TDP 1 - PRCD
Mise en oeuvre des BLAS

Etienne THIERY - Youssef SEDRATI

20 octobre 2016

1 Prélude

1.1 Questions

La leading dimension d'une matrice stockée par colonne A est l'offset en mémoire entre l'élément $A_{i,j}$ et l'élément $A_{i+1,j}$, c'est à dire l'élement sur la même ligne, mais décalé d'une colonne. Plus généralement, la leading dimension est l'offset en mémoire pour passer d'un indice à l'autre dans la première dimension.

Les trois types de routines BLAS 1 , 2 et 3 correspondent respectivement aux opérations vecteur/vecteur, matrice/vecteur et matrice/matrice.

La première lettre d'une routine BLAS désigne le type des élements manipulés : `s` pour des réels simple précision, `d` pour des réels double précision, `c` pour des complexes simple précision, `z` pour des complexes double précision. Le reste spécifie l'opération. Ainsi la routine `zdot` réalise un produit scalaire entre vecteurs de nombres complexes en double précision. Et la routine `saxpy` réalise l'opération $y = a * x + y$ sur des vecteurs de nombres réels en simple précision.

Les CPUs des noeuds de Plafrim sont des Intel Xeon E5-2670, à 20 cores, cadencés à 2.5 GHz, et capables d'exécuter 16 opérations flottantes par cycle (avec AVX et FMA, 8 additions et 8 multiplications). Nous estimons donc la pic théorique à $20 * 2.5 * 16 = 800 GFlops/s$.

1.2 Benchmarking

Les fichiers `util.c|h` contiennent des routines d'allocations de vecteurs et matrices (alignés sur 16 octets pour permettre l'utilisation plus rapide de vecteurs AVX de taille allant jusqu'à 256 bits), de libération de ceux-ci, et d'affichage de portions de ceux-ci.

Les fichiers `benchtools.c|g` contiennent des outils de benchmark un peu plus poussés que ceux fournis initialement.

Nous avons tout d'abord développé une méthode permettant de détecter les types et tailles des différents caches présents sur la machine (uniquement pour les processeurs intels). Cela permet notamment de différencier lors de nos tests le cas où les opérandes sont en cache, et celui où ils ne le sont pas [1]. Pour obtenir un rapport sur les caches de la machine :

```
./myLib -i
```

Nous avons aussi développé un Timer amélioré qui permet de calculer automatiquement la durée moyenne d'une opération répétée un nombre donné de fois, sa vitesse moyenne en flops par seconde, mais aussi de mesurer sous Linux les nombres moyens :

— d'instructions exécutées
— de Cache Miss sur le cache L1
— de Cache Miss sur le cache de dernier niveau

1. Nous utilisons pour cela les méthodes MultiTimingSamples, et surtout MultCallFlushLRU présentées dans l'article http://www.csc.lsu.edu/~whaley/papers/timing_SPE08.pdf

Cela permet de mieux interpréter les résultats obtenus, et notamment de déterminer si nos implémentations sont Compute-Bound ou Cache-Bound.

1.3 Tracé des courbes

Tous les graphes présents dans ce rapport peuvent être reproduits à l'aide de gnuplot et des scripts présents dans le répertoire `graph`.

1.4 Compilation

Pour compiler notre projet sur Plafrim, les modules suivants sont nécessaires :
- `compiler/gcc/4.8.4` (pour une raison inconnue, la version 5.1.0 de gcc disponible sur Plafrim n'apprécie pas une réduction OpenMP, d'où l'utilisation de cette version moins récente)
- `compiler/intel/64/2016`
- `intel/mkl/64/11.2/2016.0.0`
- `intel-tbb-oss/intel64/43_20150424oss`

L'ensemble du projet peut être compilé simplement à l'aide de la commande :
`make`

Pour compiler la version de `ddot` vectorisée à l'aide de l'extension vectorielle de gcc :
`make myLib CUSTOMFLAGS=-DVECTORIZED`

Pour compiler la version de `ddot` parallélisée (à l'aide d'open MP) pour des vecteurs de taille supérieure à un entier N :
`make myLib CUSTOMFLAGS='-DPARALLEL_THRESH=N'`
Pour utiliser OpenMP pour toutes les tailles, utiliser la valeur $N = 0$.

Les options de vectorisation et de parallélisation peuvent être combinées :
`make myLib CUSTOMFLAGS='-DVECTORIZED -DPARALLEL_THRESH=N'`

Les différentes options d'exécution seront données au fur et à mesure du rapport.

