Université Paris-Saclay[1]
Institut Polytechnique de Paris[2]

Master SETI

A0 Architecture des processeurs

Agustin Coitinho[2]
Alaf Do Nascimento Santos[2]
Irene Asensio Benedicto[2]
Simon Berthoumieux[1]

2023/2024

Contents

1 Analyse de performances

1.1 Objectifs

Parmi les objectifs de ce travail pratique (TP), il est important de souligner les suivants :

- la familiarisation avec les méthodes de mesure de performances de programmes;
- l'étude de l'impact des options de compilation sur les performances;
- la mise en évidence de l'influence des caches sur les temps d'exécution.

D'abord, du code C permettant d'avoir accès à un compteur de cycles du processeurs (TimeStamp Counter), en plus des fonctions de calculs d'intérêt nous ont été fournies par le professeur. Les codes base données lors du début du TP sont disponibles dans les Annexes A et B.

Sachant que nous allons travailler avec l'analyse de performances, les résultats obtenus seront complètement liée à la machine où les expériences ont été réalisées. Donc, il est essentiel de présenter les configurations du ordinateur qui a été utilisé.

Dans un premier moment, nous exécutons la commande Linux `lscpu`, qui nous donne les informations suivantes :

```
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         39 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  12
  On-line CPU(s) list:   0-11
Vendor ID:               GenuineIntel
  Model name:            11th Gen Intel(R) Core(TM) i5-11400H @ 2.70GHz
    CPU family:          6
    Model:               141
    Thread(s) per core:  2
    Core(s) per socket:  6
    Socket(s):           1
    Stepping:            1
    CPU max MHz:         4500,0000
    CPU min MHz:         800,0000
    BogoMIPS:            5376.00
    Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mc
                         a cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss
                         ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art
                          arch_perfmon pebs bts rep_good nopl xtopology nonstop_
                         tsc cpuid aperfmperf tsc_known_freq pni pclmulqdq dtes6
                         4 monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx16 xtpr p
                         dcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline
```

```
                        _timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefe
                        tch cpuid_fault epb cat_l2 invpcid_single cdp_l2 ssbd i
                        brs ibpb stibp ibrs_enhanced tpr_shadow vnmi flexpriori
                        ty ept vpid ept_ad fsgsbase tsc_adjust bmi1 avx2 smep b
                        mi2 erms invpcid rdt_a avx512f avx512dq rdseed adx smap
                         avx512ifma clflushopt clwb intel_pt avx512cd sha_ni av
                        x512bw avx512vl xsaveopt xsavec xgetbv1 xsaves split_lo
                        ck_detect dtherm ida arat pln pts hwp hwp_notify hwp_ac
                        t_window hwp_epp hwp_pkg_req avx512vbmi umip pku ospke
                        avx512_vbmi2 gfni vaes vpclmulqdq avx512_vnni avx512_bi
                        talg avx512_vpopcntdq rdpid movdiri movdir64b fsrm avx5
                        12_vp2intersect md_clear flush_l1d arch_capabilities
Virtualization features:
  Virtualization:       VT-x
Caches (sum of all):
  L1d:                  288 KiB (6 instances)
  L1i:                  192 KiB (6 instances)
  L2:                   7,5 MiB (6 instances)
  L3:                   12 MiB (1 instance)
NUMA:
  NUMA node(s):         1
  NUMA node0 CPU(s):    0-11
Vulnerabilities:
  Gather data sampling: Mitigation; Microcode
  Itlb multihit:        Not affected
  L1tf:                 Not affected
  Mds:                  Not affected
  Meltdown:             Not affected
  Mmio stale data:      Not affected
  Retbleed:             Not affected
  Spec rstack overflow: Not affected
  Spec store bypass:    Mitigation; Speculative Store Bypass disabled via prctl
                         and seccomp
  Spectre v1:           Mitigation; usercopy/swapgs barriers and __user pointer
                         sanitization
  Spectre v2:           Mitigation; Enhanced IBRS, IBPB conditional, RSB fillin
                        g, PBRSB-eIBRS SW sequence
  Srbds:                Not affected
```

Ensuite, la commande `cat /proc/cpuinfo` nous présente les fréquences de chaque processeurs, prenons la sortie relative au processeur 1 par exemple :

```
processor : 1
vendor_id : GenuineIntel
cpu family : 6
```

```
model : 141
model name : 11th Gen Intel(R) Core(TM) i5-11400H @ 2.70GHz
stepping : 1
microcode : 0x4e
cpu MHz : 1116.830
cache size : 12288 KB
physical id : 0
siblings : 12
core id : 1
cpu cores : 6
apicid : 2
initial apicid : 2
fpu : yes
fpu_exception : yes
cpuid level : 27
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov [...]
vmx flags : vnmi preemption_timer posted_intr invvpid ept_x_only ept_ad [...]
bugs : spectre_v1 spectre_v2 spec_store_bypass swapgs eibrs_pbrsb gds
bogomips : 5376.00
clflush size : 64
cache_alignment : 64
address sizes : 39 bits physical, 48 bits virtual
power management:
```

Étant donné que les fréquences ne sont pas toujours égale pour tous les processeurs, dans le cadre de ce travail, nous allons toujours exécuter notre programme dans le processeur 1. Pour le faire, il suffit juste de se servir de la commande `taskset -c 1 ./tp1`, avec tp1 étant le nom de notre exécutable.

