Programmation des architectures parallèles

PIERRE-ANDRE WACRENIER

Table des matières

Introduction

Ou trouve-t-on du parallélisme ?
Au niveau : des circuits, des données, des instructions, des threads, des grappes de machines, des nuages.

Il faut extraire suffisamment de parallélisme :

- Pour occuper toutes les unités de calcul
- Comment décomposer le calcul pour l'exécuter le plus rapidement possible ?
  - Gros gain : efficacité des unités de calcul
  - Parallélisme : gain fin
  - Efficacité des unités de calcul ; surcout ? parallélisme
- Et utiliser correctement la mémoire
  - Ne pas être bêtement limité par la bande passante

Est-il facile d'obtenir les performances crête ?

C'est difficile, même dans les cas faciles
Multiplication de matrices à l'aide de bibliothèques.

Loi d'Amdahl : accélération = speedup $= \frac{temps\ du\ meilleur\ prog\ séquentiel}{temps\ du\ meilleur\ prog\ parallèle}$

Soit S la part séquentielle (le chemin critique non parallélisable) de l'application A pour une donnée d.
L'accélération de l'exécution de $A(d)$ sur une machine à p processeurs est bornée par :

$$speedup(p) \leq \frac{1}{S + \frac{1-S}{p}}$$

« Si 1% de l'application est séquentielle on n'arrivera pas à aller 100 fois plus vite »

Nos objectifs

- Se plonger dans des environnements de programmation parallèle
  - Maitriser OpenMP
  - S'initier à MPI et OpenCL

Approche mémoire partagée

Programmation (OpenMP, pthreads)

- Une API pour la programmation parallèle en mémoire partagée
  - C, C++, Fortran
  - Portable

    - Porté par l'*architecture review board*
- Basée sur des annotations : #pragma omp directive
- Et des fonctions : omp_fonction()
- Permet de paralléliser un code de façon plus ou moins intrusive
    - Plus on en dit plus on a de performances
    - Facile à mettre en œuvre par un non spécialiste
- Ne permet pas de créer ses propres outils de synchronisation
- https://computing.llnl.gov/tutorials/openMP

Le programme suivant écrit avec OpenMP :

```
#include <omp.h>

int main() {
	#pragma omp parallel
	printf("bonjour\n");

	printf("au revoir\n");
	return EXIT_SUCCESS;
}
/*
gcc -fopenmp bon.c
./a.out
>bonjour
>bonjour
>bonjour
>bonjour
>au revoir
*/
```

Et sa traduction en programme C avec les threads :

```
void fun() {
	printf("bonjour\n");
}

int main() {
	pthread_t tid[N];
	for (int i= 0; i < N; ++i)
		pthread_create(&tid[i], NULL, fun, NULL);
	printf("bonjour\n");
	for (int i= 0; i < N; ++i)
		pthread_join(tid[i], NULL);
	printf("au revoir\n");
	return EXIT_SUCCESS;
}
```

Parallélisme fork-join

- Un unique thread exécute séquentiellement le code de main.
- Lors de la rencontre d'un bloc parallèle, tout thread :
    - Crée une équipe de threads
    - … A COMPLETER

Pour faire un for parallèle, il faut utiliser les deux directives :
#pragma omp parallel
#pragma omp for

Programmer plusieurs appels d'une fonction sur un même élément

$$out[i] = f^k(in, i)$$

```
#pragma omp parallel private (etape)
for (etape = 1; etape < X; ++etape) {
	#pragma omp for
	for (int i = 0; i < N; ++i) {
		out[i] = f(in, i);
	}
	// barrière de synchronisation implicite
	//if (omp_get_thread_num() == 0)
	#pragma omp master
		memcpy(out, in N);
	#pragma omp barrier
}
```

Ou une version un peu différente

```
#pragma omp parallel private (etape)
for (etape = 1; etape < X; ++etape) {
	#pragma omp for
	for (int i = 0; i < N; ++i) {
		out[i] = f(in, i);
	}
	// barrière de synchronisation implicite
	//if (omp_get_thread_num() == 0)
	#pragma omp single
		memcpy(out, in N);
}
```

Voire avec remplacement sur le memcpy()

```
#pragma omp parallel private (etape)
for (etape = 1; etape < X; ++etape) {
	#pragma omp for
	for (int i = 0; i < N; ++i) {
		out[i] = f(in, i);
	}
	// barrière de synchronisation implicite
	//if (omp_get_thread_num() == 0)
	#pragma omp single
		swap(&out, &in);
}
```

Et même avec moins de barrières de synchronisation

```
#pragma omp parallel private (etape) firstprivate(in, out)
for (etape = 1; etape < X; ++etape) {
	#pragma omp for
	for (int i = 0; i < N; ++i) {
		out[i] = f(in, i);
	}
	swap(&out, &in);
}
```

Equilibrage de charge

La loi de Amdahl nous dit qu'un programme parallélisé ne gagnera pas forcément autant de vitesse que de fois qu'il sera parallélisé.

Dans le cas d'un problème comme : `int out[i] = f(i);` .
On commence par regarder la complexité de la fonction f.

Une stratégie pour réduire la charge, on va découper le problème en autant de parties qu'on a de threads.

```
#pragma omp parallel for
for (i=0; i<N; ++i)
	out[i] = f(i);
```

Mais si la charge de travail est inégale en fonction du paramètre de la fonction, certains threads auront fini avant les autres.

Il est possible de faire une distribution cyclique, c'est-à-dire que le système découpe en plein de petits blocks le travail à faire (plus de blocks que de threads disponibles). Puis distribue les parties les unes après les autres successivement à chaque thread. T1=P1, T2=P2, T1=P3, T2=P4, T1=P5,T2=P6.

```
#pragma omp parallel for schedule(static, 1)
for (i=0; i<N; ++i)
	out[i] = f(i);
```

On peut changer « static, 1 » par « static, n » pour distribuer le problème par tranche de n éléments.

Dans le cas d'une fonction avec une complexité aléatoire, pas du tout incrémentale ou suivant une fonction (on ne connait pas la charge de travail à l'avance), on dit que le problème est irrégulier.

```
#pragma omp parallel for schedule(dynamic, n)
for (i=0; i<N; ++i)
	out[i] = f(i);
```

Ce code aura pour effet de découper le problème en sous parties, puis à donner une sous partie à chaque thread libre. Donc chaque thread prendra du travail tant qu'il en reste et ne passera jamais de temps libre tant qu'il restera des choses à faire.

```
// version avce des threads
int guichet()
{
	static int i = 0;
	pthread_mutex_lock(&m);
	int r = i++;
	pthread_mutex_unlock(&m);
	return r;
}

while ((i= guichet())<k)
	f(i);
```

En c on utilisera des instructions atomiques pour éviter les mutex coûteux.

Exemple :
On a une fonction f() et on veut calculer $\sum f(i)$ en fonction de :
Si la complexité est constante (distribution un morceau par thread)

```
#pragma omp parallel for schedule(static)
for (i=0; i<N; ++i) {
	#pragma omp atomic
	out[i] = f(i);
}
```

En plus optimisé :

```
#pragma omp parallel
{
	int x = 0;
	#prama omp for nowait
	for (i=0; i<N; ++i)
```

```
            x+= f(i);
      #atomic
      s += x;
}
```
Ou en plus court :
```
#pragma omp parallel
{
      int x = 0;
      #prama omp for reduction(+:s)
      for (i=0; i<N; ++i)
            s+= f(i);
}
```
Si la complexité est exponentielle (distribution cyclique)
```
#pragma omp parallel for reduction(s:+) schedule(dynamic)
for (i=N-1; i>=0; --i)
      s += f(i);
```
On a inversé le parcours de la boucle, car on sait que la fonction f à une plus grande charge de travail pour les grandes valeurs de i. Donc en commençant avec un grand i, on aura plein de petites charges de travail pour rééquilibrer les threads ayant fini.