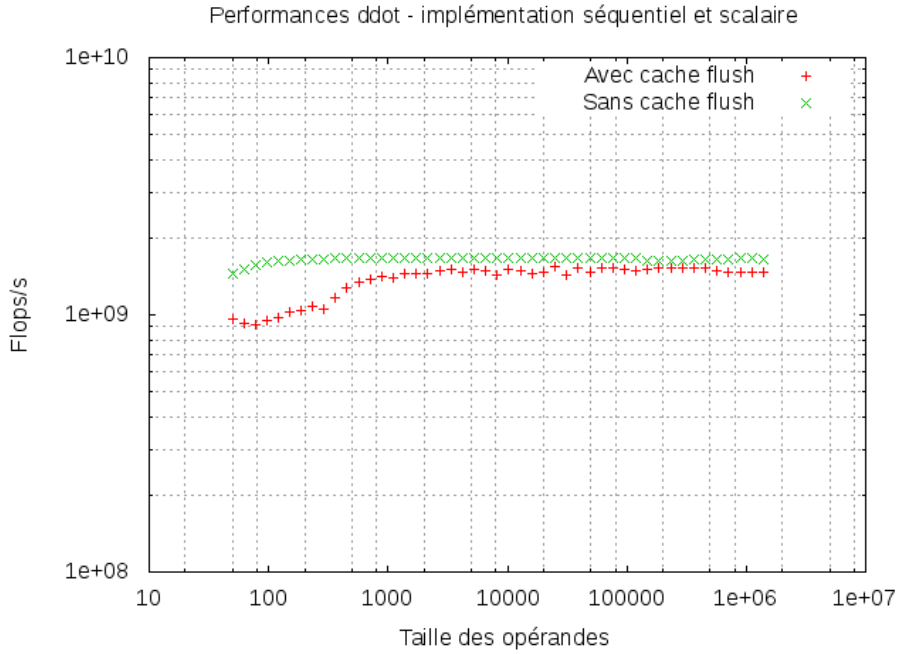
2 Produit scalaire

La routine `ddot` sur des vecteurs de dimensions N nécessite au minimum N produits + $N-1$ additions = $2*N-1$ opérations flottantes et $2*N$ lectures en mémoire.

Commençons par profiter de la simplicité de la version séquentielle non optimisée de `ddot` pour étudier les conséquences de la présence ou non des opérandes dans le cache. Générons la figure 1 :

```
make myLib
./myLib -b ddot > data/myLib_ddot_scalar_sequential_withoutFlush.dat
./myLib -b ddot -f > data/myLib_ddot_scalar_sequential_withFlush.dat
cd data ; gnuplot fig1.p ; cd ..
```

FIGURE 1 – Exploitation des caches par ddot

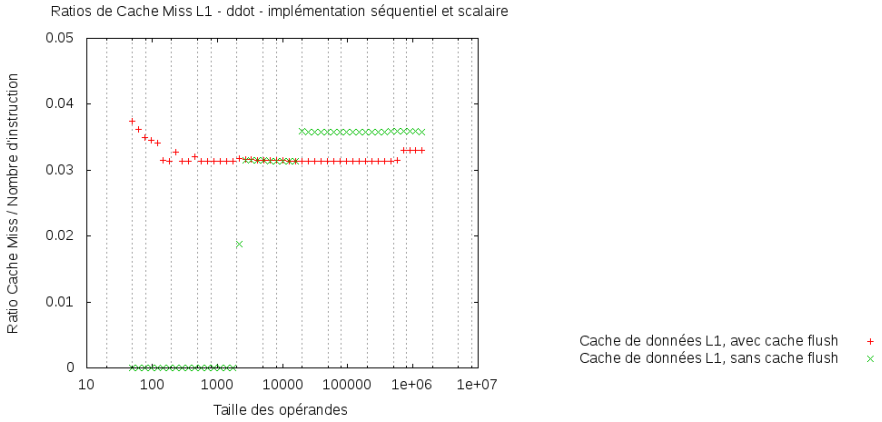


On peut constater sans surprise que lorsqu'on s'assure que les opérandes ne sont pas dans les caches avant chaque appel à `ddot`, les performances sur des petits vecteurs chutent. Explication probable : sachant que le cache de données L1 du noeud de Plafrim utilisé fait 32kB, il peut contenir 2 opérandes d'environ 2048 doubles. Une fois cette taille dépassé, le cache L1 est saturé et donc l'écart de performance se tasse. On peut toujours observer un petit écart, puisqu'il est toujours plus rapide de charger les données depuis les caches L2 et L3 que depuis la mémoire principale.

