# *INF 560*
# *Calcul Parallèle et Distribué*
# *Cours 2*

Eric Goubault

CEA, LIST & Ecole Polytechnique

20 janvier 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)

ÉCOLE POLYTECHNIQUE

# 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.

ÉCOLE POLYTECHNIQUE

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: c'est la valeur du dernier processeur qui écrit 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

- 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

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

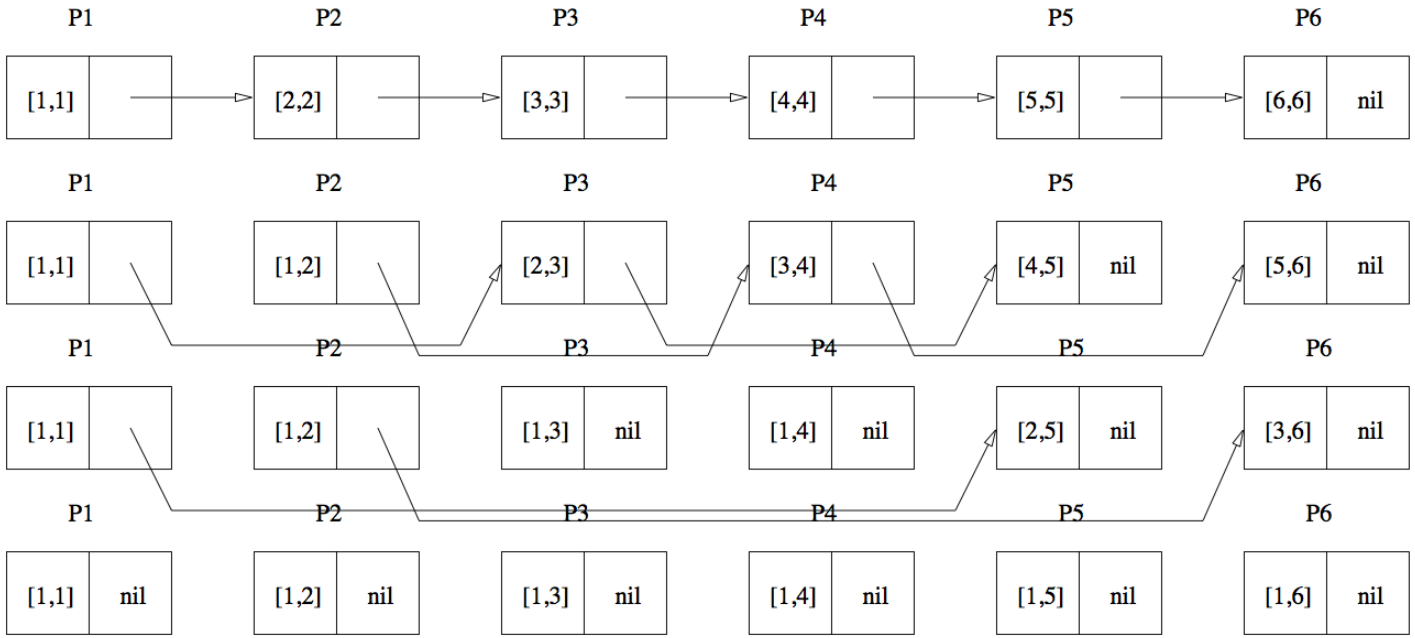
- Pour résoudre ce problème on choisit une PRAM avec $n$ processeurs.

Le principe de l'algorithme est simple:

- à chaque étape de la boucle, les listes courantes sont dédoublées en des listes des objets en position paire,
- et des objets en position impaire.
- C'est le même principe que le "diviser pour régner" classique en algorithmique séquentielle.

# Illustration

Notons $[i,j] = x_i \otimes x_{i+1} \otimes \ldots \otimes x_j$ pour $i < j$.



# Algorithme

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

## En prenant les notations

- op_j pour préciser que l'opération op s'effectue sur le processeur j.
- =_j pour préciser que l'affectation est faite par le processeur j.

# Variante

Remarquez que si on avait écrit

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

à la place de

```
y[next[i]] =_next[i] op_next[i](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.

ÉCOLE POLYTECHNIQUE

# 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 boucles, en passant par un tableau temporaire:

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

devient:

```
temp[i]=_next[i] y[next[i]];
y[next[i]] =_next[i] op_next[i](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 l'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.

# Première parallélisation

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 et leur affecte la valeur d'un compteur correspondant à la profondeur actuelle.

```
actif[0] = true;
continue = true;
p = 0; /* p est la profondeur courante */

forall j in [1,n-1]
  actif[j] = false;
while (continue == true)
  forall j in [0,n-1] such that (actif[j] == true) {
    continue = false;
    prof[j] = p;
    actif[j] = false;
    if (fg[j] != nil) {
      actif[fg[j]] = true;
      continue = true;
      p++; }
    if (fd[j] != nil) {
      actif[fd[j]] = true;
      continue = true;
      p++; }
```

# Problème

- La complexité de ce premier algorithme est de $O(log(n))$ pour les arbres équilibrés, $O(n)$ dans le cas le pire (arbres "déséquilibrés") sur une PRAM CRCW.
- Nous souhaitons maintenant écrire un second algorithme dont la complexité est meilleure que la précédente.

Ceci sera basé sur le concept (théorie des graphes) de circuit Eulérien.

ÉCOLE POLYTECHNIQUE

# 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.

# 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 $C_i$ soit égale à la profondeur du nœud $i$ dans l'arbre initial.

Voir poly (sur le web).