Différence entre `critical` et `atomic`

`atomic` ne rend qu'un instruction simple atomique, on ne peut pas rendre atomique deux additions de suite par exemple.

`atomic capture` permet de stocker une variable si elle sera affectée par le calcul courant

`critical` est utile lorsque des variables sont liées. Par exemple incrémentation de deux entiers, on est obligés de le faire en deux opération mais il ne faut pas pouvoir lire l'état de l'objet en cours d'incrémentation.

`critical` est plus puissant que `atomic`.

Problème du voyageur de commerce

Un voyageur doit partir et revenir chez lui en passant par toutes les villes en un temps ou distance minimum. Il faut donc trouver un circuit minimum.

Avec n villes, il existe ! n chemins possibles, donc le graphe de ce problème aura une racine (ville 0 d'origine du voyageur) et ! n feuilles. On va ensuite faire un parcours en profondeur de cet arbre pour obtenir le meilleur chemin.

Code séquentiel du problème :
```
// travelling salesman problem
#define N 5 // nombre de villes
int min = inf; // global
/* etape : profondeur dans l'arbre
 * dist : distance parcourue
 * chemin : chemin parcouru pour y être
 */
int tsp(etape, dist, chemin[]) {
      if (N == etape + 1)
            if (min > dist + d(0, chemin[N-1])) {
                  min = dist;
                  afficher(chemin);
            }
      else {
            for (i=1; i < N; ++i) {
                  if (!present(i, chemin)) {
                        chemin[etape + 1] = i;
```

```
                        tsp(etape+1, dist+d(chamin[etape], i), chemin);
                }
        }
    }
}

// début du programme
tsp(0, 0, [0]); // résoudre le problème
```

Parallélisation du problème

```
int tsp_0() {
    #pragma omp parallel
    int chemin[N];
    #pragma omp for
    for (i=1; i<N; ++i) {
        chemin[0] = ??;
        tsp(1, d[0][i], chemin);
    }
}
int tsp_par(etape, dist, chemin[]) {
    if (N == etape + 1)
        #pragma omp critical
        if (min > dist + d(0, chemin[N-1])) {
            min = dist;
            afficher(chemin);
        }
    else {
        #pragme omp parallel for
        for (i=1; i < N; ++i) {
            if (!present(i, chemin)) {
                chemin[etape + 1] = i;
                tsp(etape+1, dist+d(chamin[etape], i), chemin);
            }
        }
    }
}
```

Efficacité de cette implémentation :

<u>On a une machine a 12 cœurs</u> :
N = 13. Bonne efficacité, car on ne traite pas la ville de départ et chaque thread s'occupera d'un sous arbre. Chaque thread aura la même charge de travail. Efficacité maximale.
N = 6. Bof, avec 7 processeurs qui ne font rien.
N = 14. Mauvais aussi, car un cœur devra traiter deux sous-arbres et le traitement du problème sera aussi long que le traitement de N=25.

Dans ce cas, on se dit qu'on aurait peut-être dû paralléliser au niveau 2.

Autre version avec une création de threads à chaque niveau, puis séquentiel au delà :

```
int tsp_par(etape, dist, chemin[]) {

    if (etape > GRAIN)
        tsp(); // séquentiel
    else { // parallèle
        #pragma omp parallel
        int monChemin[N];
        memcpy(chemin, monChemin);
        #pragma omp for
        for (i=1; i<N; ++i) {
```

```
                chemin[0] = ??;
                tsp_par(1, d[0][i], monChemin);
            }
        }
}
```

Marche bien mais pas très beau, plus de threads que de cœurs sur la machine.

EN OpenMP on peut distribuer un ensemble de boucles en parallélisant au niveau de la fusion des boucles.

```
#pragma omp for collapse(3)
for (...)
    for (...)
        for (...)
```

Pour optimiser le calcul, on peut aussi ajouter la condition, si le chemin courant + le chemin pour revenir au départ est plus long que le plus court en cours de calcul, arrêter l'itération (et le thread). **Main cette optimisation** fait passer d'un problème régulier à un problème irrégulier. Répartir la charge sera plus compliqué.

Problème de producteur – consommateur

Normalement, avec les p_thread, il faudrait :

- Mutex
- FIFO
- Condition (file d'attente pour arrêter les threads si plus de travail)
- Booléen (représente la fin, savoir s'il n'y a plus de travail dans la file parce que pas encore dispo ou parce que plus jamais)

Tâche OpenMP pour faire cela :

```
#pragma omp task
```

Exemple : Hello World !

Sans les threads :

```
int main() {
    // affiche A B mais pas de parallélisme
    #pragma omp task
    printf("A\n");
    #pragma omp task
    printf("B\n");
}
```

Avec 3 threads :

```
int main() {
    // affiche A A B B B A (avec 3 threads)
    #pragma omp thread
    {
        #pragma omp task
        printf("A\n");
        #pragma omp task
        printf("B\n");
    }
}
```

Avec 3 threads et une contrainte de nombres

```
int main() {
    // affiche A B ou B A mais sans connaitre le thread qui a fait quoi
```

```
	#pragma omp thread
	#pragma omp single
	{
		#pragma omp task
		printf("A\n");
		#pragma omp task
		printf("B\n");
	}
}
```

Autre exemple :

```
#pragma omp parallel
#pragma omp single
{
	// trois tâches (calcul x, calcul y, calcul z) mais dans n'importe
quel ordre
	int x, y, z;
	#pragma omp task
	x = f(a);
	#pragma omp task
	y = f(b);
	z = x + y;
}

#pragma omp parallel
#pragma omp single
{
	// trois tâches (calcul x, calcul y, calcul z) x et y absolument
avant z
	int x, y, z;
	#pragma omp task
	x = f(a);
	#pragma omp task
	y = f(b);
	#pragma omp taskwait
	z = x + y;
}

#pragma omp parallel
#pragma omp single
{
	// récupération des valeurs pour le calcul et après celui-ci
	int x, y, z;
	#pragma omp task shared(x) firstprivate(a)
	x = f(a);
	#pragma omp task shared(y) firstprivate(b)
	y = f(b);
	#pragma omp taskwait
	z = x + y;
}
```

Technique pour exprimer un ordre entre les calculs, des dépendances entre les tâches

```
#pragma omp parallel
#pragma omp single
{
	// récupération des valeurs pour le calcul et après celui-ci
	int x, y, z;
	#pragma omp task shared(x) firstprivate(a) depend(out:x)
	x = f(a);
	#pragma omp task shared(y) firstprivate(b) depend(out:y)
```

```
        y = f(b);
        #pragma omp task shared(x, y, z) depend(in:x, y)
        z = x + y;
        #pragma omp taskwait
        printf("Z = %d\n", z);)
}
```

Programmation du calcul de la suite de Fibonacci

$$f_{n+2} = f_{n+1} + f_n$$
$$f_0 = f_1 = 1$$

```
int fibo(int n) {
        if (n < 2)
                return 1;
        // mauvaise version
        return fibo(n - 1) + fibo(n - 2);
        // bonne version
        int x, y;
        #pragma omp task shared(x)
        x = fibo(n-1);
        #pragma omp task shared(y)
        y = fibo(n-2);
        #pragma omp taskwait
        return x+y;
}

#pragma omp parallel
#pragma omp single
printf("%d\n", fibo(12));
```

Parallélisation du traitement des éléments sur un tableau

```
#pragma omp parallel
#pragma omp single
#pragma omp task untied
for (e = first; e != NULL; e->next()) {
        #pragma omp task firstprivate(e)
        traiter(e->elem());
```

Pourquoi ne pas toujours utiliser `untied` ?