1.2 Optimisations du compilateur

En utilisant le compilateur gcc, nous avons étudier les différents options pour l'optimisation. Pour faire simples, la liste suivante représente un peu le fonctionnement de chaque optimisation utilisé:

- **O0** : cela n'effectue aucune optimisation;
- **O1** : cela effectue quelques optimisations;
- **O2** : cela effectue un niveau d'optimisation important mais pas très agressive;
- **O3** : cela effectue des optimisations agressives avec notamment vectorisation SIMD.

Pour continuer, nous faisons un étude de l'évolution des performances avec quelques fonctions du fichier *tp1.c*, comme montre de notre fonction `main` à ce stade :

```
int main()
{
  printf("Evaluation : N=%d, type="STR(TYPE)"\n",N);

  copy_ij();
  copy_ji();
  mm_ijk();
  mm_ikj();
  mm_b_ijk();

  return 0;
}
```

Sachant que dans la section 1.3 nous allons mieux étudier la performance de chaque fonction individuellement, au cours de la présente section nous allons simplifier l'analyse par la comparaison de temps de exécution, à travers la commande Linux `time`, et tailles de sections et fichiers binaires, en utilisant la commande `size`.

En utilisant la commande `gcc -O0 -c tp1.c` et après `size tp1.o`, nos pouvons obtenir la Table 1. Pour le temps d'exécution en utilisant `time taskset -c 1 ./tp1`, le résultat obtenu est donné par la Table 2.

text	data	bss	dec
4981	0	24000216	24005197

Table 1: Taille des sections du fichier objet avec optimisation O0.

real	user	sys
1m45,990s	1m45,979s	0m0,006s

Table 2: Temps d'exécution avec optimisation O0.

Ensuite, pour l'optimisation O1, la commande `gcc -O1 -c tp1.c` et après `size tp1.o`, nos pouvons obtenir la Table 3 et, pour le temps d'exécution donné par `time taskset -c 1 ./tp1`, le résultat obtenu est dans la Table 4.

text	data	bss	dec
3529	0	24000200	24003729

Table 3: Taille des sections du fichier objet avec optimisation O1.

real	user	sys
0m29,126s	0m29,118s	0m0,008s

Table 4: Temps d'exécution avec optimisation O1.

Concernant l'optimisation O2, les résultats des tailles de section sont dans la Table 5 et les informations temporelles sont présentés par la Table 6.

text	**data**	**bss**	**dec**
4995	0	24000200	24005195

Table 5: Taille des sections du fichier objet avec optimisation O2.

real	**user**	**sys**
0m25,575s	0m25,562s	0m0,011s

Table 6: Temps d'exécution avec optimisation O2.

Finalement, pour l'optimisation O3, la Table 7 montre les différentes tailles de sections et la Table 8 nous donne le temps d'exécution du binaire généré avec cette flag d'optimisation.

text	**data**	**bss**	**dec**
6790	0	24000200	24006990

Table 7: Taille des sections du fichier objet avec optimisation O3.

real	**user**	**sys**
0m20,432s	0m20,426s	0m0,005s

Table 8: Temps d'exécution avec optimisation O3.

Pour chaque optimisation, on peut vérifier que principalement la taille de la section **text** qui change, puisque les données restent les mêmes toujours, l'optimisation agira surtout au niveaux de nombre d'instructions exécutées (text section). Le plus nous augmentons l'optimisation, le plus difficile deviens l'interprétation des résultats à cause de l'agressivité des modifications réalisées par le compilateur.

Par rapport aux temps d'exécution, il est notable que le temps réel, c'est-à-dire, le temps compris entre le moment où nous tapons enter sur le clavier et la fin du programme, était inversement proportionelle aux optimisations utilisées. En autres mots, le temps réel d'exécution du programme, avec les différentes optimisations, donné par $T(Ox)$, obéit la relation :

$$T(O0) > T(O1) > T(O2) > T(O3)$$

Cependant, il est important de savoir que ça était le cas lors de l'exécution de notre expérience. En effet, l'environnement Linux a d'autres processus en parallèle et le processeur n'est pas dédié qu'à notre logiciel, cela est une des raisons pour laquelle nous observons même des temps réels plus important que la somme entre le temps d'exécution en mode utilisateur et temps système.

1.3 Étude de diverses fonctions

Maintenant on s'intéresse au étude des fonctions individuellement. Les programmes seront analyser pour différentes tailles des objets, à travers la variation du paramètre N. En plus, on pourra changer le type de données en définissant la macro TYPE. L'optimisation, pour simplifier, à être considérer est l'option O2 et les résultats vont être analyser en nombre de cycles par itération. La commande shell suivante sera utile pour faire des boucles pour obtenir un ensemble de mesure en faisant varier N.

```
for n in 100 500 1000; do gcc -O2 -DN=$n tp1.c -o tp1; taskset -c 1 ./tp1;done
```

Dans les sous-sections suivantes on va travailler avec les fonctions séparément. La méthode d'évaluation pour faire des statistiques qui a été choisie est la médiane, parce que la moyenne aurait pris en considération les mesures aberrantes (trop élevées) causées par des misses obligatoires (cold miss) et le minimum serait plutôt très optimiste et pas vraiment une représentation rapide et fiable comme la médiane dans le cadre de notre expérience.