ÉCOLE POLYTECHNIQUE

# Comparaison des différentes PRAM

Calcul le maximum d'un tableau `A` à `n` éléments sur une machine CRCW (mode consistant) à $n^2$ processeurs. Chaque processeur va contenir un couple de valeurs `A[i]`, `A[j]` plus d'autres variables intermédiaires:

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

Maximum en temps constant sur une CRCW ! $/^t \lfloor log(n) \rfloor$ dans le cas d'une CREW, et même d'une EREW.

# Comparaison des différentes PRAM

On a un $n$-uplet $(e_1, \ldots, e_n)$ de nombres tous distincts, et que l'on cherche si un nombre donné $e$ est l'un de ces $e_i$. Sur une machine CREW, on a un programme qui résoud ce problème en temps constant, en stockant chaque $e_i$ sur des processeurs distincts (donc $n$ processeurs en tout):

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

# Comparaison des différentes PRAM

- Comme tous les $e_i$ sont distincts, il ne peut y avoir qu'un processeur qui essaie d'écrire sur `res`, par contre, on utilise ici bien évidemment le fait que tous les processeurs peuvent lire e en même temps.
- Sur une PRAM EREW, il faut dupliquer la valeur de $e$ sur tous les processeurs. Ceci ne peut se faire en un temps meilleur que $log(n)$, par dichotomie.

**Théorème**

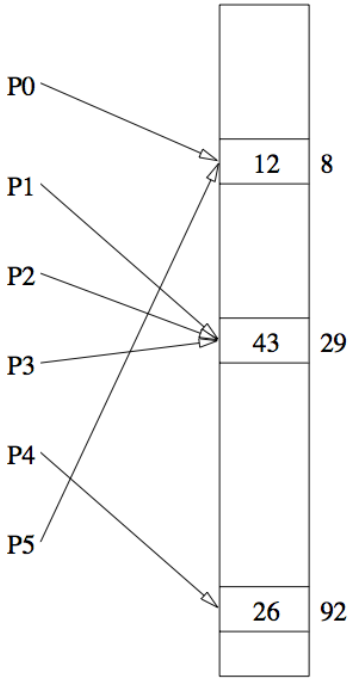
Tout algorithme sur une machine PRAM CRCW (en mode consistant) à $p$ processeurs ne peut pas être plus de $O(log(p))$ fois plus rapide que le meilleur algorithme PRAM EREW à $p$ processeurs pour le même problème.

# Preuve

- Soit un algorithme CRCW à $p$ processeurs.
- On utilise un tableau auxilaire $A$ de $p$ éléments, qui va nous permettre de réorganiser les accès mémoires.
- Quand un processeur $P_i$ de l'algorithme CRCW écrit une donnée $x_i$ à l'adresse $l_i$ en mémoire, le processeur $P_i$ de l'algorithme EREW effectue l'écriture exclusive $A[i] = (l_i, x_i)$. On trie alors le tableau $A$ suivant la première coordonnée en temps $O(log(p))$ (voir algorithme de Cole).
- Une fois $A$ trié, chaque processeur $P_i$ de l'algorithme EREW inspecte les deux cases adjacentes $A[i] = (l_j, x_j)$ et $A[i-1] = (l_k, x_k)$, où $0 \leq j, k \leq p-1$. Si $l_j \neq l_k$ ou si $i = 0$, le processeur $P_i$ écrit la valeur $x_j$ à l'adresse $l_j$, sinon il ne fait rien.

ÉCOLE POLYTECHNIQUE

# Illustration




Mode CRCW

| | | | | |
|---|---|---|---|---|
| P0 | A[0]=(8,12) | | (8,12) | P0 |
| P1 | A[1]=(29,43) | | (8,12) | P1 |
| P2 | A[2]=(29,43) | Tri: | (29,43) | P2 |
| P3 | A[3]=(29,43) | | (29,43) | P3 |
| P4 | A[4]=(92,26) | | (29,43) | P4 |
| P5 | A[5]=(8,12) | | (92,26) | P5 |

Simulation EREW

## 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

- à l'étape $i$, $A$ effectue $m(i)$ opérations, avec $\sum_{i=1}^{n} m(i) = m$.
- 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

- Calcul du maximum, sur une PRAM EREW.
- On peut agencer ce calcul en temps $O(logn)$ à l'aide d'un arbre binaire.
- 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 première étape qu'on a besoin du plus grand nombre de processeurs, donc il en faut $O(n)$.

ÉCOLE POLYTECHNIQUE

# Formellement

- Si $n = 2^m$, si le tableau `A` est de taille $2n$, et si on veut calculer le maximum des $n$ éléments de `A` en position `A[n]`, `A[n+1]`,...,`A[2n-1]`, on obtient le résultat dans `A[1]` après exécution de l'algorithme:

```
for (k=m-1; k>=0; k--)
  for each j from 2^k to 2^(k+1)-1 in parallel
    A[j] = max(A[2j],A[2j+1]);
```

- 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, et soit $T_{seq}(n)$ le temps du meilleur algorithme séquentiel connu pour résoudre $P$. Soit maintenant un algorithme parallèle PRAM qui résout $P$ en temps $T_{par}(p)$ avec $p$ processeurs. Le facteur d'accélération est défini comme:

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

- et l'*efficacité* comme

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

- Enfin, le *travail* de l'algorithme est

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

ÉCOLE POLYTECHNIQUE

# Résultat général

## Proposition

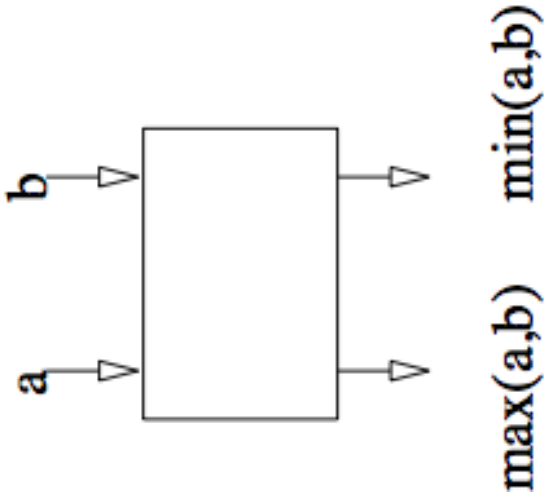
Soit $A$ un algorithme qui s'exécute en temps $t$ sur une PRAM avec $p$ processeurs. Alors on peut simuler $A$ sur une PRAM de même type avec $p' \leq p$ processeurs, en temps $O\left(\frac{tp}{p'}\right)$.

## Preuve

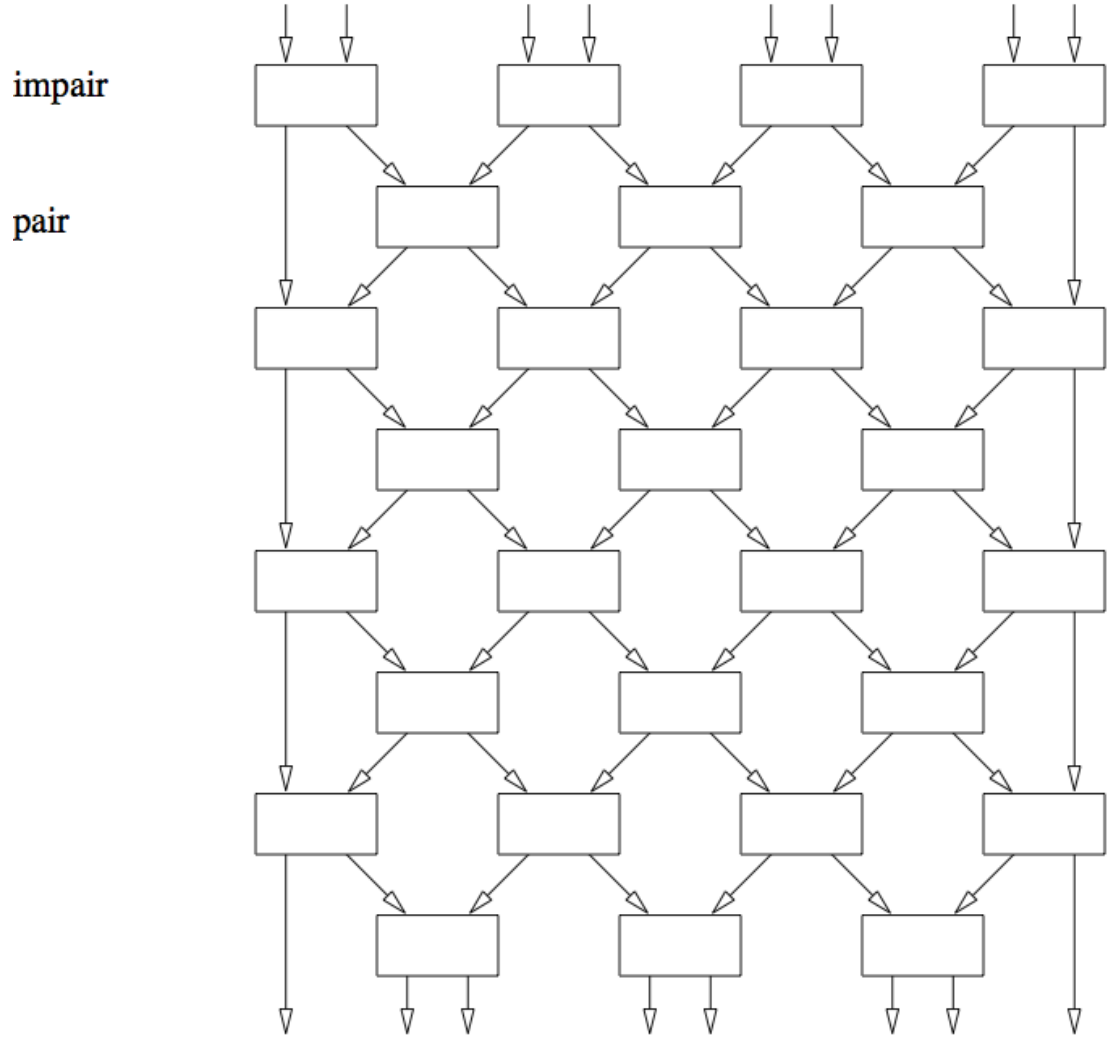
En effet, avec $p'$ processeurs, on simule chaque étape de $A$ en temps proportionnel à $\lceil \frac{p}{p'} \rceil$. On obtient donc un temps total de $O\left(\frac{p}{p'}t\right) = O\left(\frac{tp}{p'}\right)$.

ÉCOLE POLYTECHNIQUE

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 $max(a, b)$.



# Tri pair-impair




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

# Version réaliste

- 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
- 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.

ÉCOLE POLYTECHNIQUE

# Principe

- Supposons que l'on ait $n$ données à trier et que l'on dispose de $p$ processeurs, de telle façon que $n$ est divisible par $p$.
- On va mettre les données à trier par paquet de $\frac{n}{p}$ sur chaque processeur.
- Chacune de ces suites est triée en temps $O(\frac{n}{p} log \frac{n}{p})$.

