisé en "Warps" de 32 threads, qui s'exécutent en SIMD
- Chaque block a un unique `blockId`
- Chaque thread a un unique `threadId` (dans un block donné)

Attention

Les limitations (nombre max de "blocks", de threads par block etc.), dépendent de la carte, à voir en exécutant `deviceQuery` (à importer depuis la SDK et à compiler depuis `nsight`)

Grid

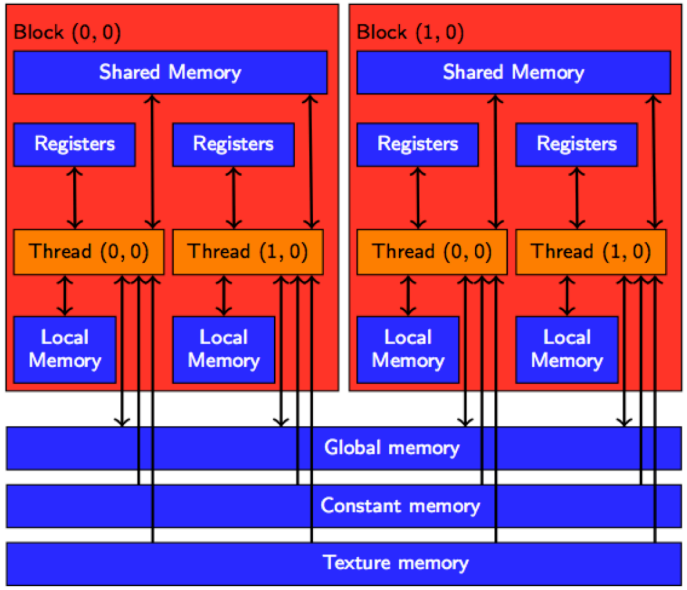


Principe de l'algo (naif) SCAN sur CUDA

```
for d:=1 to log2(n) do
  forall k in parallel do
    offset := pow(2,d-1);
    if k >= offset then
      x[out][k] := x[in][k-offset]+x[in][k];
    else
      x[out][k] := x[in][k];
  done
  assign(x[out],x[in]);
```

- Ici, 1 thread processor par instance de boucle, dans une grid (1,1,1), de blocs unidimensionnels
- Limité à des tableaux de 512 éléments (nombre de threads maxi sur un multi-processeur) - cf. `scan` de la SDK (un peu plus compliqué...)
- Et on n'utilise donc pas toute la carte...
- En fait, il faut passer à des plus gros tableaux et à une optimisation plus fine pour avoir de bonnes performances...

Pourquoi: modèle mémoire



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)

Implémentation CUDA (pas la meilleure!)

(scan_kernel.cu)

```
__global__ void scan(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;
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

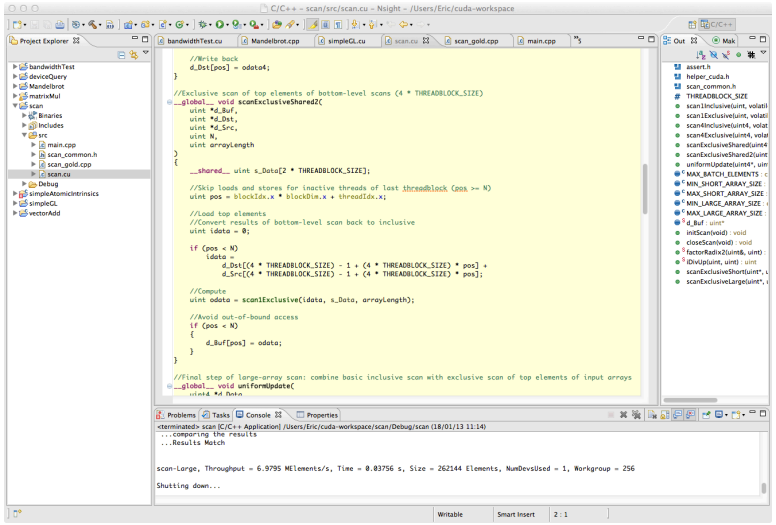
(scan.cu)

```
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 values
// 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_size));
CUDA_SAFE_CALL( cudaMalloc( (void**) &(d_odata[1]), mem_size));
CUDA_SAFE_CALL( cudaMalloc( (void**) &(d_odata[2]), mem_size));
// copy host memory to device input array
CUDA_SAFE_CALL( cudaMemcpy( d_idata, h_data, mem_size,
                            cudaMemcpyHostToDevice) );
// 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
    dim3  grid(256, 1, 1);
    dim3  threads(num_threads*2, 1, 1);

