

Analyse expérimentale du coût et des défauts de localité

Application au produit matriciel

Nicolas Vincent, Adrien Argento

2020-09-23

Q1. Mesures de temps

Le programme (i, j, k) est le plus long à l'exécution.

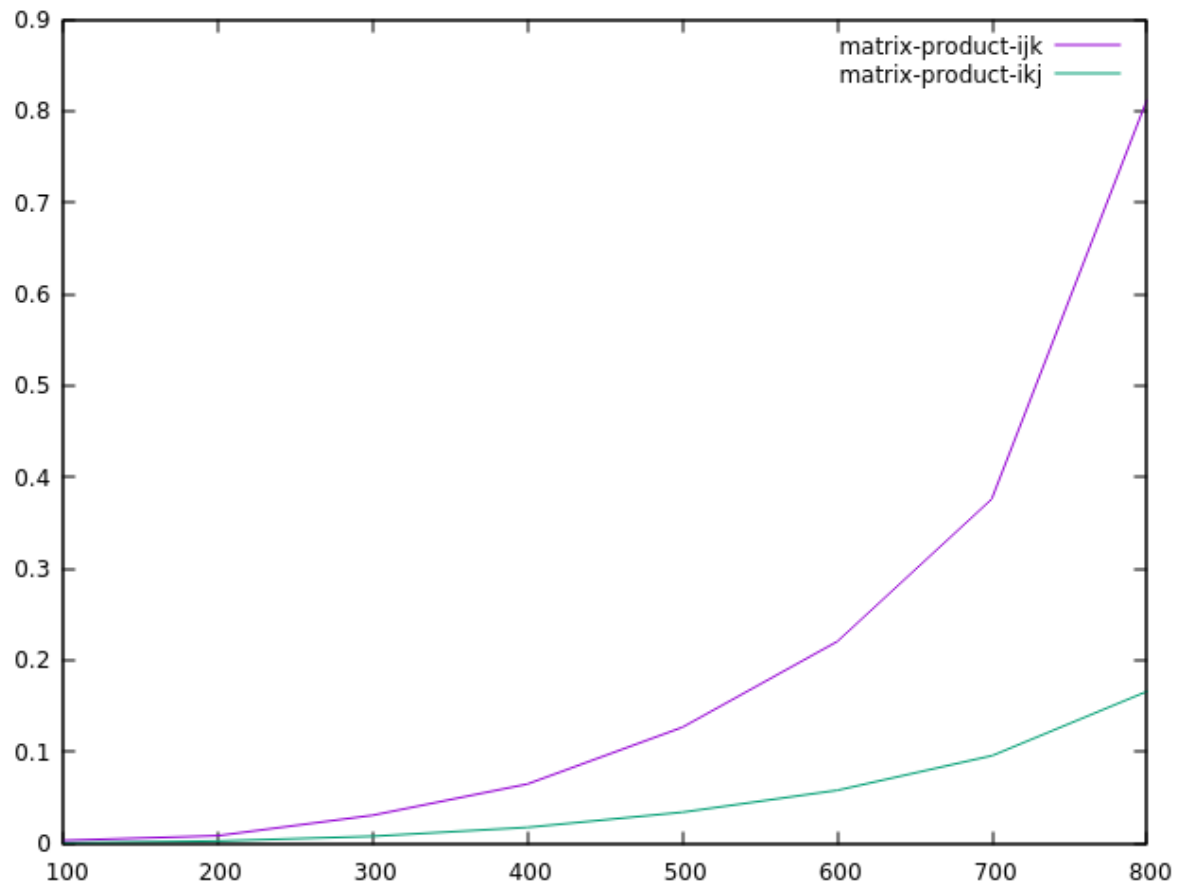


Fig. 1 : Temps d'exécution pour n allant de 100 à 1000 par pas de 100.

Les deux programmes semblent bien en $\Theta(n^3)$.