ÉCOLE POLYTECHNIQUE

# Principe

- 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 $\frac{n}{p}$ à la place d'un seul élément.
- Quand deux processeurs voisins communiquent, leurs deux suites de taille $\frac{n}{p}$ sont fusionnées, le processeur de gauche conserve la première moitié, et celui de droite, la deuxième moitié.
- On obtient donc un temps de calcul en $O\left(\frac{n}{p}log\frac{n}{p} + n\right)$ et un travail de $O(n(p + log\frac{n}{p}))$.
- L'algorithme est optimal pour $p \leq logn$.

ÉCOLE POLYTECHNIQUE

# Les threads JAVA, modèle PRAM?

- Sans synchronisation explicite les threads JAVA peuvent être utilisés pour expérimenter les algorithmes PRAM CREW!
- Voir TD... mais avant cela, revenons sur la somme par technique de saut de pointeur

Ici on crée $n$ (ici $n = 8$) threads, le thread $p$ étant chargé de calculer $\sum_{i=0}^{i=p-1} \texttt{t[i]}$.

```
public class SommePartielle extends Thread {
    int pos,i;
    int t[][];
    SommePartielle(int position,int tab[][]) {
      pos = position;
      t=tab;  }
    int pow(int a,int b) {
      int ctr,r ;
      r=1;
      for (ctr=1;ctr<=b;ctr++) r = r * a;
      return r ; };
    public void run() {
      int i,j;
      for (i=1;i<=3;i++) {
        j = pos-pow(2,i-1);
        if (j>=0) {
          while (t[j][i-1]==0) {} ; // attendre que le resultat soit pret
          t[pos][i] = t[pos][i-1]+t[j][i-1] ;
          } else { t[pos][i] = t[pos][i-1] ; }; }; } }
```

# Principe

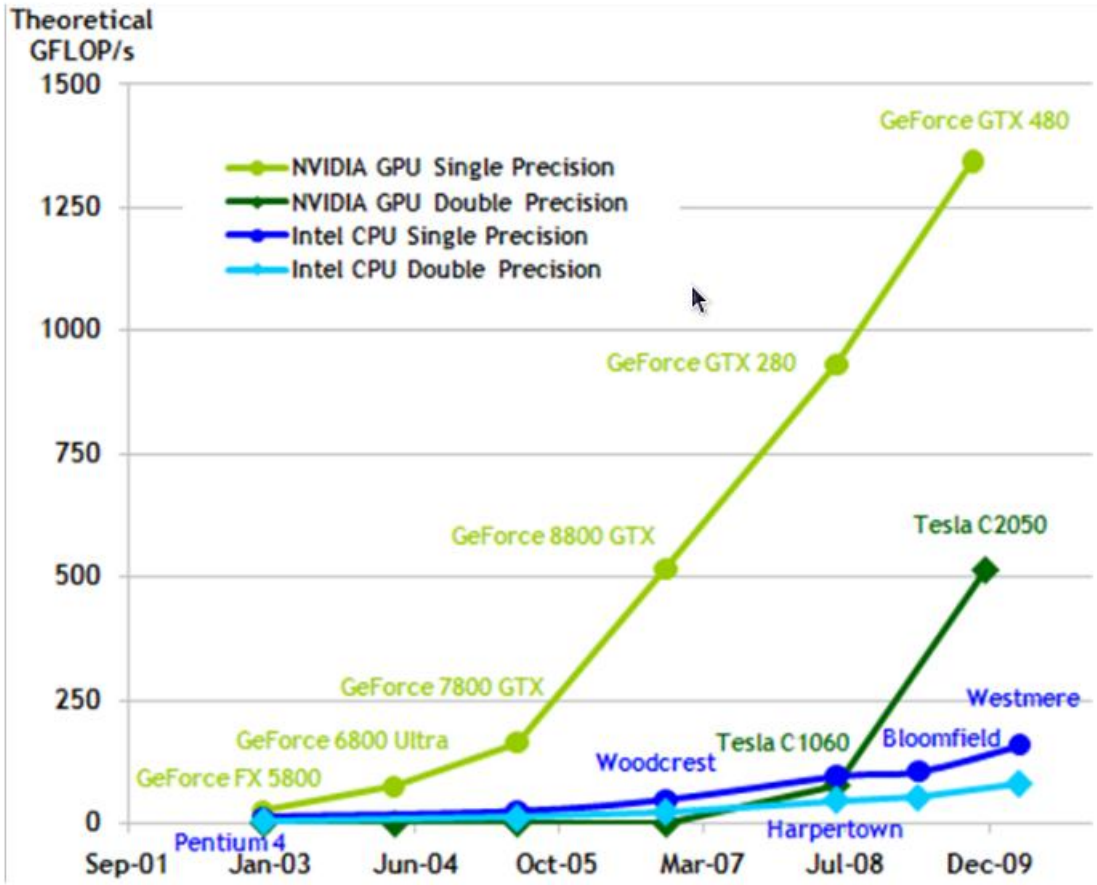
- L'idée est que le résultat de chacune des $3 = log_2(n)$ étapes, disons l'étape $i$, du saut de pointeur se trouve en $t[proc][i]$ ($proc$ étant le numéro du processeur concerné, entre 0 et 8).
- Au départ, on initialise (par l'appel au constructeur `SommePartielle`) les valeurs que voient chaque processeur: $t[proc][0]$.
- Le code suivant initialise le tableau d'origine de façon aléatoire, puis appelle le calcul parallèle de la réduction par saut de pointeur:

# Saut de pointeurs simple avec des threads JAVA

```
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:



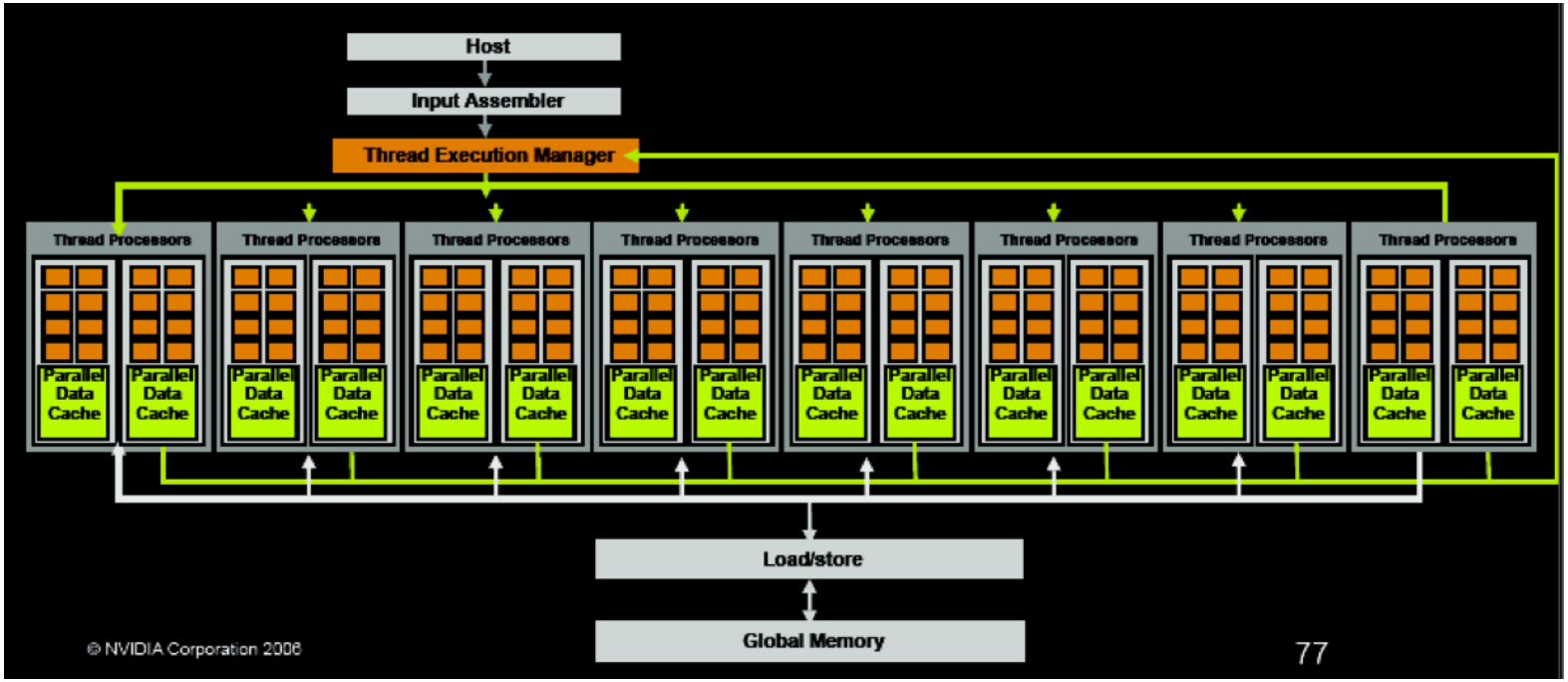

ÉCOLE POLYTECHNIQUE

# 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...

ÉCOLE POLYTECHNIQUE

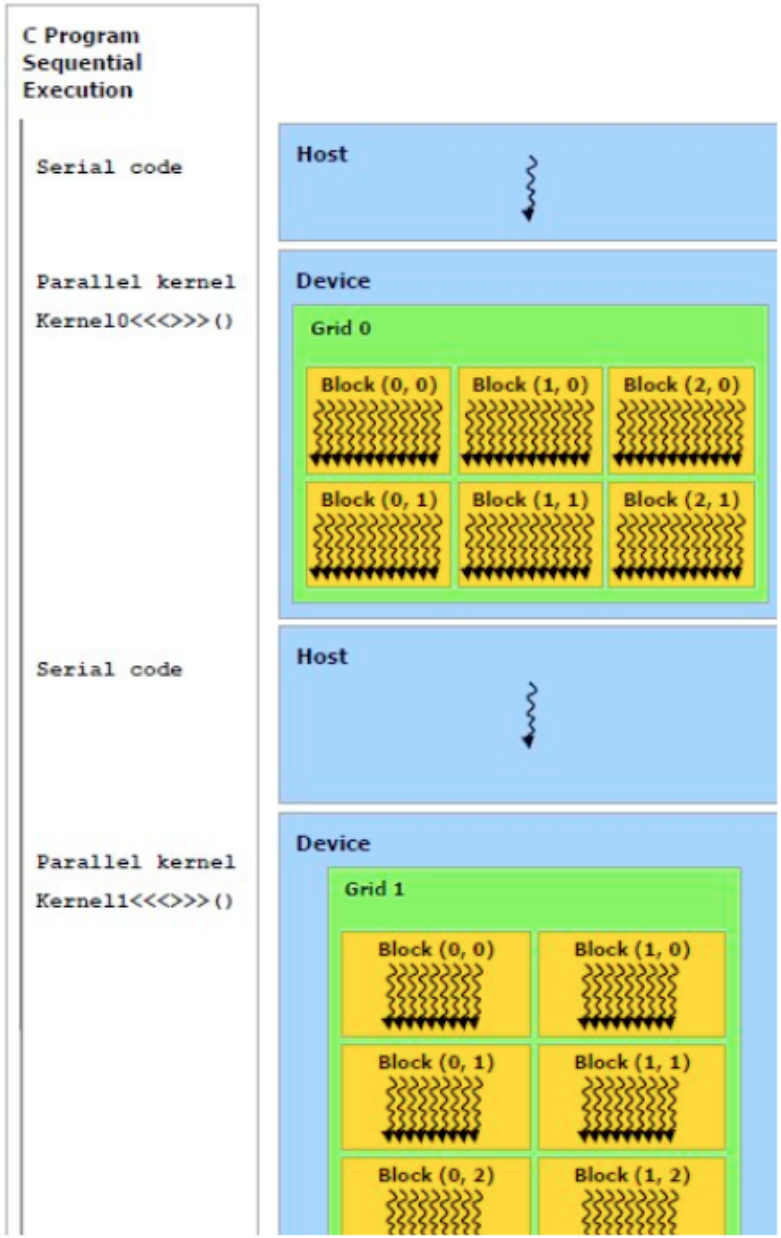
# Architecture physique




(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"

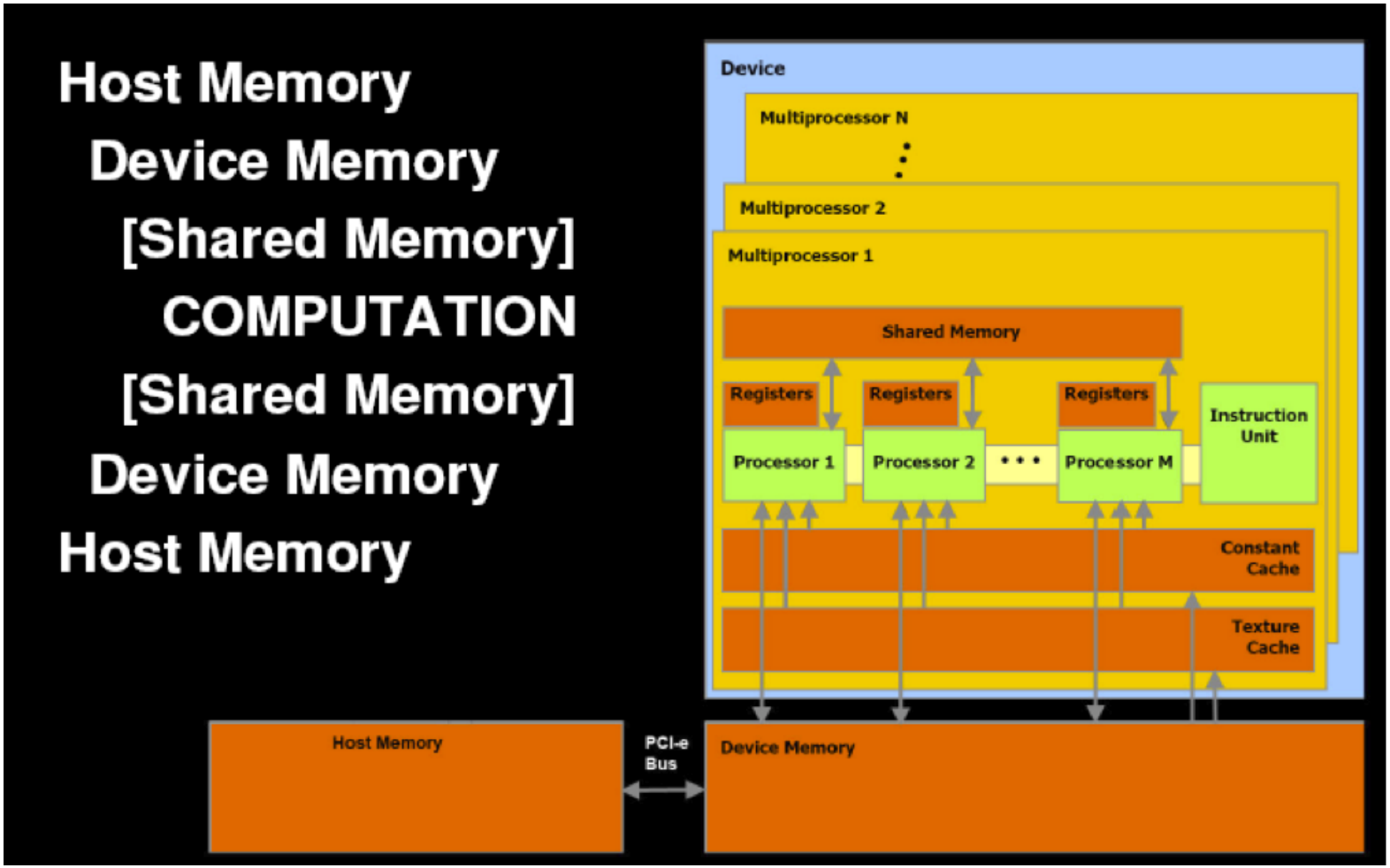