- Localité mémoire
- Programmation
    - Section critique
    - Lock
    - N° du thread

Untied = préhemptible A TOUT MOMENT !

Exemple avec tsp :

```
void tsp(int *path, int hops, double len, int mask) {
        if (len + ... >  min)
                return;
        if (hope < grain) {
                for (i=1; i<nbTours; ++i) {
                        #pragma omp task firstprivate(hops,len,mask,i)
shared(path)
                        if (!present(i, ...) {
                                int myPath[N];
```

```
                        memcpy(myPath, path, N * sizeof(int));
                        myPath[hops] = i;
                        tsp(path, hops+1, ...);
                  }
            }
      } else {
            // seq
      }
}
```

Exemple qui marche pas, car le programme peut avoir terminé la fonction et détruit le path au moment où on le copie dans myPath.

```
void tsp(int *path, int hops, double len, int mask) {
      if (len + ... >  min)
            return;
      if (hope < grain) {
            for (i=1; i<nbTours; ++i) {
                  #pragma omp task firstprivate(hops,len,mask,i)
shared(path)
                  if (!present(i, ...) {
                        int myPath[N];
                        memcpy(myPath, path, N * sizeof(int));
                        myPath[hops] = i;
                        tsp(path, hops+1, ...);
                  }
            }
      #pragma omp taskwait
      } else {
            // seq
      }
}
```

Version qui fonctionne, on ne dépile pas avant d'avoir fini les sous-threads.

```
void tsp(int *path, int hops, double len, int mask) {
      if (len + ... >  min)
            return;
      if (hope < grain) {
            for (i=1; i<nbTours; ++i) {
                  if (!present(i, ...) {
                        int *myPath = (int*) malloc(sizeof(int) * N);
                        memcpy(myPath, path, N * sizeof(int));
                        #pragma omp task firstprivate(mypath)
                        myPath[hops] = i;
                        tsp(path, hops+1, ...);
                        free(myPath);
                  }
            }
      } else {
            // seq
      }
}m
```

Autre version avec des malloc()

Infos

Task :

- Tied : la tâche doit être exécutée par le thread qui l'a lancée, untied : tâche peut être reprise par un autre thread, mais est du coup préhemptible (changer plusieurs fois de therads pendant l'exécution). Untied permet plus de performances mais ne permet plus de mutex.
- Final : en dessous d'un grain, ne plus créer de parallélisme

Taskloop :
Remplace *parallel single* suivi d'une boucle *for*. Par *taskloop* suivi d'une boucle *for*.

- Grainsize : donne le nombre d'indices répartis par threads

Architecture (pipeline, cache, SMP, NUMA)

Il y a deux types de parallélisme : **ILP Instruction Level Parallelism – TLP Thread Level Parallelism**

ILP : orienté latence, TLP : orienté débit.

ILP Instruction Level Parallelism

Pipeline :

- processeur CISC : add -> 4 cycles, mul -> div -> 57 cycles.
- Processeur MIPS :
    - RISC : Reduced Instruction Set, processeur a jeu d'instruction réduit. Chaque instruction est réduite en une suite d'instructions simples. Par exemple, pipeline -> décode -> lire registre -> opération -> store.

Cache-L'intérêt, le processeur parallélise l'application pour nous.
Un autre avantage, accélère la vitesse d'horloge : le nombre d'instructions exécutées par secondes dans le pipeline est égale à la vitesse se son élément le plus lent. On découpe le module lent en plusieurs modules rapides, on accélère le pipeline en l'allongeant.
On peut aussi multiplier les unités lentes pour avoir un pipeline « super scalaire » et faire plusieurs fois la même chose en parallèle.

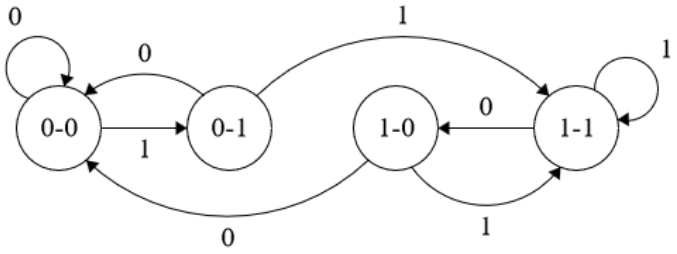
Il y a cependant des problèmes : il y a des *dépendances de données*, des *dépendances de contrôle* et des *disponibilités des* données pour rendre l'application parallèle. Une instruction a besoin d'un résultat pas encore calculé.
Cela crée des bulles dans le pipeline, des endroits vides qui ne calculent plus en attendant un résultat.

On peut ajouter des circuits pour diminuer l'effet des bulles.
Pour les dépendances de données, il est possible de mettre en place un circuit qui va donner une valeur calculée a un nœud plus tôt qui en a besoin avant de la stocker dans un registre (économie d'un store, d'un read et d'une bulle).

Il y a aussi l'éffet *Out Of Order* qui permet à des instructions de passer devant d'autres pour éviter des bulles. Une instruction se fait avant une autre pour remplir une bulle et gagner du temps. On peut dans ce cas mettre des barrières pour empêcher cette intervention de données. Des circuits sont consacrées à savoir si deux instructions peuvent se dépasser ou pas.

Il y a aussi des circuits de renommage, qui vont renommer des registres pour les exécuter dans le désordre. Il pourra par exemple dupliquer un registre pour garder sa valeur dans le cas d'une instruction en ayant besoin et bloquant une autre pour la doubler.

Il y a encore d'autres circuits qui font de l'*exécution spéculative*. Dans le cas d'une condition, le processeur calcule la suite (fait un choix entre les deux possibilités ou les deux en même temps) puis une fois l'évaluation du test finie, il a soit déjà gagné du temps, soit repart de zéro sans rien avoir perdu. Tant qu'aucun résultat n'est écrit, on pourra toujours revenir en arrière.
Cette exécution a une notion de pari, et l'application est d'autant plus performante qu'on a fait un bon pari. Différentes stratégies : *pari* statique toujours prendre le code du if (pas du else), etc… Le programmeur peut donc « guider » le processeur dans sa prédiction.
Les processeurs modernes ont une stratégie plus performante avec une table de hachage dans laquelle on stocke les précédents sauts (adresse du saut, le saut a été utile). Et faire une stratégie « comme la dernière fois » avec 1bit si la dernière fois était bonne ou inverse de la dernière fois si elle était mauvaise. Avec 2 bits on peut avoir une stratégie « comme d'habitude ».