// make sure there are no CUDA errors before we start
CUT_CHECK_ERROR("Kernel execution failed");
unsigned int numIterations = 100;
for (unsigned int i = 0; i < numIterations; ++i)
  {
  scan<<< grid, threads, 2 * shared_mem_size >>>
           (d_odata[0], d_idata, num_elements);
  }
cudaThreadSynchronize();
...
```


La version de la SDK 5.0



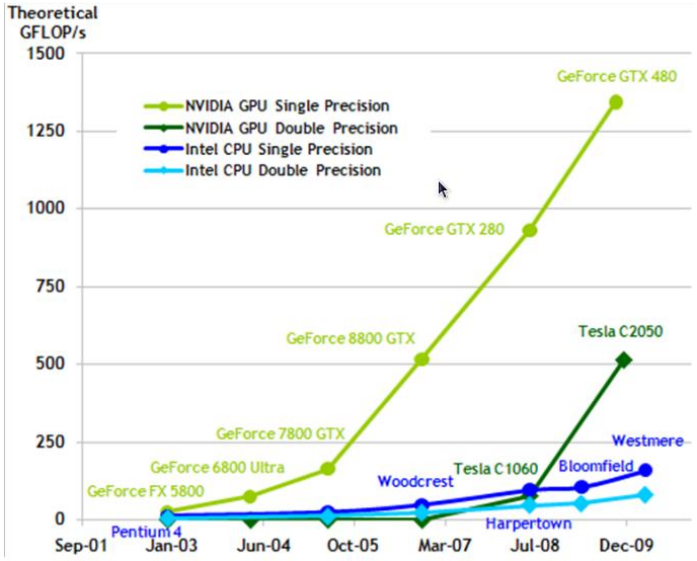
Performance SCAN

Performance

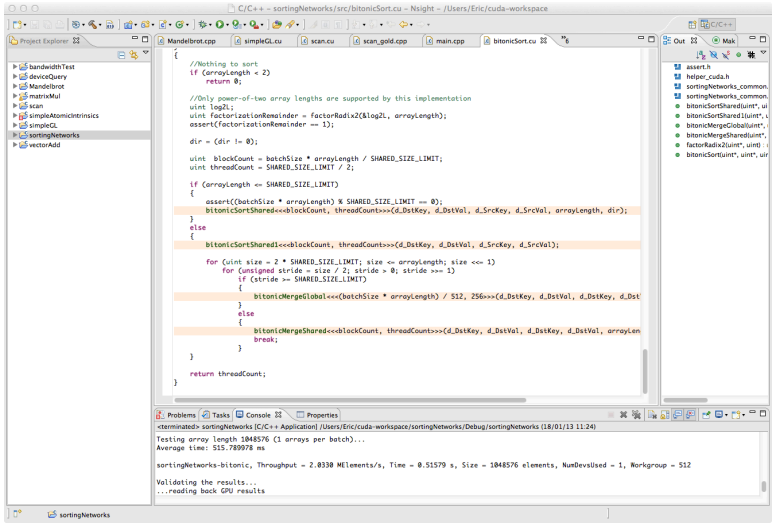
# elements	CPU Scan (ms)	GPU Scan (ms)	Speedup
1024	0.002231	0.079492	0.03
32768	0.072663	0.106159	0.68
65536	0.146326	0.137006	1.07
131072	0.726429	0.200257	3.63
262144	1.454742	0.326900	4.45
524288	2.911067	0.624104	4.66
1048576	5.900097	1.118091	5.28
2097152	11.848376	2.099666	5.64
4194304	23.835931	4.062923	5.87
8388688	47.390906	7.987311	5.93
16777216	94.794598	15.854781	5.98

Table 2: Performance of the work-efficient, bank conflict free Scan implemented in CUDA compared to a sequential scan implemented in C++. The CUDA scan was executed on an NVIDIA GeForce 8800 GTX GPU, the sequential scan on a single core of an Intel Core Duo Extreme 2.93 GHz.

La carte GeForce 8800 n'est plus à l'état de l'art



Remarque: exemple de réseau de tri en CUDA



La suite

- TD2 (Salle Info 32) à 10h15: Threads Java (tri pair-impair, ordonnancement), exo PRAM
- Cours 3 en Amphi Lagarrigue lundi 05 janvier: programmation Cuda (calcul matriciel simple)