# Architecture physique

- 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)

ÉCOLE POLYTECHNIQUE

# 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"

ÉCOLE POLYTECHNIQUE

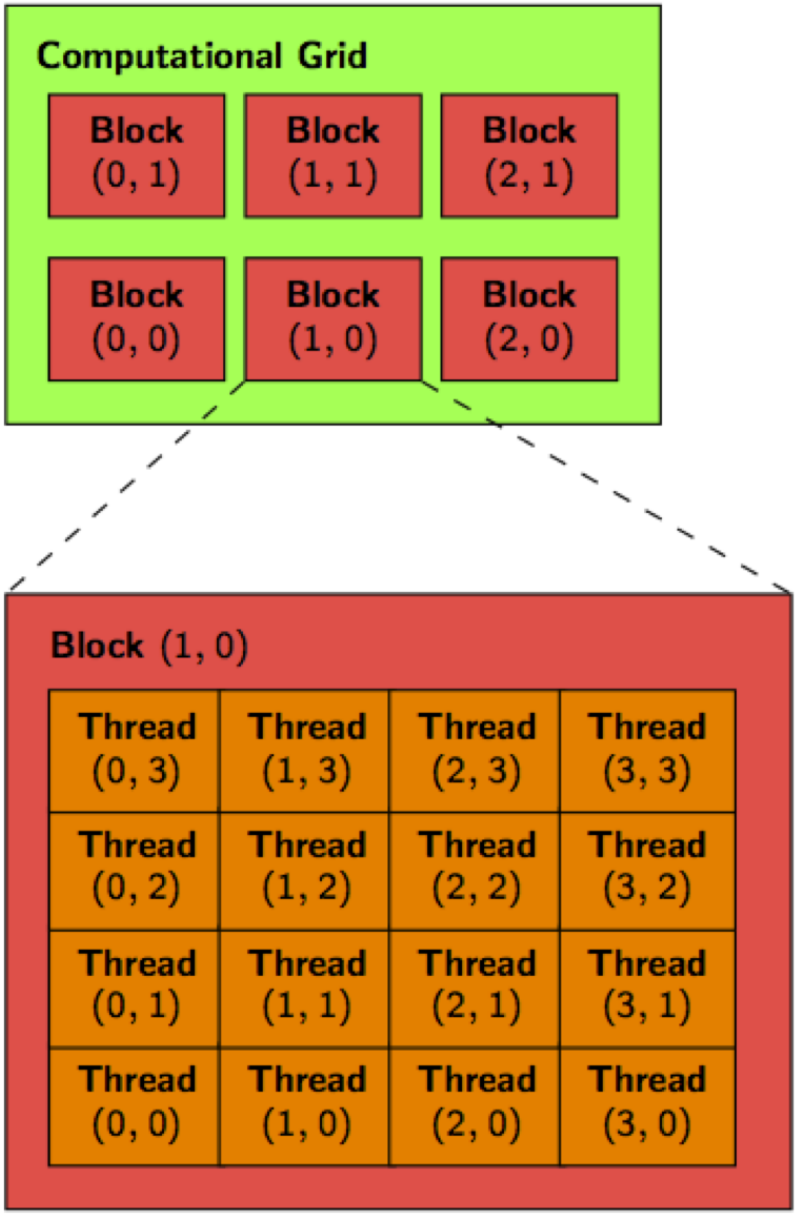
# Abstraction logique - grid

- 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)

**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`)

# Grid




ÉCOLE POLYTECHNIQUE