1.3.1 Mise à zéro d'un vecteur

Initialement, pour la fonction `zero()`, qui met un vecteur de taille $N * N$ à zero, on regarde l'évolution du temps d'exécution par rapport à taille de vecteurs. Cette analyse est illustré par la Figure **??**.

1.3.2 Copie de matrices

1.3.3 Addition de matrices

1.3.4 Produit scalaire de deux vecteurs

1.3.5 Produit de matrices

1.4 Questions additionnelles

2 Simulateur de processeur MIPS

2.1 Introduction :

Nous utiliserons, pour ce TP, la version 1.2.9 de Edumips, les versions plus récentes étant incompatibles avec la version de java disponible au bâtiment 625.
Nous nous intéresserons, en début de TP, à l'influence du forwarding. Cependant, nous supposerons par la suite que nous n'avons aucun mécanisme d'envoi pour le reste du TP.

2.2 Étude de pipeline sur des programmes élémentaires

On commence par exécuter les programmes `sum.s`, `mul.s`, `sumf.s`, `mulf.s`, `divf.s` et `abs.s` avec et sans forwarding. On obtiendra les résultats suivants :

Sans forwarding :

Programme	**Nb d'instructions**	**Nb de cycles**	**Nd de cycles d'attente**	**CPI**
sum.s	5	13	4	2.600
sumf.s	5	16	7	3.200
mul.s	6	16	6	2.666
mulf.s	5	19	10	3.800
divf.s	5	36	27	7.200
abs.s	8	19	7	2.375

Avec forwarding :

Programme	**Nb d'instructions**	**Nb de cycles**	**Nd de cycles d'attente**	**CPI**
sum.s	5	10	1	2.000
sumf.s	5	13	4	2.600
mul.s	6	11	1	1.833
mulf.s	5	16	7	3.200
divf.s	5	33	24	6.600
abs.s	8	16	4	2.000

Le nombre d'instructions à réaliser ne changera, sans surprise, pas entre les deux modes, car nous continuerons d'exécuter le même programme. Cependant, nous pouvons constater que pour tout ces programmes le nombre de cycles d'attente diminuera.
Le nombre de cycles diminuant, mais le nombre d'instruction restant constant, notre CPI diminuera aussi.
La diminution du nombre de cycles d'attente s'explique assez facilement par le fait que tout ceux-ci sont dû à des dépendances de données read-after-write (RAW). Or, le forwarding réduisant la latence, en rendant les résultats disponibles plus tôt dans le pipeline, les temps d'attentes imposés par les dépendances de données se trouveront réduits par le

forwarding (ce qui ne serait pas forcément le cas pour des dépendances matérielles, par exemple).

Le pipeline du processeur Mips possède 5 étages :

- IF : Instruction Fetch
- ID : Instruction Decode
- EX : Execution
- MEM : Memory Management
- WB : Write-Back

Dans le cas sans forwarding, la donnée sera toujours disponible après l'étage WB, alors que dans le cas où nous avons du forwarding, celle-ci sera disponible après l'étage EX pour les opérations arithmétiques, et après l'étage MEM pour les opérations de load. La latence sera donc toujours deux cycles plus longues sans forwarding pour les opérations arithmétiques, et un cycle plus longue sans forwarding pour le chargement des données (il est à noter que les opérations de chargement d'un registre dans un autre, tel que `mflo` ou `mov.d` auront des résultats disponibles à la fin de l'étage EX).
On commence à analyser les différents programmes. Pour `sum.s`, avec forwarding, on aura :

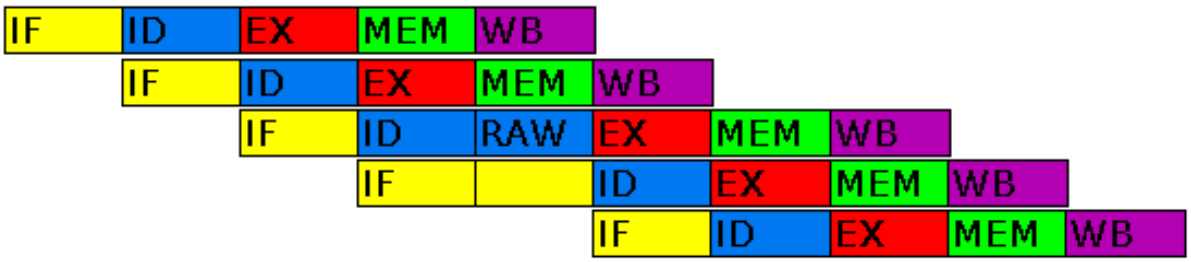

Figure 1: Exécution obtenue pour `sum.s` avec forwarding

La latence correspondra au nombre de cycle qu'il faudra attendre avant de pouvoir lancer une autre instruction qui utiliserait la donnée produite par la première dans son étage EX, et qui s'exécuterait sans cycle d'attente. Une latence de 0 signifierait que (en négligeant les dépendance matérielles) on pourrait lancer notre instruction productrice et notre instruction consommatrice sur le même cycle, et ne pas avoir de cycle d'attente. Cela ne sera généralement pas le cas, et toutes les instructions auront généralement au minimum un cycle de latence.
Les `ld` sur les deux premières lignes ont une dépendance RAW avec la troisième, pareil pour le `dadd` de la troisième ligne et le `sd` de la ligne 4. On peut donc calculer la latence que les instructions utilisées auront avec(*sans*) forwarding : On peut voir que `ld` aura