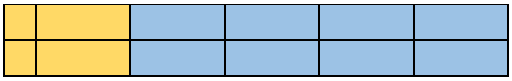
Registres

Le temps d'accès à un registre est de 1 cycle. Pas de création de bulles pour l'accès.
Le temps d'accès à une donnée en cache L1 est de 4 cycles. (32k) Pas de création de bulles pour l'accès.
Le temps d'accès à une donnée en cache L2 est de 12 cycles. (256k) Plus gros mais plus lent, génération de bulles.
Le temps d'accès à une donnée en cache L3 est de 30 cycles. (8Mb) Encore plus gros mais plus lent, plus grosses bulles.
Le temps d'accès à une donnée dans la RAM est de 200 cycles.

Un cache doit pouvoir répondre à deux questions : est-ce que tu possèdes la donnée ? et si oui, quelle est la valeur de la donnée ?

- Observations
  - Plus une mémoire est petite, plus son temps d'accès est rapide.
  - Localité spatio-temporelle des programmes
- Contient une copie d'une faible partie de la mémoire
  - Cache hit / cache miss
  - Nécessite l'éviction de données, la synchronisation avec la mémoire
  - Il faut éviter les aller-retours mémoire/cache (pingpong)
- Mécanisme de **LRU** (Least Recent Used)
  - Supprime les données utilisées il y a le plus longtemps

Cache direct :



En Jaune la clé (adresse de la ligne) : la clé est composé, les bits de poids faible sont utilisés pour connaitre l'octet voulu, ceux de poids intermédiaire pour connaitre le numéro de la ligne dans la tabe.
En bleu les 4 octets de donnée

Ce cache fait du LRU, mais seulement sur une partie de l'adresse.

Cache k-associatif :

La même chose que le cache direct, sauf que le cache est coupé en plusieurs parties. Chaque morceau cherche la donnée et si elle existe, un seul la connait. Cela permet la recherche de données en parallèle.

Cache associatif :

Proche du cache associatif, mais ne contient qu'une ligne par partie de cache.

Une ligne de cache : fait en général 64 octets.

Exemple d'utilisation des caches :

- Cache associatif : TLB
- Cache k-associatif : L1, L2, L3, Branch Target Buffer

Un cache 2-associatif de taille T est aussi performant qu'un cache direct de taille 2T.

Cache – Optimisation des programmes

Parcours séquentiel d'un tableau

```
#define N 8192
typedef ling matrix[N][N];

for (i = 0; i < N; ++i)
      for (j = 0; i < N; ++j)
            C[i][j] = A[i][j] + B[i][j];
```

Avec la boucle i avant la boucle j, on optimise le cache, le programme est optimisé. On utilise le cache comme une fenêtre glissante sur la mémoire, on utilise une ligne de cache puis la suivante. On fait varier les bits de poids faible du cache.
Si on inverse i et j, alors le cache ne sera plus optimisé. On fait varier les bits de poids fort du cache. 1 défaut de cache par lecture + saturation de la TLB.

Comment bien programmer des boucles avec le cache :

- Minimiser le nombre de défaut de cache
    - Réutiliser les données chargées dans le cache
        - Adapter les structures de données (listes chainées = le mal)
        - Aligner les données (aligner les données avec la mémoire, ne pas charger une donnée de 32 bits sur deux lignes avec 16 bits par ligne)
- Limiter le nombre de pages fréquemment accédées
    - Eviter les défauts de TLB
    - Huge page de linux
- Techniques de pavage (tiling)
    - ```
      for (I =; ...; I+=T)
            for (J =; ...; J+=T)
                  for (i = I; ...; i++)
                        for (j = J; ...; j++)
      ```

- deux styles de programmation adaptés aux caches :
    - algorithmes *cache conscious*, tenant compte de la taille du cache
    - algorithmes *cache oblivious*, pensés pour optimiser la localité du travail (divide & conquer)
- les scientifiques utilisent des bibliothèques spécialisées caches-conscious les « basic linear algebra system » BLAS.
      ```

        - BLAS1 : vecteur mémoire O(N) pour O(N) opérations.
        - BLAS2 : vecteur matrice mémoire O(N^2) pour O(N^2) opérations
        - BLAS3 : matrice matrice mémoire O(N^2) pour O(N^3) opérations
    - Les BLAS de niveau 3 permettent de mieux exploiter les caches : on peut obtenir des facteurs 10 en performance
    - Ramener le problème à des opérations matrice/matrice quitte à faire plus d'opérations
    - Facilite la portabilité des performances

TLP Thread Level Parallelism

Prefetching et multithreading au niveau du pipeline.

On peut soit changer de contexte rapidement (architecture a gros grain) lors de défauts de cache. On peut aussi mélanger les instructions dans le pipeline (architecture a grain fin).

Multithreading CMT – FMT – SMP

- Avantages
    - Meilleure utilisation du pipeline
    - Réactivité (lock)
    - Partage de données entre 2 threads
    - Régulation de charge
- Inconvénients
    - Partage de cache -> perte de performance
    - Gestion des priorités
    - Difficultés de mise en œuvre des techniques de scrutation
- Solutions
    - Désactivation du multithreading
    - Appariement de threads compatibles

Machine à mémoire partagée

- SMP (le passé)
    - Mémoire centralisée, réseau en étoile autour de la mémoire (processeurs – interface – mémoire)
- NUMA (maintenant) [Non Uniform Memory Architecture]
    - Mémoire répartie, brique contenant le processeur et une mémoire (chaque processeur amène sa mémoire)
    - Il y a aussi un bus de données entre les mémoires des processeurs.

SMP Synchronisation de la mémoire

- Une même donnée peut être à la fois :
    - En mémoire
    - Dans un cache, dans des caches
    - Dans un registre, dans des registres
- Quelle est la bonne version ?
    - Faire en sorte qu'il n'y ait qu'une version
        - Seulement aux yeux du programmeur
        - Donner au programmeur la même vision de la mémoire que sur une machine monoprocesseur programmée à l'aide de threads (exécutés en séquence)

- Consistance séquentielle (Lamport 1979), sémantique de l'entrelacement
  Un multiprocesseur est séquentiellement consistant si toute exécution résulte d'un entrelacement des séquences d'instructions exécutées par les processeurs et qui préserve chacune des séquences. En particulier tous les processeurs voient les écritures dans le même ordre.
- Exemple : au départ { x = 0, y = 0 } on lance deux threads :
  - T1 { x = 1 ; print y ; }
  - T2 { y = 1 ; print x ; }
  - On ne devrait pas voir { 0, 0 }
    Utilisation de barrières mémoire (vider le pipeline) pour attendre que tous les résultats soient écrites.

Implémentation de la cohérence séquentielle

Implémenté grâce au protocole MSI (Modified Shared Invalid)

- Optimisation de MSI
  - Ajout de l'état Exclusive
  - Seul détenteur d'une copie propre
  - Permet une transition silencieuse si variable non partagée
- Nécessite de savoir si la valeur provient d'un cache ou de la mémoire
  - BusS : si la variable vient d'un cache
  - !BusS : la variable vient de la mémoire

Instructions atomiques

- Protéger une unique opération sans prendre de mutex
  - Permet d'assurer la cohérence séquentielle d'une donnée chargée dans un registre le temps d'un calcul élémentaire
  - Enchainement atomique « lecture L1 – opération – écriture L1 »
  - Concerne des objets de quelques octets
- Réalisation matérielles
  - Bloquer le bus
  - Passer en *modified* et conserver la donnée jusqu'à l'écriture (bloque le cache)
  - Passer en *modified* et observer si la donnée est toujours là au moment de l'écriture (ne bloque pas le cache).
- Builtin gcc (<stdatomic.h>)
  - `type __sync_fetch_and_add(type *ptr, type value, …)`
  - `type __sync_val_compare_and_swap(type *ptr, type oldval, type newval, …)`
  - `...`