Q2. Mesures de défauts de cache

valgrind-cachegrind pour la boucle (i, j, k).

```
$valgrind --tool=cachegrind --log-file=valgrind-cachegrindijk-1000.txt ./matrix-product
matrix2d_product_ijk() took 86.578572 seconds to execute for an entry n = 1000

--198803-- warning: L3 cache found, using its data for the LL simulation.
==198803== brk segment overflow in thread #1: can't grow to 0x483b000
==198803== (see section Limitations in user manual)
==198803== NOTE: further instances of this message will not be shown
==198803==
==198803== I   refs:      7,135,382,486
==198803== I1  misses:      1,254
==198803== LLi misses:      1,249
==198803== I1  miss rate:      0.00%
==198803== LLi miss rate:      0.00%
==198803==
==198803== D   refs:      3,063,569,074 (3,044,367,354 rd + 19,201,720 wr)
==198803== D1  misses:      1,253,011,068 (1,251,756,391 rd + 1,254,677 wr)
==198803== LLd misses:      125,759,735 ( 125,380,280 rd + 379,455 wr)
==198803== D1  miss rate:      40.9% ( 41.1% + 6.5% )
==198803== LLd miss rate:      4.1% ( 4.1% + 2.0% )
==198803==
==198803== LL refs:      1,253,012,322 (1,251,757,645 rd + 1,254,677 wr)
==198803== LL misses:      125,760,984 ( 125,381,529 rd + 379,455 wr)
==198803== LL miss rate:      1.2% ( 1.2% + 2.0% )
```

valgrind-cachegrind pour la boucle (i, k, j).

```
$valgrind --tool=cachegrind --log-file=valgrind-cachegrindikj-1000.txt ./matrix-product
matrix2d_product_ikj() took 26.232037 seconds to execute for an entry n = 1000

--199022-- warning: L3 cache found, using its data for the LL simulation.
==199022== brk segment overflow in thread #1: can't grow to 0x483b000
==199022== (see section Limitations in user manual)
==199022== NOTE: further instances of this message will not be shown
==199022==
==199022== I   refs:      4,130,887,438
==199022== I1  misses:      1,256
==199022== LLi misses:      1,251
==199022== I1  miss rate:      0.00%
==199022== LLi miss rate:      0.00%
==199022==
==199022== D   refs:      2,063,569,053 (1,545,367,338 rd + 518,201,715 wr)
==199022== D1  misses:      125,899,531 ( 125,644,851 rd + 254,680 wr)
==199022== LLd misses:      125,896,038 ( 125,641,837 rd + 254,201 wr)
==199022== D1  miss rate:      6.1% ( 8.1% + 0.0% )
==199022== LLd miss rate:      6.1% ( 8.1% + 0.0% )
==199022==
==199022== LL refs:      125,900,787 ( 125,646,107 rd + 254,680 wr)
==199022== LL misses:      125,897,289 ( 125,643,088 rd + 254,201 wr)
==199022== LL miss rate:      2.0% ( 2.2% + 0.0% )
```

Le programme (i, k, j) fait 4,130,887,438 instructions et 125,899,531 défauts de cache L1.

Le programme (i, j, k) fait 7,135,382,486 instructions et 1,253,011,068 défauts de cache L1. C'est 10 fois plus de défauts de cache!!

Comment calculer le temps perdu par le processeur à cause de chaque défaut de cache (pas seulement ceux du cache LL1)?

On a besoin de la fréquence de notre processeur et du nombre d'instructions par cycle. On obtient ces données avec l'utilitaire `perf`.

Pour le programme (i, j, k) :

```
$sudo perf stat -e task-clock,cycles,instructions,cache-references,cache-misses,L1-dcache-loads,L1-dcache-load-misses ./matrix-product
1000 12.058128

Performance counter stats for './matrix-product':

    12096,02 msec task-clock          #    0,995 CPUs utilized
    27655923827 cycles                #    2,286 GHz              (37,19%)

    7195103470 instructions           #    0,26 insn per cycle    (53,84%)
    160614882 cache-references        #   13,278 M/sec            (66,66%)

    127941970 cache-misses            #   79,658 % of all cache refs (83,36%)

    3001572602 L1-dcache-loads         #   248,146 M/sec           (83,31%)
    1507670021 L1-dcache-load-misses   #   50,23% of all L1-dcache hits (33,27%)

    12,151104421 seconds time elapsed

    12,074769000 seconds user
    0,027999000 seconds sys
```

Pour le programme (i, k, j) :

```
$sudo perf stat -e task-clock,cycles,instructions,cache-references,cache-misses,L1-dcache-loads,L1-dcache-load-misses ./matrix-product
1000 1.006262

Performance counter stats for './matrix-product':

    1045,28 msec task-clock          #    0,998 CPUs utilized
    2464033043 cycles                #    2,357 GHz              (35,75%)

    4149347020 instructions           #    1,68 insn per cycle    (52,56%)
    70023951 cache-references        #   66,991 M/sec            (66,59%)

    20188624 cache-misses            #   28,831 % of all cache refs (83,39%)

    1558986613 L1-dcache-loads         #  1491,457 M/sec           (82,69%)
    128811001 L1-dcache-load-misses   #    8,26% of all L1-dcache hits (33,03%)

    1,047782942 seconds time elapsed

    1,030066000 seconds user
    0,016032000 seconds sys
```

The cache-misses event represents the number of memory access that could not be served by any of the cache.