2(*3*) cycles de latence (2 avec forwarding, 3 sans), et que pour `dadd` aura 1(*3*) cycles de latence.
Pareillement pour sumf :

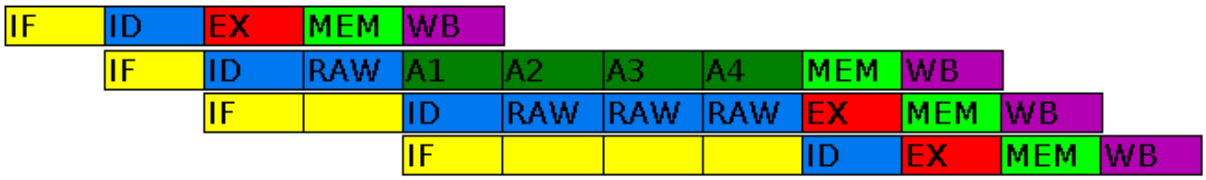

Figure 2: Exécution obtenue pour `sumf.s` avec forwarding

On peut voir que `add.d` a 4(*6*) cycles de latence.
Pareillement pour les autres programmes, on pourra voir que `dmult` et `mflo` ont 1(*3*) cycles de latence chacun (il y a besoins de deux instructions séparées car la multiplication devra envoyer le résultat sur deux registres correspondants aux parties low et high. On récupère ici seulement la partie low.), `mul.d` a 7(*9*) cycles de latence, `div.d` a 24(*26*) cycles de latence, `mov.d` et `c.lt.d` ont 1(*3*) cycles de latence, et `sub.d` a 4(*6*).

On peut constater que les opérations flottantes (avec un .d) tendront à avoir une latence plus grande que les opérations entières. La division entière est la plus longue opérations de toutes celles-ci, car elle nécessite le calcul de l'inverse d'un flottant, ce qui sera généralement fait par résolution de $fx - 1 = 0$ par la méthode de newton, o

sur abs.s c'est possible d'optimiser en inversant les lignes

```
l.d f2, A1(r0)          ; f2=A1
l.d f0,zero(r0)
```

code original

```
;; B1=abs(A1)
.data
A1: .double -10.0
B1: .double 0.0
zero: .double 0.0
.text
main:

l.d f0,zero(r0)
l.d f2, A1(r0)          ; f2=A1
mov.d f4, f2 ; f4=B1 = A1
c.lt.d 7, f2, f0 ; B1 < 0 ?
bc1f 7, done ; if false, nothing to do
sub.d f4, f0, f2 ; f4=-A=|A|= 0.0 - f4
done:
```

```
s.d f4,B1(r0)
halt
```

après changement

```
;; B1=abs(A1)
.data
A1: .double -10.0
B1: .double 0.0
zero: .double 0.0
.text
main:
l.d f2, A1(r0)          ; f2=A1
l.d f0,zero(r0)
mov.d f4, f2 ; f4=B1 = A1
c.lt.d 7, f2, f0 ; B1 < 0 ?
bc1f 7, done ; if false, nothing to do
sub.d f4, f0, f2 ; f4=-A=|A|= 0.0 - f4
done:
s.d f4,B1(r0)
halt
```

3 Simulation de cache de données et TLB

3.1 Objectifs du TP

3.2 Produit-scalaire

3.3 Produit Matrice Vecteur

3.3.1 Observaciones Iniciales

Observar que para realizar el producto matriz vector, cada fila de la matriz sólo se emplea una vez (es decir, solo se lee una vez). Sin embargo, el vector se reutiliza una vez por cada una de las filas de la matriz. Esto implica que las faltas por conflicto o capacidad solo pueden ser causadas por accesos fallidos en cache a este vector.

3.3.2 Impacto número WAY

- Respecto a échec obligatorios: No cambian, pues son los accesos obligados que hay que hacer a memoria la primera vez que se accede a una dirección (a un dato). Como estos accesos a memoria son del mismo tamaño, no influye el aportar más conjuntos a la configuración de la cache en cuanto a accesos obligatorios a la memoria.

- Respecto a échec por conflicto: Aumentar el número de ways implica que cada vez que se trae un dato a una línea de cache como consecuencia de un miss, ya no sea estrictamente necesario quitar el dato que había previamente. Esto se debe a que la política LRU asegura que el dato más recientemente utilizado tiene prioridad para permanecer en caché mientras que el que no se ha usado desde hace más tiempo será expulsado en favor del nuevo dato. De este modo, dado que el vector se utiliza en cada iteración por las filas de la matriz, el dato que se expulsa cuando una línea necesita ser usada por un nuevo dato es el correspondiente a la matriz. Así pues, se consigue ya con 2 ways mantener siempre el vector en caché, eliminando los posibles fallos por conflicto.

- En cuanto a échec por capacidad: Puesto que los échecs por conflicto se reducen al aumentar el número de vías, también lo hacen los échecs de capacidad (pues el la naturaleza del conflicto es la misma con la única diferencia de que se clasifican en esta categoría si la caché esta llena).