Cette explication est confirmée par l'observation des bonds dans la proportion de Cache Miss L1 par rapport au nombre d'instructions lorsque les caches ne sont pas vidés, sur la figure 2 générée par :

```
cd data ; gnuplot fig2.p ; cd ..
```

FIGURE 2 – Saturation du cache L1 avec la croissance de la taille des opérandes



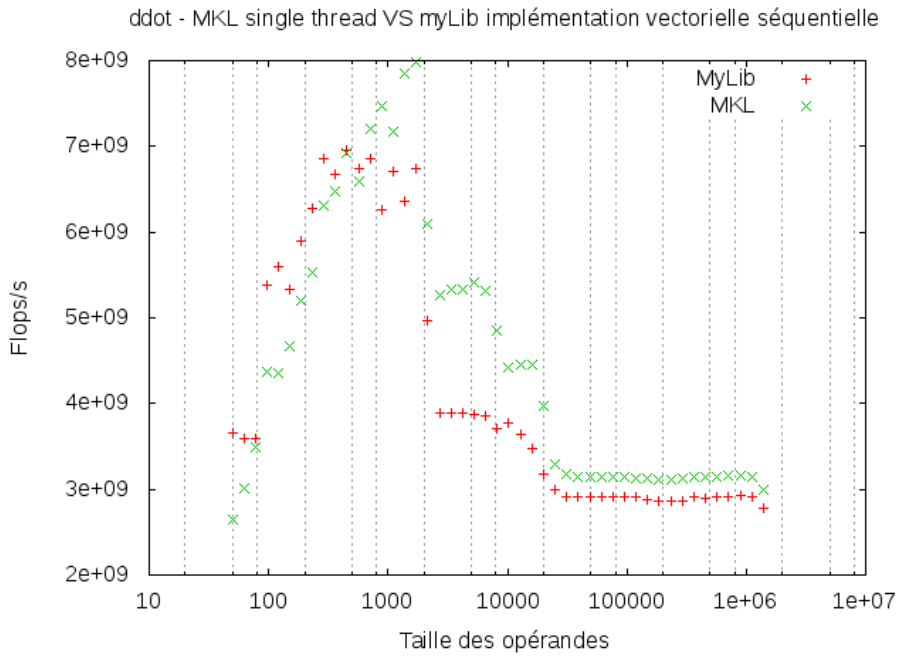
Cette implémentation semble donc cache-bound, au moins pour des opérandes dont la taille dépassent celle du cache L1. Pour la suite de ce rapport, on considère toujours des tracés obtenus en préchargeant les opérandes dans les caches, scénario qui nous semble assez réaliste puisqu'on

initialise généralement les opérandes peu avant d'effectuer des opérations dessus.

Nous avons optimisé `ddot` en utilisant l'extension vectorielle de GCC (après avoir essayé les instrinsics AVX, un peu moins efficaces), atteignant des performances très honorables face à MKL en single thread, représentées en figure 3, obtenue via :

```
make myLib CUSTOMFLAGS=-DVECTORIZED
make mkl
./myLib -b ddot  > data/myLib_ddot_vectorial_sequential_withoutFlush.dat
OMP_NUM_THREADS=1 ./mkl -b ddot  > data/mkl_ddot_singleThread_withoutFlush.dat
cd data ; gnuplot fig3.p ; cd ..
```