The ratio of cache-misses to instructions will give an indication how well the cache is working; the lower the ratio the better.

The higher IPC (Instruction per clock cycle) the more efficiently the processor is executing instruction on the system. The IPC will be affected by delay due to cache misses.

Par ailleurs, on sait que

$$CPI = CPI_{\text{ideal cache}} + (\text{cache-misses rate}) * (\text{coût d'un défaut de cache})$$

CPI est le nombre de cycles par instructions. `perf` nous donne le nombre d'instructions par cycles.

$CPI_{\text{ideal cache}}$ représente les performances du cache en l'absence de défauts.

Le coût d'un défaut d'un cache est exprimé en cycles perdus par défaut de cache. On le suppose identique entre nos deux simulations.

On a donc

$$\begin{aligned} (\text{coût d'un défaut de cache}) &= \frac{CPI_{ijk} - CPI_{ikj}}{(\text{ijk cache-misses rate}) - (\text{ikj cache-misses rate})} \\ &= \frac{(\frac{1}{0.26} - \frac{1}{1.68})}{(0.79658 - 0.28831)} \\ &= 6.4 \text{ cycles} \end{aligned}$$

Ce qui multiplié à la fréquence du processeur donne environ 15 **ns** comme temps moyen d'un défaut de cache.

Voici les temps d'exécution des deux programmes sans `valgrind` :

```
$/matrix-product
matrix2d_product_ijk() took 8.568009 seconds to execute for an entry n = 1000
```

```
$/matrix-product
matrix2d_product_ikj() took 0.785889 seconds to execute for an entry n = 1000
```

Q3. Analyse du travail en nombre d'opérations

- $W_{\times}(n)$: on fait n^3 multiplications entre `ptA` et `ptB` (trois boucles) pour l'algorithme (i, j, k) . Le raisonnement est analogue pour l'algorithme (i, k, j) .
- $W_{+}(n)$: on fait n^3 additions entre `ptC` et `ptA * ptB` (trois boucles) pour l'algorithme (i, j, k) . Le raisonnement est analogue pour l'algorithme (i, k, j) .
- Le nombre d'additions de pointeurs sur `ptC` est n^2 (deux boucles) et le nombre d'additions de pointeurs sur `ptA` et `ptB` est n^3 (trois boucles, algorithme (i, j, k)). Le nombre d'additions de pointeurs sur `ptA` est n^2 (deux boucles) et le nombre d'additions de pointeurs sur `ptB` et `ptC` est n^3 (trois boucles, algorithme (i, k, j)).
- $W_{ptr+}(n) = 0$ car dans les boucles, on est sur des valeurs pointées et non sur des pointeurs (it's a trap).
- $W_{ptrx}(n) = n^2 + n^3$ car on a n^2 additions et n^3 additions (pour les deux algorithmes).

Les deux programmes effectuent un nombre analogue d'opérations : $O(n^3)$.

Q4. Analyse de l'impact des itérateurs

Si le **compilateur sait** (il le sait généralement) que le foncteur $M(i, j)$ est implémenté par $M(i * n + j)$ où la valeur de n est un attribut de M qui n'est pas modifié par l'appel $M(i, j)$.

Dans les 2 programmes, il n'est **pas (peu) utile** d'éliminer les multiplications pour l'accès aux coefficients des 3 matrices.

Dans de nombreux cas, le simple fait d'utiliser un indexage nécessite d'ajouter une couche supplémentaire au problème. Lorsque vous utilisez un tableau, vous utilisez un compteur que vous incrémentez. Pour calculer la position, le système multiplie ce compteur par la taille de l'élément du tableau, puis ajoute l'adresse du premier élément pour obtenir l'adresse. Toute cette séquence d'étapes est atteinte par une seule instruction (`mov ecx, DWORD PTR _a[eax*4]`) en x86, ce qui ne donne aucun avantage à l'accès par des pointeurs. Cette optimisation est propre au compilateur. Il est devenu plus courant d'avoir un compilateur sophistiqué à portée de main, de sorte que le code à base de pointeurs n'est pas toujours plus rapide.