3.3.3 Impacto tamaño LINE

- Échecs obligatorios: Aumentar el tamaño de las lines permite hacer transferencias de mayor tamaño desde la memoria principal a la caché. Así pues, al aumentar el tamaño de línea, para un acceso se traen a caché el dato de interés y los sucesivos bloques de memoria que el programa encontrará presentes en caché cuando necesite acceder a dichas direcciones (y por lo tanto no necesitará ir a buscar estos datos a la memoria principal). Esto aporta una reducción del número de fallos obligatorios directamente proporcional al factor de aumento del tamaño de line.

- Échecs por conflicto: Un tamaño de caché constante en el que se ha aumentado el tamaño de línea, implica que esta nueva configuración dispone de menor número de líneas y por lo tanto, mayor número de direcciones de memoria mapeadas a una misma dirección de caché. Esto provoca mayor tasa de fallos por conflicto que para una configuración de caché con menor tamaño de línea.

- Échecs por capacidad: igual al caso anterior ya que el problema subyacente de este tipo de echec es el mismo que por conflicto con la salvedad de que la memoria se encuentra llena.

3.3.4 Impacto tamaño N

- Échec obligatorio: Dado que los échecs obligatorios vienen marcados por la transferencia de cada una de las filas de la matriz a la caché, variar N no impacta esta tasa, pues si bien la cantidad absoluta de "misses" es mayor, el número de accesos también lo es, lo que mantiene esta tasa constante en todos los casos.

- Échec por conflicto y capacidad: Aumentar N implica que el programa está manejando un tamaño mayor de matriz y de vector. Consecuentemente, se requiere mucho más espacio de memoria para poder almacenar los datos empleados para cada iteración de la multiplicación. Esto se observa en la tasa de fallos por capacidad, que aumenta considerablemente a medida que aumenta el tamaño de la matriz y del vector, pues para un mayor tamaño de N antes se encontrará la caché llena y antes se clasificarán los conflictos causados por el mapeo de direcciones de memoria principal a la caché como fallos por capacidad. Por esta razón se disminuyen los fallos por conflicto, pues antes son catalogados como fallos por capacidad.

3.3.5 Impacto size cache

- Échec obligatorio: La cantidad de veces que traemos datos por primera vez no depende del tamaño de la cache. Por lo tanto, esta tasa se mantiene constante.

- Échec por conflicto: La tasa se mantiene constante porque no depende de la cantidad de espacio disponible en caché, sino del mapeo de direcciones de memoria a líneas de caché. Por ejemplo: si en una caché de 4kB tenemos X échecs por conflictos, al duplicar el tamaño de la caché la cantidad de direcciones de memoria mapeadas a una línea de caché se reduce a la mitad. En consecuencia, la cantidad de conflictos en los primeros 4k de memoria va a ser X/2 y en los segundos 4k será también X/2. Así, la cantidad de échec por conflicto se mantiene constante, y como el número de accesos necesarios a memoria también lo es (por la necesidad de traer las filas de la matriz), la tasa permanece constante.

- Échec por capacidad: La tasa se ve mejorada ya que al poder almacenar más datos simultáneamente, la caché se llenará más tarde (si es que llega a hacerlo).

Para un tamaño de memoria muy grande en comparación con el tamaño de datos que maneja el programa, los fallos por conflicto y capacidad disminuyen mucho ya que

a efectos prácticos, podemos considerar que tenemos una caché con mapeo directo en el que hay espacio más que suficiente para almacenar todos los datos sin necesidad de sobreescribir ninguno.

3.4 Produit de matrices

3.4.1 Observaciones Iniciales

Dado un producto AxB, con A,B $\in M(\mathbb{R})$, y con el algoritmo de multiplicación de orden ijk, tenemos que A será recorrida por filas cargando cada fila una única vez en cache. Al mismo tiempo, cada columna de B será recorrida N veces. Dado que dichos datos deberán ser reutilizados, su presencia en caché puede dar lugar a eventuales hits/miss. Observar que el orden en el que son cargadas las matrices en memoria es por filas consecutivas.

3.4.2 Análisis

Vemos que al aumentar el número de ways tenemos una anomalía, pues obtenemos un mayor número de echecs. Primero, teniendo en cuenta que las matrices estan guardads en memorias por filas consecutivas, cuando hacemos A.B, los vectores pertenecientes a A los traemos por fila y una unica vez cada uno, lo que genera echec obligatorios. Segundo, los vectores pertenecientes a B los traemos por fila. Sin embargo, la forma de recorrer esta matriz para hacer el producto deberia ser traerlos por columna para explotar la localidad espacial. Por lo tanto, en cada transferencia estamos trayendo muchos datos que no utilizamos y a su vez, para acceder a los datos de una columna necesitamos probablemente muchos mas accesos a memoria. Es decir, la manera en la que traemos los datos y en la que los usamos es diferente, por lo tanto la cache no logra explotar la distribucion de los datos. Obviamente, este efecto se ve mermado cuando el numero de ways es muy grande y cuando el tamano de la cache y el de los datos son de tamanos equiparables. Si lo segundo no se cumple, aumentar el número de ways no garantiza una mejora.