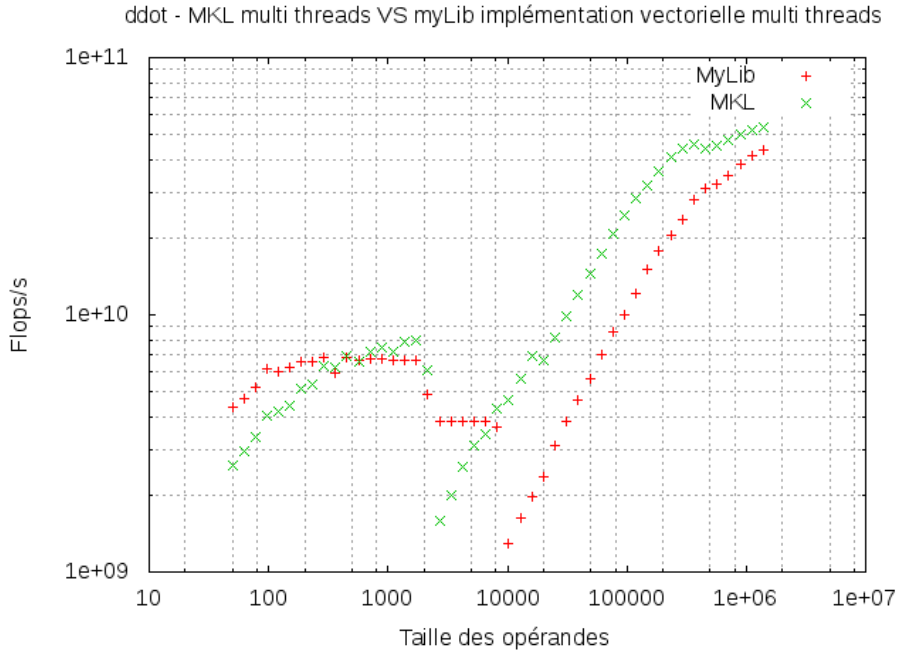
FIGURE 3 – Comparaison en single thread de la version vectorielle de ddot avec MKL



Pour finir avec ddot, nous avons parallélisé notre implémentation en utilisant OpenMP. Pour des vecteurs de petite taille, l'overhead de la création des threads est plus important que les gains dus au parallélisme. C'est pour cette raison que nous avons ajouté le flag de compilation `PARALLEL_THRESH`, qui permet de choisir statiquement une taille d'opérandes en dessous de laquelle utiliser la version séquentielle et au dessus de laquelle utiliser la version parallèle. Cette valeur pourrait être tunée à la compilation, pour obtenir les meilleures performances pour toutes les tailles d'opérandes. La figure 4 montre un exemple avec un seuil de 10000 sur un noeud de Plafrim, comparé au performance de MKL sur le même nombre de threads. On constate qu'on pourrait augmenter légèrement le seuil pour avoir de meilleures performances sur des opérandes de tailles 10000-20000. On observe également que les résultats de MKL sont similaires, la même méthode est sûrement utilisée.

```
make myLib CUSTOMFLAGS='-DVECTORIZED -DPARALLEL_THRESH=10000'
make mkl
OMP_NUM_THREADS=20 ./myLib -b ddot  > data/myLib_ddot_vectorial_parallel_withoutFlush.dat
MKL_NUM_THREADS=20 ./mkl -b ddot  > data/mkl_ddot_multiThread_withoutFlush.dat
cd data ; gnuplot fig4.p ; cd ..
```

FIGURE 4 – Comparaison en multi threads de la version vectorielle de ddot avec MKL



3 Produit matriciel

3.1 Version séquentielle

Nous avons implémenté directement la forme plus générale : $C = \alpha {}^tA \times B + \beta C$ sur n'importe quelles matrices rectangulaires.

Avec une implémentation séquentielle classique avec une triple boucle imbriquée, la complexité varie selon l'ordre des boucles. En considérant le produit d'une matrice $M \times K$ par une matrice $K \times N$:

- pour les ordres (i, j, k) et (j, i, k), on peut calculer ${}^tA \times B$ en $M * N * K$ produits et $M * N * (K - 1)$ sommes, puis effectuer la combinaison linéaire avec C en $M * N * 2$ produits et $M * N$ sommes. La complexité en calculs totale est donc de $M * N * (2 * K + 2)$ opérations flottantes.
- pour l'ordre (k, i, j) en revanche, il faut nécessairement ajouté avant la triple boucle une double boucle pour effectuer le produit βC en $M * N$ produits (voir implémentation). Puis la triple boucle demande $K * M * N * 2$ produits et $K * M * N$ sommes. La complexité totale en calculs est donc de $M * N * (3 * K + 2)$.