⚠ Ne remplace pas les mutex ⚠

Le problème du faux partage

- 2 variables indépendantes peuve,t être sur la même ligne de cache
  ```
  int x, y;
  ```
  Cette ligne de cache peut faire du ping-pong entre deux processeurs.

```
int Tab[nb_threads];
void *thread_fun(void *p) {
      while (...)
            Tab[id] += 1;
}
```

- Solutions
  - Utiliser des variables locales au thread (allocation dans la pile)
  - Faire du bourage d'octets (padding)
  - Utiliser des directives d'alignement
    int x __atributeè_((__aligned__(64))); (void C11)

SMP conclusion

- Machines relativement faciles a programmer
  - Comme en séquentiel grâce au protocole MESI
- Mémoire centralisée = goulet d'étranglement
  - Contention, latence mémoire importante
    - Limitation des performances par le débit mémoire
    - Limitation du nombre de processeurs par le débit
  - IL FAUT DONC OPTIMISER L4UTILISATION DU CACHE
    - Cold miss : premier accès à une variable
    - Capacity miss : le cache est complet. Travailler sur la localité (pavage)
    - Conflict miss :la cache n'est pas complet mais son associativité est trop faible. Revoir l'alignement
    - True sharing miss: le cache défaut nécessaire à la communication entre les threads. Limiter les synchronisations
    - False sharing miss : défaut inutile. Revoir l'alignement.
- Programmation lock-free (via les opérations atomiques)
  - The art of multiprocessor programming, par M. Herlithy & N. Shavit chez Morgan Kaufman
- L'apparition du multi cœur a condamné cette approche pour les multiprocesseurs.
  - Trop de cœurs par processeur

NUMA non uniform memory access

- Nœud NUMA
  - Briques : processeurs + mémoires
- Réseau de communication inter nœud
  - La latence mémoire dépend de la distance entre le processeur et la mémoire en jeu
- Disponibilité

Outils pour le placement

- Libnuma (linux)
  - Placement des threads
    - `Int sched_setaffinity(pid_t pis, size_t cpusetsize, cpu_set *mask);`
  - Politique de placement mémoire
    - Local, distant, relative
  - Numactl en ligne de commande

- Allocation first touch (IMPORTANT)
    - Allocation paresseuse des pages
    - Placement réalisé à la première utilisation
        - Suivant la politique définie
        - Par défaut la page est placée sur le nœud numa du cœur ayant causé l'allocation
    - Nécessite la connaissance de l'architecture pour faire des placements optimaux

Placement OpenMP

- OpenMP
    - Variable OMP_PLACES pour décrire l'architecture
        - OMP_PLACES = `{cpu-id de la 1ere place}, {cpu-id de la 1ère place}…`
    - Clause **proc_bind(master|close|spread)** de la directive parallel
- Exemple
    - Créer 2 équipes de threads chacune sur 1 processeur :
      OMP_PLACES = `{0,1,2,3,4,5},{6,7,8,9,10,11}` ou bien `{0:6},{6:6}` ou `{0:6} :2 :6`
      omp_set_nested(1);
      #pragma OMP PARALLEL num_threads(2) proc_bind_(spread)
      #pragma OMP PARALLEL num_threads62) proc_bind_(master)

Machine tesla

- OMP_PLACES="threads"
- OMP_PLACES="cores"
- OMP_PLACES="sockets"
- OMP_PLACES="cores(6)"

Stratégie Placement thread / mémoire

- Stratégie de placement thread / mémoire
    - Déterminer les dimensions ou l'application à le comportement le plus régulier possible
    - Découper le travail en tâches équilibrées et les affecter de façon statique aux threads
    - Réaliser l'allocation physique des pages
        - Boucle d'initialisation a blanc, réalisé par chaque thread (first touch)
    - Lancement de l'application
- Commentaires
    - Stratégie bien comprise par les programmeurs OpenMP un peu expérimentés
    - Ne s'applique pas aux problèmes irréguliers
        - « move on next touch »
    - Ne tient pas compte de la charge mémoire des nœuds
        - Ni de la consommation de la bande passante

MESI sur Opteron & Nehalem

MOESI / MESIF

- **O** – Owned :
    - La ligne de cache est valide
    - Cette ligne peut être partagée par d'autres caches qui sont dans l'état Shared
    - La valeur contenue par la mémoire principale est incorrecte
    - Ce cache est responsable de l'écriture en mémoire de la ligne

- **F** – Forwarding :
    - La ligne de cache est vide et inchangée par rapport à la mémoire
    - Cette ligne est peut-être partagée par d'autres caches qui eux sont dans l'état Shared
    - Ce cache est responsable de la transmission de la ligne aux autres caches

Conclusion NUMA

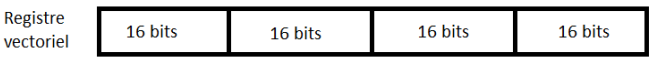
- Le tournevis
    - Solution universelle et performante
    - Maus coûteuse en temps de développement et en matière grise
        - Cercle de plus en plus restreint d'experts
- Portabilité des performances
    - Utiliser des bibliotheques développées per des experts
- ……..

Conclusion architecture, Que faire du silicium ?

- ILP
    - Améliorer le rendement du pipeline
        - Augmentation de la fenêtre d'exclusion spéculative
        - Amélioration del'apprentissage
        - Augmentation de la consommation d'énergie pour un gain faible
    - Approche orthogonale
        - Architecture EPIC (itanium) : le programmeur / compilateur code les dépendances entre les instructions
        - Succès des processeurs *in order* dans le domaine de l'embarqué (ARM) et des accélérateurs
- Multi cœurs
    - Multiplier les cœurs encore et encore
    - Développement des architectures hétérogènes
        - Quelques gros cœurs pour l'ILP
        - Beaucoup de petits cœurs pour le TLP
- Quel avenir pour l'approche mémoire commune ?
    - Bugs difficiles
    - Performances pénalisées par la cohérence séquentielle
    - Mémoire spécialisée (latence bs débit)

Retour sur le parallélisme dans les cœurs

- Pipeline, processeur super scalaire, Hyper-Threading -> on multiplie les registres (dont le compteur ordinal) et on pioche les instructions depuis plusieurs flots indépendants. Le souci est que le cache est partagé par les threads d'un même cœur. Du coup, on ne sait pas comment les différents threads de chaque processus utilisent la mémoire (50/50, 95/5).
- En 1996, Intel sort le Pentium "MMX" qui permet de décoder les vidéos et le son plus rapidement.
    - Technologie "vectorielle" : grands registres qui fonctionnent comme des tableaux de registres. Chez intel, on peut soit utiliser un « gros » registre comme un gros registre ou comme un tableau de plus petits registres.



ces registres vectoriels permettent de faire des opérations dessus plus rapidement, un vecteur 4 + un vecteur 4 = 4 fois une addition et pas 1 addition 4 fois.

- 1999 : Intel sort les instructions SSE "Streaming SIMD Extension" "Simple Instruction Multiple Data" et passe à 128 bits pour les registres vectoriels.
- Aujourd'hui, les derniers processeurs Intel utilisent AVX "Advanced Vector Instruction" qui utilisent des registres 512 bits. Soit 16 floats ou 8 doubles.