Si el tamano de los datos es mucho mayor al de la cache, la cache no consigue retener los datos necesarios para la ejecucion del programa por lo que necesita constantemente acceder a memoria para hacer sus operaciones. Por lo anterior, ninguna de nuestras configuraciones de cache se comporta bien para N=128 ni 256. Ademas, es esperable un aumento en el tiempo de ejecucion por la gran cantidad de transferencias que deben hacerse.

3.4.3 Caso para N no potencia de 2

3.4.4 Comparativa ikj vs ijk

En esta sección, repetimos los mismos pasos que en el caso precedente pero para el orden de multiplicación ikj. Observamos que para mismos escenarios, se obtienen unas tasas de échec mejores. Esto se debe a que en este caso, estamos recorriendo la matriz B por filas en lugar de por columnas. De este modo, se consigue aprovechar el modo en el que los datos se traen a caché (datos almacenados en orden dentro de una fila).

3.5 Produit de matrices par blocs

Para empezar, para que tenga efecto la aplicación de este algoritmo es necesario que B sea menor a N. En caso contrario, estaríamos volviendo a ejecutar el caso anterior.

Ejecutando el programa para las configuraciones de caché propuestas y los distintos valores de B y N observamos distintas tasas de échec. Analizando las tasas de échec dentro de cada configuración podemos determinar el valor óptimo de B fijándonos en qué valor de este parámetro proporciona un valor mínimo de tasa de fallos. La fijación de B implica un tradeoff entre el número de columnas de A y el número de filas y columnas de B que consideramos en cada ventana. Tras una lectura del funcionamiento del algoritmo, vemos que la elección del parámetro B fija unas submatrices A_B y B_B tal que $A_B \in M_{NxB}$ y $B_B \in M_{BxB}$. Tomar la decisión de fijar B muy pequeño hace que A_B se parezca a un vector columna (implicando un acceso a los datos que no sigue su orden en memoria). Un parámetro B grande hace que nos alejemos de la aproximación de multiplicación por bloques (acercándonos a la aproximación original con las desventajas que ello implica y que ya han sido comentadas anteriormente).

3.6 Simulation de TLB

Expliquez pourquoi un TLB de e entrées, avec une associativité a et gérant des pages de taille p, aura le même taux d'échec qu'un cache ayant la même associativité a, e/a ensembles de a lignes et des lignes de taille p.

- Análisis del TLB: Para un TLB de e entradas y asociatividad a tenemos un número total de entradas "efectivas" de $e \cdot a$. Sabiendo que cada una de esas entradas permite gestionar una página de memoria de tamaño p, tenemos que un TLB gestiona simultáneamente $\boldsymbol{e \cdot a \cdot p}$ posiciones de memoria.

- Análisis de la caché: Aplicando mismo razonamiento, una caché que tiene $\frac{e}{a}$ conjuntos de a líneas presenta globalmente un total de $\frac{e}{a} \cdot a$ líneas. Si tenemos una asociatividad de a ways por cada línea, el número total efectivo de líneas será $\frac{e}{a} \cdot a \cdot a = e \cdot a$. Si además sabemos que cada línea tiene un tamaño p, entonces el número simultáneo de datos que pueden coexistir en la caché es de $\boldsymbol{e \cdot a \cdot p}$.

Así pues, dado que ambos componentes pueden manipular bloques de un mismo tamaño, es de esperar que la tasa de accesos fallidos sea igual para un determinado número de accesos.