Les 3 implémentations scalaires se trouvent dans les fichiers `dgemm_versions.c|h`. Nous avons tenu compte de la différence de complexité pour la comparaison de leurs vitesses en flops/s. Les figures 6 et **??** peuvent être obtenue avec les commandes :

```
make myLib
./myLib -b dgemm_scalar_ijk > data/myLib_dgemm_scalar_ijk_withoutFlush.dat
./myLib -b dgemm_scalar_jik > data/myLib_dgemm_scalar_jik_withoutFlush.dat
./myLib -b dgemm_scalar_kij > data/myLib_dgemm_scalar_kij_withoutFlush.dat
cd data ; gnuplot fig5.p ; cd ..

cd data ; gnuplot fig6.p ; cd ..
```

FIGURE 5 – Influence de l'ordre des boucles sur la performance en temps

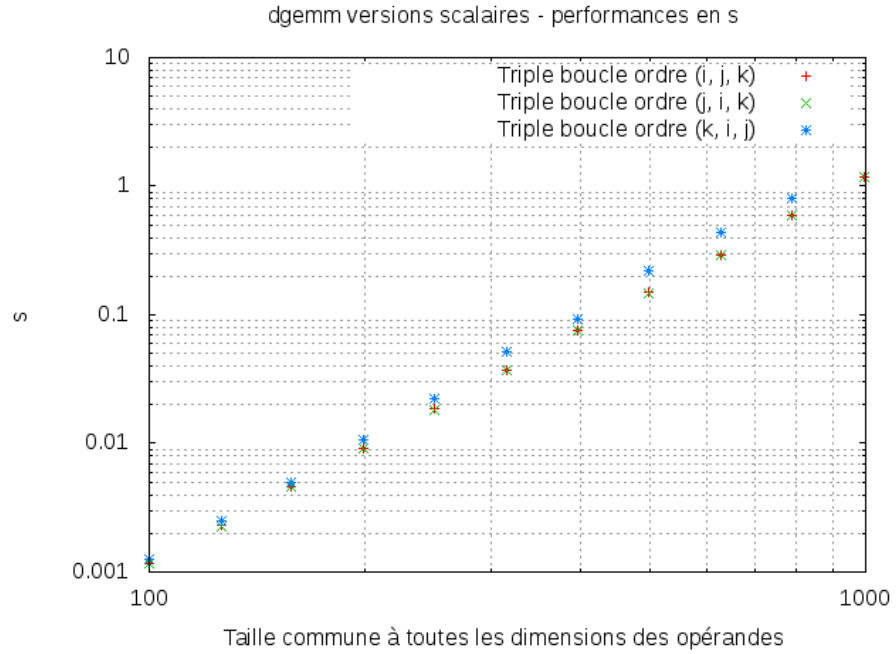
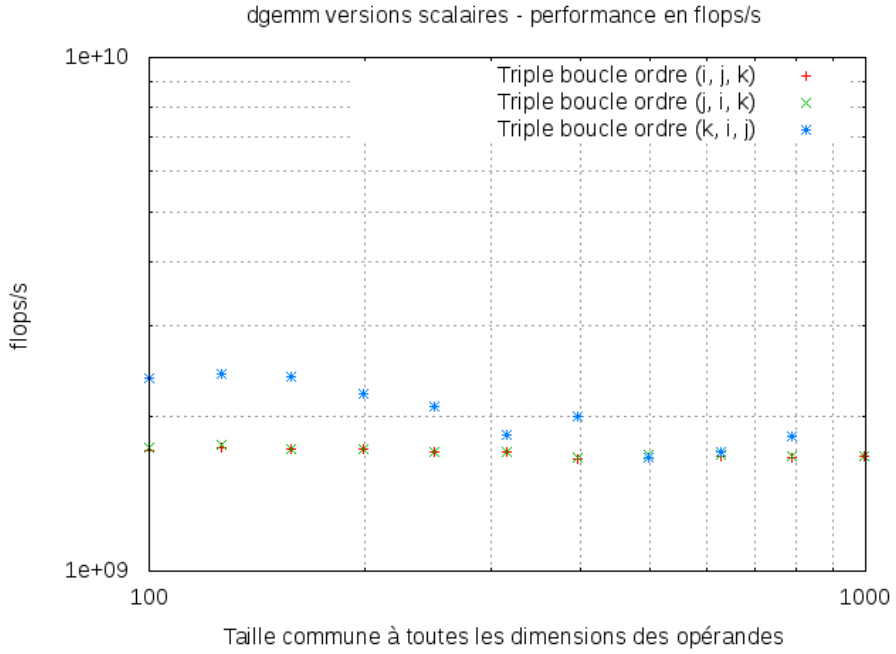


FIGURE 6 – Influence de l'ordre des boucles sur la performance en flops/s



Quelques commentaires sur les résultats :

— Les performances pour les ordres (i, j, k) et (j, i, k) sont les mêmes, logiquement. D'une part ces deux implémentations font le même nombre d'opérations. D'autre part on parcourt A avec l'indice i, B avec l'indice j, et ces deux matrices sont de même taille dans le cas présent, donc le problème est totalement symétrique. Peu importe l'ordre des 2 premières boucles on profite toujours d'une lecture d'éléments consécutifs pour l'une des deux matrices.