Afin de transformer $M(i, j)$ pour utiliser des pointeurs, le compilateur doit analyser toute la boucle et déterminer que, par exemple, chaque élément est accédé. Ce processus combine des optimisations appelées *Recherche de sous-expressions communes* et *induction variable strength reduction*.

Lors de l'écriture avec des pointeurs, l'ensemble du processus d'optimisation n'est pas nécessaire car le programmeur se contente généralement de parcourir le tableau. Comme les tableaux doivent généralement être **contigus**, un autre avantage des pointeurs est la création de structures allouées de manière incrémentielle. En C, on a de nombreux choix d'allocation mémoire pour représenter une matrice. Une déclaration statique assure que la localité des données est respectée. Cette propriété n'est généralement pas assurée par l'allocation dynamique d'un tableau 2d mais est possible (impossible en C++). Utiliser un tableau 1d pour représenter une matrice assure la localité des données et les avantages qui y sont associés en termes d'accès au cache/mémoire mais impose un accès par $M[i * n + j]$.

Il est important de noter que les index sont plus robustes, car ils survivent souvent à la réaffectation des tableaux. Prenons un exemple : disons que vous avez un tableau qui croît dynamiquement lorsque vous ajoutez des éléments, un index dans ce tableau et un pointeur sur ce tableau. Vous ajoutez un élément au tableau, épuisant ainsi sa capacité, et il doit maintenant augmenter sa taille. Vous appelez `realloc`, et vous obtenez un nouveau tableau (ou l'ancien tableau s'il y avait suffisamment de

mémoire supplémentaire après la fin donnée). Le pointeur que vous aviez est maintenant invalide , l'index cependant est toujours valide.

L'inconvénient des indices est surtout la commodité. Nous devons avoir accès au tableau que nous indexons en plus de l'index lui-même, alors que le pointeur vous permet d'accéder à l'élément sans avoir accès à son conteneur.

Q.5 Un cache très grand

Supposons que le cache est très grand pour contenir les trois matrices A , B et C , soit $Z > 3n^2$.

Le programme (i, j, k)

Comme une matrice a n^2 éléments et qu'on suppose $Z > 3n^2$ alors il y aura $O(\frac{n^2}{L})$ défauts de cache par matrice (on remplace jamais des lignes de cache selon la politique LRU comme notre cache est assez grand pour toutes les contenir), soit $Q(n, L, Z) = O(3\frac{n^2}{L})$ défauts de cache au total.

Le programme (i, k, j)

On procède pareil que pour l'algorithme (i, k, j) : comme une matrice a n^2 éléments et qu'on suppose $Z > 3n^2$ alors il y aura $O(\frac{n^2}{L})$ défauts de cache par matrice (on remplace jamais des lignes de cache selon la politique LRU comme notre cache est assez grand pour toutes les contenir), soit $Q(n, L, Z) = O(3\frac{n^2}{L})$ défauts de cache au total. On peut penser qu'on fera plus de défauts de cache comme B n'est pas parcourue dans le sens du stockage (row-major), mais non car encore une fois, aucune ligne de cache n'est remplacée par une autre, comme on a supposé la taille du cache plus grande que la taille des trois matrices réunies.

Q.6 Un cache très petit

Le cache est très petit, soit $Z \ll n$. On suppose qu'une ligne de nos matrices ne tient pas dans le cache.

Le programme (i, j, k)

Lors du calcul du premier coefficient, on a $\frac{n}{L} + n$ défauts de cache. La localité spatiale est bonne pour les accès à la matrice A mais mauvaise pour les accès à la matrice B. En effet, chaque lecture d'un coefficient de B charge une ligne de cache entière contenant plusieurs coefficients mais un seul est utilisé. On doit charger dans le cache la **ligne** $A[i, :]$ ce qui cause $\frac{n}{L}$ défauts de cache et la **colonne** $B[:, j]$ ce qui cause n défauts de cache (car $Z \ll n$). On a le même nombre de défauts de cache pour le calcul du second coefficient etc... On a donc $n^2 * (\frac{n}{L} + n)$ défauts de cache sur les matrices A et B (car $\frac{n}{L} + n > n$).