A tsc.h

```
// Pour mesurer les performances d'un pentium

// Suivant la machine, le temps de mesure peut varier.
// Utiliser la fonction eval_tsc_cycles() pour avoir une mesure un peu plus fine
#include <x86intrin.h>

// si correction systématique nécessaire. 100.0 donne en général de bons résultats
#define TSCCYCLES 0.0

//#define USE_RDTSC_INTRINSIC

// suivant l'architecture, le nombre et types de registres touchés par rdtsc varie
#ifdef __i386__
#  define RDTSC_DIRTY "%eax", "%ebx", "%ecx", "%edx"
#elif __x86_64__
#  define RDTSC_DIRTY "%rax", "%rbx", "%rcx", "%rdx"
#else
# error unknown platform
#endif


static inline unsigned long long start_timer()
{
#if ! defined(USE_RDTSC_INTRINSIC)   // utiliser du code assembleur x86
  unsigned int hi = 0, lo = 0;

  asm volatile("cpuid\n\t" /* pour vider le pipeline */
       "rdtscp\n\t" /* on lit le registre TSC */
       "mov %%edx, %0\n\t" /* on restore les registres */
       "mov %%eax, %1\n\t"
       : "=r" (hi), "=r" (lo) /* Le résultat */
       :: RDTSC_DIRTY);       /* les registres à sauvegarder dans la pile */
  unsigned long long that =   (unsigned long long)((lo) |
   (unsigned long long)(hi)<<32);

  return that;
#else
  return _rdtsc();
#endif
}

static inline unsigned long long stop_timer()
{

#if ! defined(USE_RDTSC_INTRINSIC)   // utiliser du code assembleur x86
```

```
  unsigned int hi = 0, lo = 0;

  asm volatile("rdtscp\n\t"    /* on lit le registre TSC */
       "mov %%edx, %0\n\t" /* on restore les registres */
       "mov %%eax, %1\n\t"
       "cpuid\n\t"       /* barrière hw */
       : "=r" (hi), "=r" (lo) /* Les PF et pf du résultat */
       :: RDTSC_DIRTY);       /* les registres à sauvegarder dans la pile */
  unsigned long long that =   (unsigned long long)((lo) |
   (unsigned long long)(hi)<<32);

  return that;
#else
  return _rdtsc();
#endif
}

// convertit la différence en double pour faciliter les calculs
// et ajuste en enlevant avec le temps moyen d'excécution de l'instruction rdtsc
static inline double dtime(long long debut, long long fin)
{
  double time;
  return (double) (fin - debut - TSCCYCLES) ;
}

#ifdef EVALTSCTIME

// Programme pour évaluer le temps d'une mesure par rdtsc sur un ordinateur.
// Copier dans un ficher avec l'extension.c
// et compiler avec gcc -O3 -DEVALTSCTIME tsc.c -lm

#define N 100000
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int cmp(const void *x, const void *y)
{
  double xx = *(double*)x, yy = *(double*)y;
  if (xx < yy) return -1;
  if (xx > yy) return  1;
  return 0;
}



double tmean(double ar[], int n){
```

```
  // moyenne arrangée pour enlever les points abherrants.
  // On cherche la médiane m et les quartiles q1 et q3.
  // On calcule la moyenne sur tout ce qui est dans [m-k*(q3-q1),m+k*(q3-q1)]
  // k=1.0 marche bien

  double q1,q2,q3,sum=0.0,k=1.0;
  int ii,jj;
  // On trie les données pour avoir mediane et quartiles
  qsort(ar, n, sizeof(double), cmp);
  q1=ar[N/4];
  q2=ar[N/2];
  q3=ar[3*N/4];

  // calcul de la moyenne des valeurs dans l'intervalle [q2-k*(q3-q1),q2+k*(q3-q1)]
  ii=0;
  while(ar[ii++]<q2-k*(q3-q1)) ;
  jj=ii;
  while(ar[jj]<q2+k*(q3-q1)) {
    sum += ar[jj];
    jj++;
    if(jj >= n-1) break;
  }
  return sum/(double)(jj-ii);
}

void eval_tsc_cycles(){

  double tsum=0.0, t2sum=0.0, tavg, tvar, tmin=1e30, tmax=0.0, ttavg;
  double tresults[N];
  long long debut, fin;
  double t;

  for(int i=0;i<N;i++){

    debut=start_timer();
    fin=stop_timer();

    t = (double) (fin - debut) ;
    // mise dans un tableau pour moyenne arrangée
    tresults[i]=t;
    // pour calcul moyenne et variance brutes
    tsum += t;
    t2sum += t*t;
    tmin=(tmin<t ? tmin : t);
    tmax=(tmax>t ? tmax : t);
  }
  tavg = tsum/N; // moyenne brute
```

```
  tvar = sqrt((t2sum + (tavg*tavg*N) - 2 * tavg * tsum)/N); // variance brute
  ttavg=tmean(tresults,N); // moyenne arrangée
  printf("moyenne=%f\tmin=%f\tmax=%f\tvariance=%f\tavg=%f\n",tavg,tmin,tmax,tvar,ttavg);
}



int main(){
  eval_tsc_cycles();
}

#endif
```