# Principe de l'algorithme (naif) CUDA

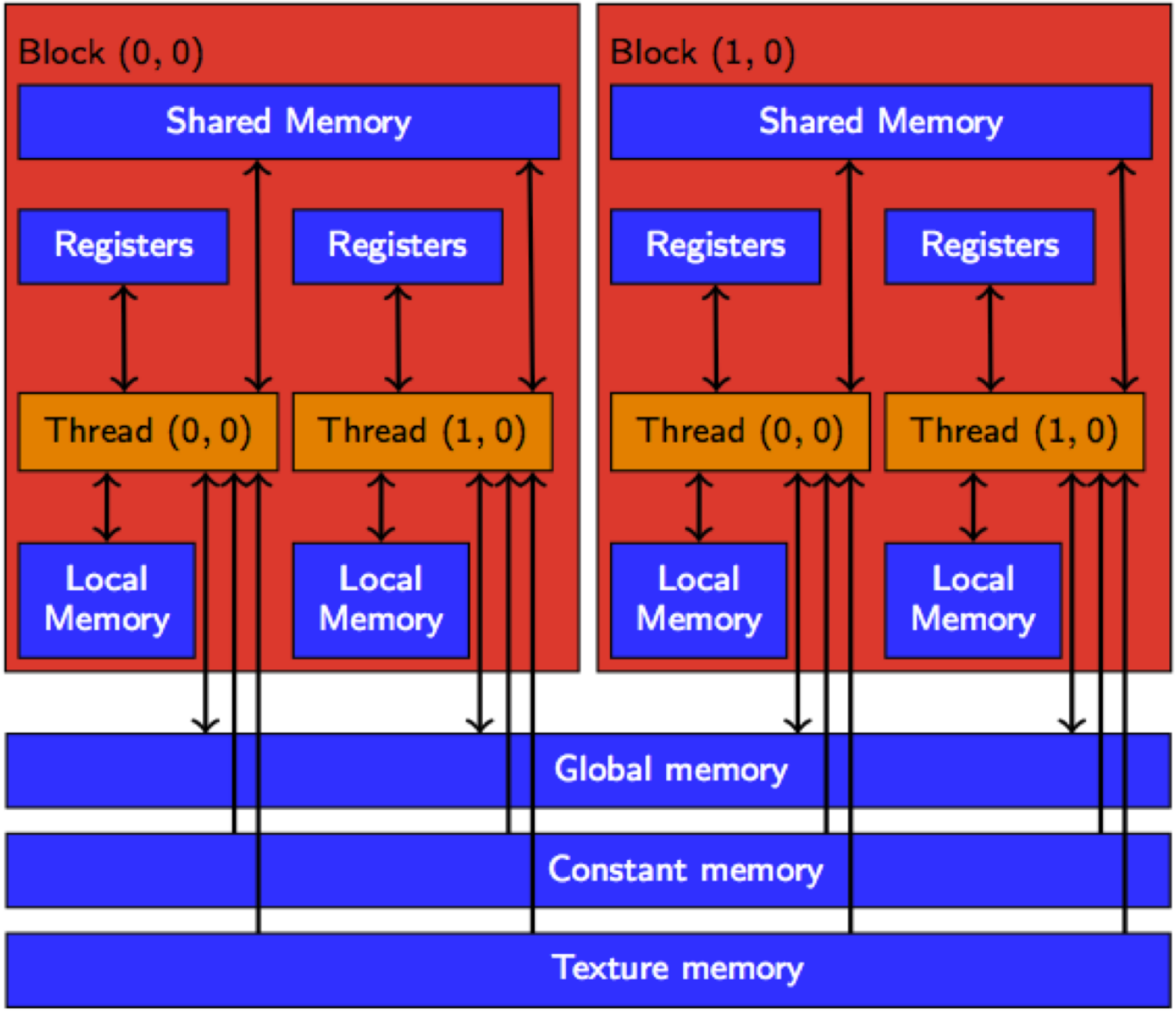
- Limitée à des tableaux de 512 éléments (nombre de threads maxi sur un multi-processeur) - cf. `scan` de la SDK (un peu plus compliqué...)

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

- Ici, 1 thread processor par instance de boucle, dans une grid (1,1,1), de blocs unidimensionnels