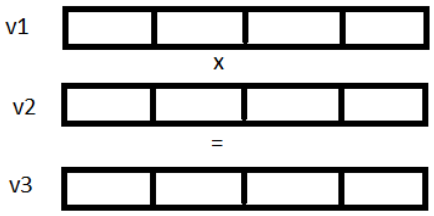
Exemple de calcul vectoriel :

```
float v1[N], v2[N], v3[N];
v3 = k * v1 + v2; // k : un scalaire

// version C
for (int i = 0;i < N; i++)
      V3[i] = k * v1[i] + v2[i];

// version asm
__asm__(
      "movss %1, %%xmm1" // %1 : k | mettre k a la fin d'un vecteur de 4
éléments
      "shufps $0, %%xmm1, %%xmm1" // mettre k dans tout le vecteur
      "boucle:"
            "movaps v1(%%rbx), %%xmm0"
            "mulps %%xmm1, %%xmm0"
            "addps v2(%%rbx), %%xmm0"
            "movaps %%xmm0, v3(%%rbx)"
            "addq $16, %%rbx"
            "cmpq %%rcx, %%rbx" // %%rcx contient n * sizeof(float)
      "jne boucle"
);

// version avec librairie qui va bien
#include "xmmintrin.h" // pour utiliser les fonctions intrinsiques
assembleur en C
__m128 valK = _mm_set1_ps(k); // vecteur de 128 bits contenant plein de
fois k
for (int i = 0; i < N; i += 4) {
      __m128 val1 = _mm_load_ps(&v1[i]);
      __m128 val2 = _mm_load_ps(&v2[i]);
      __m128 val3 = _mm_mul_ps(val1, valK);
      _mm_store_ps(&v3[i], val3);
}
```



Tests de ces versions sur un Intel Xeon Phi :

	-O0	*-O1*	*-O2*	*-O3*
C	2550	800	800	500
ASM	390	390	390	390
INTRINSICS	2090	390	390	390

Dans le cas d'un calcul avec une condition dans une boucle, le compilateur a plusieurs options :

- Si la condition est simple (if else) le compilateur peut faire deux calculs vectoriels (cas ou if vrai et cas ou if false) puis ne récupère que les cases des deux vecteurs de sortie qui correspondent. Ou alors remplir un nouveau vecteur avec une opération capable de prendre une valeur sur N.
- Si la condition est trop complexe, alors le compilateur traite la boucle en séquentiel.

Dans le cas de tableaux de taille non multiple de la taille d'un vecteur, alors le traitement des derniers éléments se font en séquentiel. Il est donc utile que le tableau fasse quelques cases de plus pour vectoriser jusqu'au bout, les instructions vectorielles (en trop) étant gratuites.

Il est également possible de demander à OMP de paralléliser en gardant la vectorisation du code (option smid).

```
#pragma omp parallel for simd schedule(static,1)
for (i=0; i<N; i++)
      v3[i] = k*v1[i] + v2[i];
```

Dans le cas d'un programme qui contient une référence vers une case pas encore calculée car dans le même vecteur, le compilateur peut vectoriser le code jusqu'à des vecteurs de n éléments

```
for (i=d; i<N; i++)
      v3[i] = k*v3[i-2] + v1[i];
```

Ce code pourra être vectorisé avec un vecteur de taille 2 (à cause du -2 sur les indices de v3).

OMP utilise l'option "safelen(n)" qui permet de définir la longueur du vecteur (safe)

Calcul sur accélérateur

Architecture des GPU / MIC (cartes graphiques)

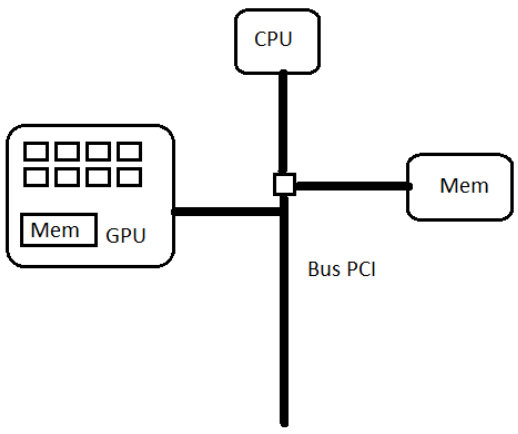
2D, 3D, Voodoo 3dFx

Cartes graphiques programmables (uniquement de petits programmes pour faire de petites actions comme « faire un truc par vertice sur ce modèle » ou « faire cela sur chaque face de ce modèle », etc.)

- DirectX : Langage de programmation pour carte graphique
- OpenCL : petits programmes appelés shaders

GPU (Graphical Processing Unit)
GPGPU (General Purpose Graphical Processing Unit) « carte graphique programmable »

- NVIDIA : a compris qu'il existait un marché avec les cartes graphiques programmables. → Création de CUDA (2003).
    - Bibliothèque de fonctions + langage de programmation pour la carte graphique.
    - Carte graphique dédiée au calcul (famille tesla) « Mémoire ECC (autocorrigée) »
- AMD : tentatives diverses pour proposer des alternatives

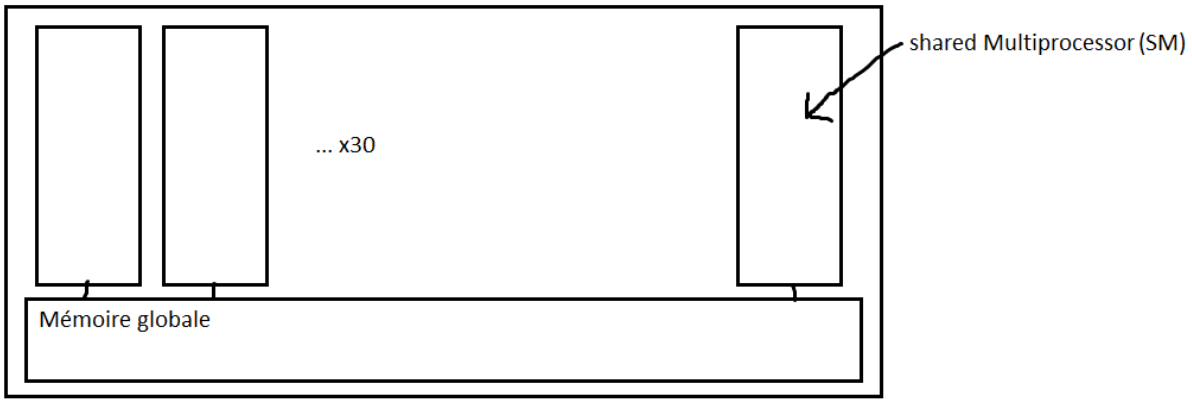


- CTM : Assembleur des cartes graphiques
- Firestream : bibliothèque C++

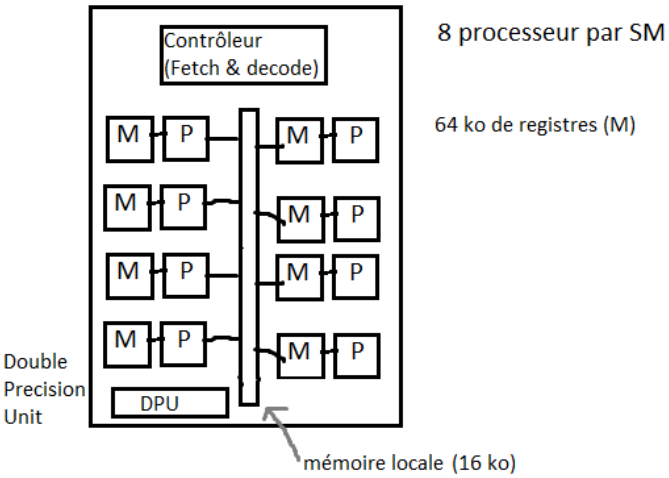
- Consortium Kronos (réfléchir et formaliser)
  - OpenCL : comme CUDA (routines + langage) ouvert, 2008
    Marche aussi sur CPU Intel

Commencer à allouer de la mémoire sur le GPU (sorte de malloc pour GPU)

Exemple d'architecture de carte graphique : NVIDIA GTX 280



Processeur permettant de gérer une grande quantité de threads avec la possibilité de commuter rapidement entre eux. Par exemple, changer de thread lors d'un load car temps d'accès trop long et passer à un autre thread pour travailler sans pause.