— Les performances en flops sont ici très légèrement plus élevées pour l'ordre (i, j, k). C'est probablement à cause de la double boucle calculant βC et profitant bien du préfetching

avec l'ordre (i, j). Ses performances en temps sont bien sur moins bonnes, la différence en nombre d'opération flottantes effectuées étant assez conséquente. Notons que si l'on faisait la même étude avec un beta constant fixé à 1, pour se débarasser de cette double boucle, on s'attendrait à ce que l'ordre (k, i, j) soit moins performant, car avoir la boucle indicé par k à l'intérieur permet d'accéder dans l'ordre à des éléments consécutifs des matrices A et B.

3.2 Version par bloc

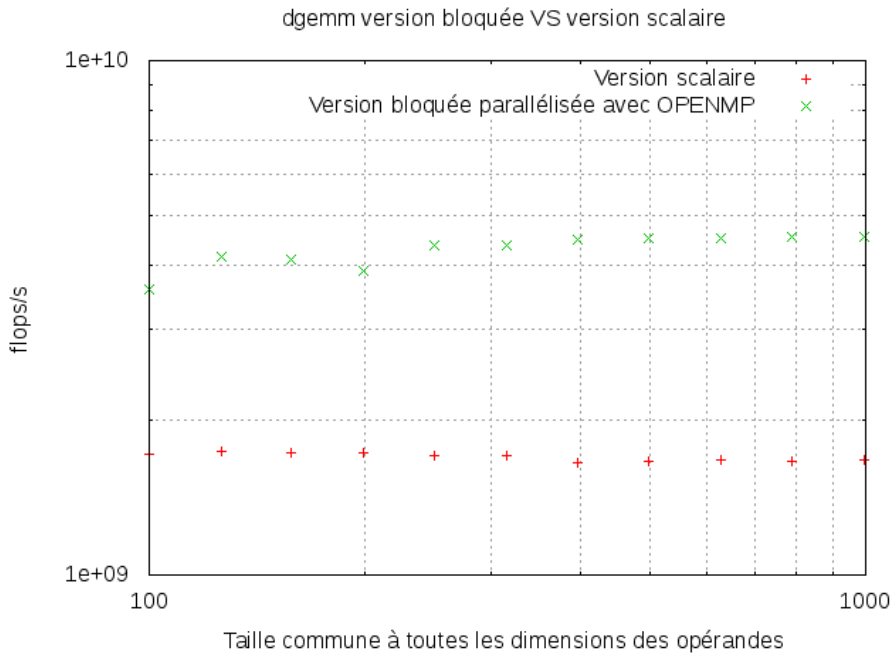
Avec la version par bloc, on augmente la "localité temporelle", c'est à dire l'aspect parallèle du code. On peut éventuellement avoir un speed-up du à l'Out of Order.

Nous avons développé une implémentation par bloc s'appuyant sur la version scalaire (i, j, k), et nous l'avons parallélisé à l'aide d'openMP.

Elle passe les tests, mais ses performances sont bien loin du pic théorique de la machine, et le speedup bien loin d'être idéal... Nous n'avons pas eu le temps d'en faire plus !

```
make myLib
OMP_NUM_THREADS=20 ./myLib -b dgemm_bloc > data/myLib_dgemm_bloc_withoutFlush.dat
./myLib -b dgemm_scalar_ijk > data/myLib_dgemm_scalar_ijk_withoutFlush.dat
cd data ; gnuplot fig7.p ; cd ..
```

FIGURE 7 – Parallélisation du produit par bloc



Nous n'avons pas affiché sur cette figure les performances de MKL, meilleures d'un facteur 10 à 100 selon la taille des vecteurs. Noter que lorsque compilé avec la version 4.8.4, tout appel à l'implémentation MKL de `dgemm` par notre driver provoque des lectures invalides à l'intérieur de la librairie. Elle marche normalement avec les versions supérieures, mais on ne peut alors plus compiler myLib à cause d'une réduction OpenMP qui ne semble plus valide.

```
make mkl
./mkl -b dgemm_bloc
```