- En fait, il faut passer à des plus gros tableaux et à une optimisation plus fine pour avoir de bonnes performances...

ÉCOLE POLYTECHNIQUE

# 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 ($\sim$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

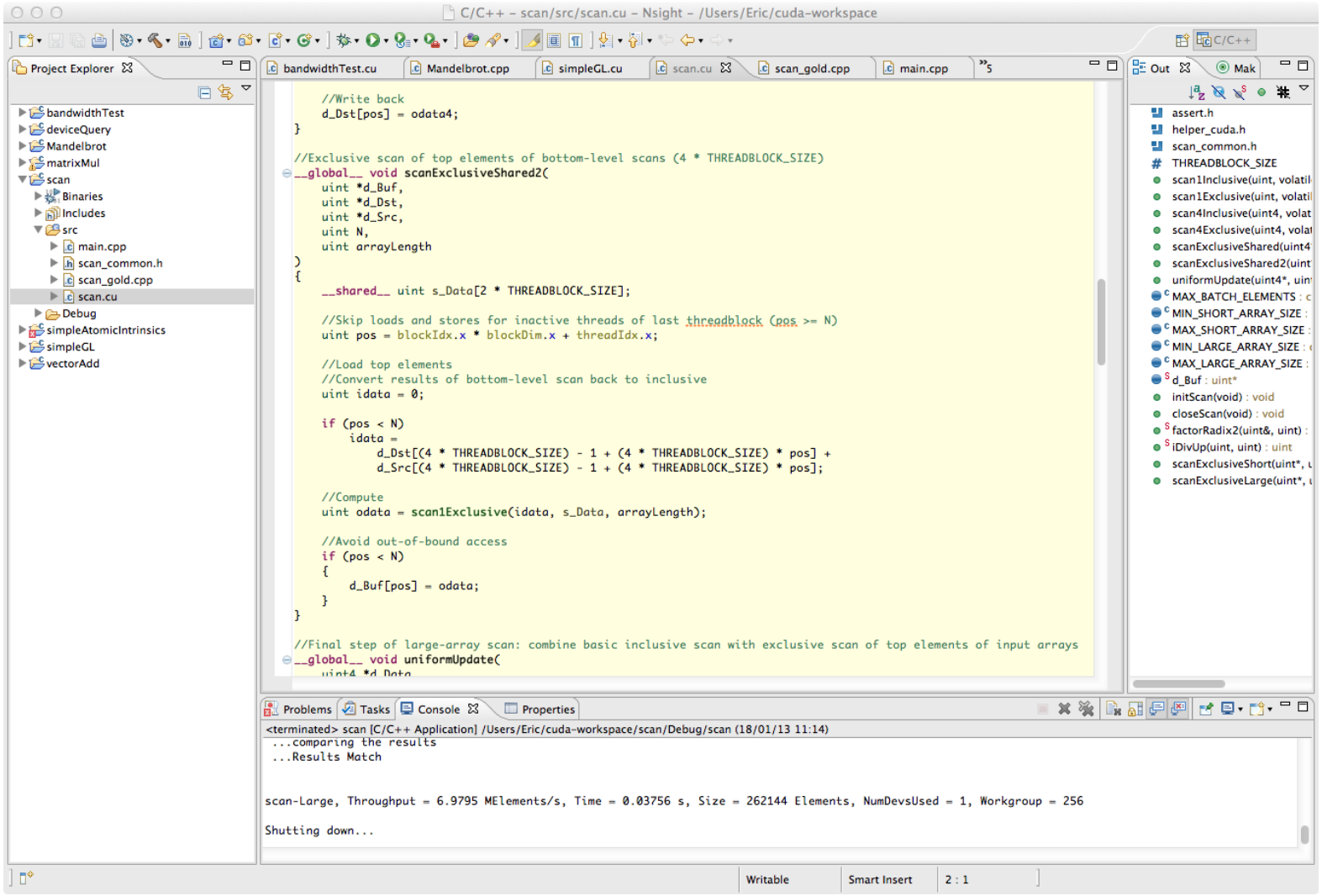
(scan.cu)

```
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



# Remarque: exemple de réseau de tri en CUDA