Si une ligne ou une colonne est plus grande que la taille du cache, on ne peut réutiliser les données entre les calculs des éléments de la matrice C. Les accès à la matrice C sont linéaires, ils causent $O(\frac{n^2}{L})$ défauts de cache. Détaillons plus précisément : avec i et j fixés, nous ne sommes pas sûrs de faire un seul défaut de cache pour $C(i, j)$. En effet, on va stocker $C(i, j)$ en cache, puis traiter les coefficients de A et B mais pour A et B, on a $\frac{n}{L} + n$ défauts de cache. Or $Z \ll n$ et $\frac{n}{L} + n > n$ donc selon la politique LRU, on remplace la plus ancienne valeur en cache par la plus récente, donc on remplace $C(i, j)$ pour mettre une ligne de valeurs de A/B. Donc, avec i et j fixés, on aura forcément un moment où on va stocker au moins 2 fois $C(i, j)$, d'où le $O(\frac{n^2}{L})$ défauts de cache pour C.

Ainsi $Q(n, L, Z) = O(n^3)$.

Le programme (i, k, j)

On a $Q(n, L, Z) = O(\frac{n^3}{L} + \frac{n^2}{L})$.

On exploite la localité puisque la matrice B est accessible par ligne dans la boucle la plus interne. Mais une réutilisation temporaire n'est pas possible, car la capacité est insuffisante pour contenir tout B jusqu'à ce que la boucle extérieure i change. Ainsi, il y aura $O(\frac{n^3}{L})$ défauts de cache pour les caches à mappage direct et les caches complètement associatifs.

Q.7 Un produit par blocs

Pour améliorer l'algorithme dans les conditions de la question précédente, on peut effectuer un produit par blocs de taille B par B , avec $B > L$.

Si la taille B du block vérifie $3B^2 < Z$, on peut conserver à tout instant un bloc de la matrice A, un bloc de la matrice B et un bloc de la matrice C dans le cache.

Multiplier deux blocs ne cause pas de défauts de cache en dehors de la lecture des blocs de A et B et de l'écriture du résultat dans C. Pour chaque bloc dans la matrice C, on doit lire $\frac{n}{B}$ blocs dans la matrice A et dans la matrice B. La lecture d'un bloc cause $\frac{B^2}{L}$ défauts de cache. On a donc au total :

$$\left(\frac{n}{B}\right)^2 \left(\frac{n}{B} * 2 * \frac{B^2}{L} + \frac{B^2}{L}\right) = O\left(\frac{n^3}{B.L}\right)$$

On veut B le plus grand possible sachant que $3B^2 < Z$. On prend donc $B = \sqrt{\frac{Z}{3}}$, ce qui donne $Q(n) = O\left(\frac{n^3}{L.\sqrt{Z}}\right)$ défauts de cache.

L'algorithme précédent a besoin de connaître la taille du cache Z . On peut atteindre la même borne sans utiliser cette valeur et obtenir ainsi un algorithme cache-oblivious. Pour cela on utilise l'algorithme de multiplication de matrices diviser pour régner. On décompose récursivement le produit en 8 produits de matrices $n/2$ par $n/2$. La complexité en nombre d'instructions est

$$\begin{aligned} W(1) &= O(1) \\ W(n) &= 8W\left(\frac{n}{2}\right) + O(n^2) \end{aligned}$$

ce qui donne $W(n) = O(n^3)$. On calcule de la même manière le nombre de défauts de cache. L'addition des sous-produits accède aux données linéairement et cause $O(n^2/L)$ défauts de cache. De plus, lorsque $3n^2 < Z$, les 3 sous-matrices tiennent dans le cache donc les appels récursifs ne génèrent plus de défauts de cache. On a donc :

$$\begin{aligned} Q(n) &= \frac{n^2}{L} && \text{si } 3n^2 < Z \\ Q(n) &= 8Q\left(\frac{n}{2}\right) + O\left(\frac{n^2}{L}\right) && \text{sinon.} \end{aligned}$$

ce qui donne

$$Q(n) = O\left(\frac{n^3}{L.\sqrt{Z}}\right)$$

Cet algorithme atteint asymptotiquement la même performance que l'algorithme précédent sans qu'il ne soit nécessaire de connaître la taille du cache.