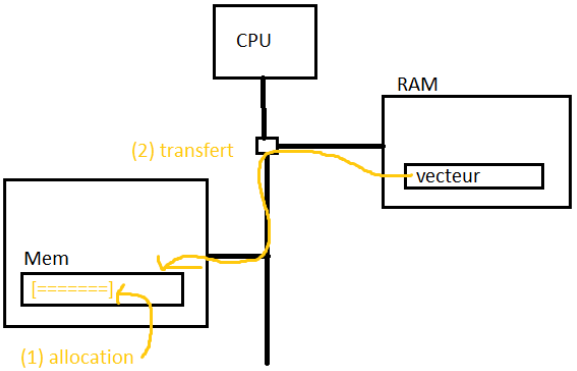
Le contrôleur leur demande de faire les opérations 4 fois car il est trop lent pour décoder les opérations. Du coup chaque SM fait 32 fois la même exécution.

La carte graphique est contente que si elle a au moins 4 threads à calculer à la fois car sinon il y a des espaces sans calcul (blancs). Notion "d'équipe" de 32 threads "warp" pour NVIDIA (et "waveFont" chez AMD, paquet de 64 threads)

Il est possible de placer jusqu'à 128 threads par processeur, soit 1024 threads par SM et 30720 sur toute la carte.

Programmation (OpenCL)

On écrira les programmes dans des fichiers .cl qui seront ensuite copiés dans la carte graphique après compilation OpenCL




```
/// Dans la RAM, vecteur déclaré comme
float vecteur[N];

// __global : dire que le vecteur sera dans la mémoire globale
// on suppose que ce noyau est exécuté avec 1 thread par élément du vecteur
// IL FAUT lancer ce noyau avec N threads
__kernel void vec_mul(__global float *vec, float k)
{
	// code exécuté par chaqu'un des threads de la carte graphique
	int index = get_global_id(0); // numéro du thread courant (appel gratuit)
	vec[index] *= k;
}
```

Si on suppose qu'on exécute le noyau avec seulement $N/2$ threads, alors on ne traitera que la moitié du tableau.

```
__kernel void vec_mul(__global float *vec, float k)
{
	// version qui marche avec N/2 threads
	int index = get_global_id(0);
	vec[index] *= k;
	vec[index + get_global_size(0)] *= k;
}

// version mieux sans chercher les cases trop lointaines (mais pas bonne sur GPU)
__kernel void vec_mul(__global float *vec, float k)
{
	// version qui marche avec N/2 threads
	int index = get_global_id(0);
	vec[2*index] *= k;
	vec[2*index + 1] *= k;
}
```

Pourquoi ne pas respecter le cache est plus performant que bien programmer comme en OpenMP ?

```
// ne multiplier que les éléments dont l'index est pair et mettre les autres a 0
// mauvaise version
__kernel void vec_mul(__global float *vec, float k)
{
	int index = get_global_id(0);
	if (index%2 == 0)
		vcc[index] *= k;
	else
		vcc[index] = 0.0;
}

// bonne version
```

```
__kernel void vec_mul(__global float *vec, float k)
{
	int index = get_global_id(0);
	if (index<get_global_size(0) == 0)
		vcc[index] *= k;
	else
		vcc[index] = 0.0;
}
```

Pour exécuter un noyau OpenCL, il faut préciser :

- Les valeurs des paramètres
- La dimension du problème : 1D, 2D, 3D (façon de numéroter les threads)
- Le nombre de threads à créer sur chaque dimension
- La dimension des "workgroups" (grouper les threads de force, par exemple 64 sur le même SM, pas sur deux SM différents). Les threads peuvent partager de la mémoire locale.

(En OpenCL, les threads se nomment les "work items")

Cas d'un tableau de taille $N = 1024 \times 1024$:

Créer un thread ne coûte rien, elle crée des threads facilement. Elle conserve juste le nombre de threads déjà lancés et ceux qui restent à faire.
Dans le cas où tous les threads figurent sur la carte, CUDA autorise la création d'une barrière globale.
Les threads ne peuvent par ailleurs pas communiquer entre eux.

Comment un thread peut-il faire un malloc dans la mémoire locale ?

Utiliser le mot clé __local devant la déclaration de la mémoire.

La carte graphique pourra cependant refuser la création de nouveaux threads si cette mémoire est trop grande par thread. Donc on peut ne pas remplir le nombre de threads si ceux-ci prennent trop de place en mémoire partagée.

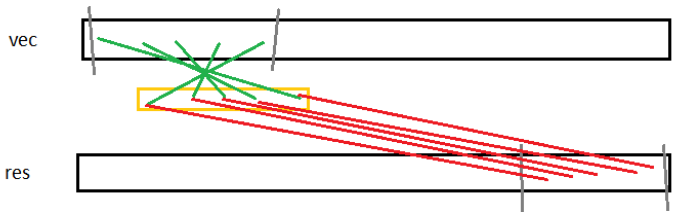
Exemple de programme qui prend un vecteur et le retourne (premier élément devient le dernier et dernier le premier).

```
// fonctionne lentement, car lecture mémoire contigüe mais pas dans le bon
sens
__kernel void reverse(__global float *vec, __global float *res)
{
	int index = get_global_id(0);
	int N = get_global_size(0);
	res[N - index - 1] = vec[index];
}
```

Memo

```
get_num_groups(d); // nombre de groupes
get_group_id(d); // numéro du groupe courant
get_local_id(d); // id local au sein du groupe
get_local_size(d); // combien de personne dans chaque groupe
```

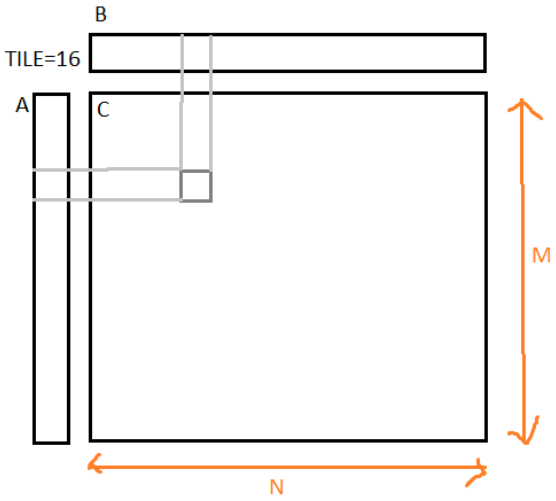
Idée : écrire un petit tableau en mémoire locale pour calculer l'inverse d'une tranche, puis recopier vers res de manière contigüe.




```
// TILE_SIZE calculé au
moment de l'exécution du programme C générenat le OpenGL
__kernel void reverse(__global float *vec, __global float *res)
{
	__local float tile[TILE_SIZE];
	int index = get_global_id(0);
	int iLoc = get_local_id(0); // indice local

	tile[TILE_SIZE - iLoc - 1] = vec[index];
	res[((get_num_groups(0)  get_group_id(0) - 1) * TILE_SIZE) + iLoc] =
tile[iLoc];
}
```