B tp1.c

```
#include <stdio.h>
#include <stdint.h>

#ifndef N
#define N 1000 // la taille des matrices [N][N]/vecteurs[N^2]
#endif
#ifndef M
#define M 16 // nombre d'itérations de chaque fonction
#endif
#ifndef TYPE
#define TYPE float // ou double ou int, etc. On peut essayer les
// différents types de données
#endif

// pour éviter des conversions int-float inutiles
#if TYPE==int || TYPE==short || TYPE==char || TYPE==uint64_t
#  define ZERO 0
#elif  TYPE==float
#  define ZERO 0.0f
#else
#  define ZERO 0.0
#endif


#ifndef BL
#define BL 16 // La taille des blocs pour la mult de matrices par blocs
#endif

#define STR1(x) #x
#define STR(x) STR1(x)

// Permet de lire le compteur interne du nombre de cycles du
// processeur (TSC timestamp counter).
// Le TSC renvoie le nombre de cycles codé sur 64 bits (long long)
// start_timer() initie la mesure et stop_timer() la termine
// dtime() convertit en double la différence pour simplifier les calculs ultérieurs
//      et retranche le temps (approximatif) bécessaire à la mesure du TSC

static unsigned long long start_timer() ;
static unsigned long long stop_timer() ;
static double dtime(long long debut, long long fin);


// le fichier où sont les fonctions d'accès au TSC
#include "tsc.h"
```

```
TYPE AF[N][N], YF[N][N], XF[N][N], YT[N][N],// matrices [N][N]
  BF[N*N], CF[N*N],// vecteurs N^2
  SF; // accumulateur

double resultats[M];                              // les temps de calcul en double

static inline int min (int a,int b)
{
  if (a<b) return a;
  else return b;
}

long long debut, fin;
double benchtime;

// pour les calculs, on met les temps obtenu pour chaque essai dans un tableau.
// Le contenu du tableau est ensuite affiché et/ou traité.
void add_res(double res, int where)
{
  resultats[where]=res;
}

// la fonction d'affichage des résutats. Peut être redéfinie en fonction des besoins
// pour faire des statistiques, enlever les point aberrants, etc
// Le deuxième paramètre est le nombre d'itérations pour normaliser
void print_res(char *funcname, double nb_iter){
  double normalize = 1.0/nb_iter;
  printf("%s\t", funcname);
  for(int i=0; i<M; i++)
    printf("%g%s",resultats[i]*normalize, (i==M-1 ? "\n" : "\t"));
}

// entre deux mesures d'un même programme. Rédéfinir si nécessaire
void separateur(){
  printf("\n");
}

//Mise à zéro d'un vecteur
void zero(){

  int i, j, m ;

  for (m=0;m<M;m++)
    {
      debut=start_timer();
```

```
      for (i=0;i<(N*N);i++)
BF[i]=ZERO;
      benchtime=dtime(debut, stop_timer());
      add_res(benchtime,m);
    }
  print_res ("ZERO",(N*N));
  separateur();
}

//copie de matrices
void copy_ij(){
  int i, j, m ;

  for (m=0;m<M;m++)
    {
      debut=start_timer();

      for (i=0;i<N;i++)
for (j=0;j<N;j++)
  AF[i][j]=YF[i][j];
      benchtime=dtime(debut, stop_timer());
      add_res(benchtime,m);
    }
  print_res ("COPY_ij", (N*N));
  separateur();
}

void copy_ji(){
  int i, j, m ;

  for (m=0;m<M;m++)
    {
      debut=start_timer();

      for (j=0;j<N;j++)
for (i=0;i<N;i++)
  AF[i][j]=YF[i][j];
      benchtime=dtime(debut, stop_timer());
      add_res(benchtime,m);
    }
  print_res ("COPY_ji", (double)(N*N));
  separateur();
}

// addition de deux matrices
void add_ij(){
```

```
  int i, j, m ;

  for (m=0;m<M;m++)
    {
      debut=start_timer();

      for (i=0;i<N;i++)
for (j=0;j<N;j++)
  AF[i][j]+=YF[i][j];
      benchtime=dtime(debut, stop_timer());
      add_res(benchtime,m);
    }
  print_res ("ADD_ij", (double) (N*N));
  separateur();
}

void add_ji(){
  int i, j, m ;

  for (m=0;m<M;m++)
    {
      debut=start_timer();

      for (j=0;j<N;j++)
for (i=0;i<N;i++)
  AF[i][j]+=YF[i][j];
      benchtime=dtime(debut, stop_timer());
      add_res(benchtime,m);

    }
  print_res ("ADD_ji", (double) (N*N));
  separateur();
}


// Produit scalaire
void ps()
{
  int i, j, k, m;

  for (m=0;m<M;m++)
    {
      debut=start_timer();

      SF=ZERO;
      for (i=0;i<(N*N);i++){
SF+=BF[i]*CF[i];
```

```
      }
      benchtime=dtime(debut, stop_timer());
      add_res(benchtime,m);

    }
  print_res ("PS", (double) (N*N));
  separateur();
}



// Multiplication de matrices ijk
void mm_ijk()
{
  int i, j, k, m;

  for (m=0;m<M;m++)
    {
      debut=start_timer();

      for (i=0;i<N;i++)
for (j=0;j<N;j++)
  {
    SF=ZERO;
    for (k=0; k<N; k++)
      SF+=AF[i][k]*XF[k][j];
    YF[i][j]=SF;
  }
      benchtime=dtime(debut, stop_timer());
      add_res(benchtime,m);

    }
  print_res ("MM_ijk", ((double) N*N*N));
  separateur();
}


// Multiplication de matrices ikj
void mm_ikj()
{
  int i, j, k, m;
  for (m=0;m<M;m++)
    {
      debut=start_timer();
      for (i=0;i<N;i++)
for (k=0;k<N;k++)
  {
```

```
    SF=AF[i][k];
    for (j=0; j<N; j++)
      YF[i][j]+=SF*XF[k][j];
  }
      benchtime=dtime(debut, stop_timer());
      add_res(benchtime,m);

    }
  print_res ("MM_ikj", ((double) N*N*N));
  separateur();
}


// Multiplication de matrices par blocs
void mm_b_ijk(){
  int i, j, k, m, ii, jj, kk;

  for (m=0;m<M;m++)
    {
      debut=start_timer();

      for (jj=0;jj<N;jj+=BL)
for (kk=0;kk<N;kk+=BL)
  for (i=0;i<N;i++)
    {
      for (j=jj;j<min(jj+BL-1,N);j++)
{
  SF=ZERO;
  for (k=kk;k<min(kk+BL-1,N);k++)
    SF += AF[i][k]*XF[k][j];
  YF[i][j]=SF;
}
    }

      benchtime=dtime(debut, stop_timer());
      add_res(benchtime,m);
    }
  print_res ("MM_B_ijk", ((double) N*N*N));

  separateur();
}


int main()
{
  // Commenter et décommenter les appels de fonctions suivant les questions du TP.
  printf("Evaluation : N=%d, type="STR(TYPE)"\n",N);
```

```
  //zero();
  //copy_ij();
  //copy_ji();
  //add_ij();
  //add_ji();
  //ps();
  //mm_ijk();
  mm_ikj();
  //mm_b_ijk();

}
```