Guide mémoire de chez NVIDIA pour le calcul matriciel




```
// programme qui marche bien mais pas
optimisé
__kernel void matmul(__global float
*a, __global float *b, __global float
*c, int N)
{
	int col = get_global_id(0);
	int row = get_global_id(1);
	float sum = 0.0;
	for (int k = 0; k < TILE; k++)
		sum += a[row*TILE + k] +
b[k*N + col];
	c[row*N + col] = sum;
}

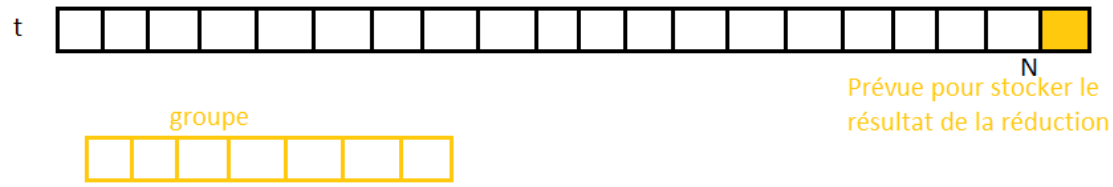
// version optimisée
__kernel void matmul(__global float *a, __global float *b, __global float
*c, int N)
{
	__local float aTile[TILE][TILE];
	int col = get_global_id(0);
	int row = get_global_id(1);
	int x = get_local_id(0);
	int y = get_local_id(1);
	float sum = 0.0;
	aTile[y][x] = a[row*TILE + x];
	// pas besoin de barrière car les accès mémoire sont alignés donc les
variables sont ok
	barrier(CLK_LOCAL_MEM_FENCE); // mais dans le doute il vaut mieux au
cas ou on change d'architecture
	for (int k = 0; k < TILE; k++)
		sum += aTile[y][k] + b[k*N + col];
	c[row*N + col] = sum;
}

// version optimisée ++ (la lecture dans B lit 16 fois chaque case, donc
utile de les stocker en local)
__kernel void matmul(__global float *a, __global float *b, __global float
*c, int N)
```

```
{
	__local float aTile[TILE][TILE];
	__local float bTile[TILE][TILE];
	int col = get_global_id(0);
	int row = get_global_id(1);
	int x = get_local_id(0);
	int y = get_local_id(1);
	float sum = 0.0;
	aTile[y][x] = a[row*TILE + x];
	bTile[y][x] = b[y*N + col];
	barrier(CLK_LOCAL_MEM_FENCE); // besoin d'une barrière car les
lectures dans bTile sont faites dans un autre warp que les écritures
	for (int k = 0; k < TILE; k++)
		sum += aTile[y][k] + bTile[k][x];
	c[row*N + col] = sum;
}
```

Réduction d'un tableau

On peut stocker le résultat de la réduction dans la dernière case du tableau.




```
__kernel void reduction(__global float *t, int N)
{
	__local float tile[TILE];
	int index = get_global_id(0);
	float s = 0.0;
	tile[get_local_id(0)] = t[index];
	index += get_global_size(0);
	while(index < N)
	{
		tile[get_local_id(0)] += t[index];
		index += get_global_size(0);
	}
	barrier(CLK_LOCAL_MEM_FENCE);
	if (get_local_id(0) = 0) // version séquentielle, facile mais le plus
lent
	{
		for (int i = 0; i < TILE; i++)
			s += tile[i];
		t[N] = s;
	}
}

__kernel void reduction(__global float *t, int N)
{
	__local float tile[TILE];
	int index = get_global_id(0);
	float s = 0.0;
	tile[get_local_id(0)] = t[index];
	index += get_global_size(0);
	while(index < N)
	{
		tile[get_local_id(0)] += t[index];
		index += get_global_size(0);
	}
```

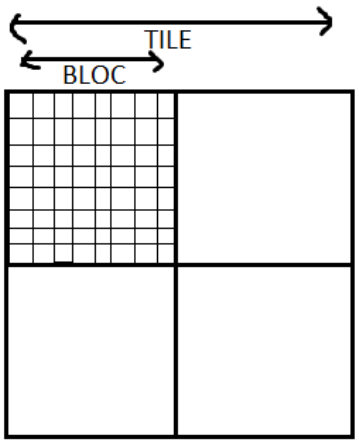
```
	// pas besoin de barrière car elle est en premier tour de boucle,
sinon il y en a besoin
	int limit = TILE >> 1; // version avec threads en parallèle
	for (; get_local_id(0) < limit; limit >>= 1)
	{
		barrier(CLK_LOCAL_MEM_FENCE);
		tile[get_local_id(0)] += tile[get_local_id(0) + limit];
	}
	if (get_local_id(0) == 0) // récupération du résultat
		t[N] ) tile[0];
}
```

Cette version est valide pour un workgroup, mais si on veut aller plus vite on peut prendre plusieurs workgroups. Cependant, les workgroups ne peuvent pas communiquer entre eux ni partager de mémoire locale.

On peut donc refaire la même chose avec plusieurs workgroups sauf que le tableau contiendra plusieurs résultats. Puis une fois le processeur ayant repris la main, il recommence le calcul avec le noyau sur les cases restantes du tableau. Et au final, au bout de n itérations, on obtient le résultat final.

Exemple de "réduction" : la pixellisation d'une image

Idée : Lancer 1thread par pixel ($DIM \times DIM$), groupes de $TILE \times TILES$ threads.




```
__kernel void pixellize(
	__global unsigned *in,
	__global unsigned *out)
{
	__loclal unsigned tile[TILEY][TILEX];
	int x = get_global_id(0);
	int y = get_global_id(1);
	int xLoc = get_local_id(0);
	int yLoc = get_local_id(1);

	// préchargement des données (tous les
threads)
	tile[yLoc][xLoc] = in[y*DIM + x];
	barrier(CLK_LOCAL_MEM_FENCE);

	// boucle sur la distance
	// m : mask
	for (unsigned d = 1, m = 1; d < BLOC; d<<=1)
	{
		if ((xLoc&d) == 0)
		{
			tile[yLoc][xLoc] = color_mean(tile[yLoc][xLoc],
tile[yLoc][xLoc+d]);
			barrier(CLK_LOCAL_MEM_FENCE);
			if (yLoc&d) == 0/* && (xLoc&d) == 0*/)
			{
				tile[yLoc][xLoc] = color_mean(tile[yLoc][xLoc],
tile[yLoc+d][xLoc]);
				barrier(CLK_LOCAL_MEM_FENCE);
			}
		}
		m = (m<<1)|1; // décalage du masque : 001, 011, 111; 1, 3, 5
	}
	// pas besoin de barrière car barrière en sortie du for
	// recolorier toute la tuile de la même couleur
```

```
        unsigned mask = ~(BLOC-1);
        out[y*DIM + x] = tile[xLoc & mask][yLoc & mask];
}
```

Des codes "Stencil"

Ce sont des codes dans lesquels la valeur d'une case évolue en fonction de la valeur des cases adjacentes.

Approche mémoire distribuée

Programmation (MPI)

Architectures (topologie, réseaux rapides)