Chapitre 9

« Memory Wall », performance de crête et équilibrage

Dans le fond, l'objectif du calcul haute-performance consiste à écrire des programmes qui atteignent des performances les plus proches possibles du maximum possible sur les machines parallèles utilisées.

9.1 Performance de crête d'une machine

Quand il s'agit du matériel, on parle de *performance de crête* (*Peak performance*), et on l'exprime typiquement en FLOPS (*Floating Point OPerations per Second*). Cette grandeur quantifie la puissance de calcul du matériel. Il s'agit généralement d'un maximum *théorique*, qui est rarement atteint en utilisation normale.

Sequoia Il y a des cas où son calcul est relativement simple.

Par exemple, la machine `sequoia` (IBM, 2012, Lawrence Livermore National Laboratory). Elle a 98 304 noeuds, chacun avec un processeur PowerPC A2 de 16 coeurs à 1.6GHz. Ces processeurs sont simples (in-order, pas superscalaires) et n'exécutent qu'une seule instruction par cycle dans le meilleur des cas. Chaque coeur dispose d'une unité vectorielle (« SIMD ») qui opère sur des vecteurs de quatre flottants double précision ($4 \times 64 = 256$ bits). Celle-ci est capable d'effectuer une opération « *Fused-Multiply Add* » (FMA) par cycle. Chaque coeur peut donc faire 8 FLOP par cycle (4 multiplications et 4 additions), dans le meilleur des mondes. La puissance de crête d'un de ces processeurs est donc de $1.6 \times 10^9 \times 16 \times 8 = 204.8$ GigaFLOPS. Sur l'ensemble de la machine, cela ferait donc $98\,304 \times 204.8 = 20.1$ PetaFlops.

Le même genre de calcul peut à peu près se faire sur une partie importante des processeurs courants pas trop récents, à ceci près qu'ils sont superscalaires, et donc qu'ils peuvent éventuellement exécuter *plusieurs* instructions par cycle (il faut trouver combien).

Jean-Zay Et il y a des cas où le calcul est franchement plus compliqué.

Penchons-nous sur `jean-zay`, la machine récemment acquise par le CNRS (on ne s'intéresse qu'à la partie CPU, on laisse les GPUs de côté, et c'est déjà assez compliqué comme ça). Elle dispose de 1528 noeuds possédant deux processeurs Intel Xeon Gold 6248 (Cascade Lake) de 20 coeurs à 2.5Ghz avec l'AVX512. Cette unité SIMD est capable de faire des opérations sur 8 flottants en double précision (ou 16 en simple précision).

Première observation, on peut sur le papier faire deux fois plus de FLOPS avec des `float` qu'avec des `double`... Mais on va se limiter aux `double` pour l'instant et pour simplifier.

L'AVX512 possède une instruction FMA. Les Xeon Scalable d'entrée de gamme (Bronze et Silver) peuvent faire un FMA 512-bits par cycle, tandis que les haut de gamme (Gold et Platinum) peuvent en faire deux.

Dans notre cas, on a donc le droit à deux FMA par cycle sur 8 `double`, donc à 32 FLOP par cycle sur chaque coeur. Et maintenant, les problèmes commencent. Combien y a-t-il de cycle par seconde ? On nous dit « 2.5Ghz », mais en fait c'est plus compliqué que ça. Ces processeurs ont la capacité d'augmenter la fréquence des coeurs individuellement, tant que la puissance dissipée n'est pas trop élevée. Commercialement, cela s'appelle le *Turbo Boost* : ils sont tous capable de fonctionner ensemble à 2.5Ghz (fréquence de base normale), mais si un seul est actif il peut aller jusqu'à 3.9Ghz (fréquence turbo normale).

Mode	Base	Turbo avec x coeurs actifs					
		1-2	3-4	5-8	9-12	13-16	17-20
normal	2.5	3.9	3.7	3.6	3.6	3.4	3.2
AVX2	1.9	3.8	3.6	3.5	3.4	3.0	2.8
AVX512	1.6	3.8	3.6	3.5	3.0	2.7	2.5

FIGURE 9.1 – Fréquence limites pour le Xeon Gold 6248 (Cascade Lake). Chaque coeur peut se trouver entre la fréquence de base et la fréquence turbo du mode concerné. Ces chiffres varient d'un modèle à l'autre.

Et en réalité, c'est encore plus compliqué. Si un coeur ne fait que des opérations scalaires ou vectorielles AVX2 « légères » (pas de flottants ni de multiplication/division), alors il est en mode « normal ». S'il fait beaucoup d opérations vectorielles AVX2 lourdes (ou des AVX512 légères), alors il passe en mode AVX2 et sa fréquence *baisse*. Enfin, s'il fait des opérations AVX512 lourdes, il passe en mode AVX512 et sa fréquence baisse encore.

Du coup, chaque coeur peut fonctionner à une fréquence différente en fonction du mode dans lequel il se trouve et de combien d'autres sont actifs. Le tableau 9.1 montre les bornes inférieures et supérieures. Au passage, on voit qu'en passant d'un code scalaire à un code AVX512, on gagne plutôt un facteur 5.1 qu'un facteur 8 (à cause de la réduction potentielle de la fréquence de 36%).

Du coup, on sait que quoi qu'il arrive, on peut atteindre 1.6Ghz sur les 20 coeurs à la fois en faisant de l'AVX512. Ceci donnerait $1.6 \times 10^9 \times 20 \times 32 = 1024$ GFLOPS par processeur, donc $1528 \times 2 \times 1024 = 3.1$ PFLOPS pour l'ensemble de la machine.

9.1.1 Peut-on atteindre la performance de crête ?

Réponse courte : ça dépend mais probablement pas (malgré des efforts importants).

En tout cas, sans faire d'efforts, c'est sûr qu'on en sera très très loin ! Le produit de matrice naïf qui apparaît au début du chapitre 1 atteint péniblement 1.6 GFLOPS sur un noeud de `jean-zay` capable d'en faire 2048 « en crête ». On est donc à 0.08% de la puissance de crête *d'un seul noeud* !

```
/* très mauvais */
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++)
            C[i][j] += A[i][k] * B[k][j];
```

Il faut bien se comprendre : le but est de finir les calculs le plus vite possible, pas de faire le plus de FLOPS possible. Si on a un algorithme $\mathcal{A}$ (par ex. un solveur linéaire creux) qui termine en 10 minutes en faisant peu de FLOPS, il vaut mieux l'utiliser plutôt que l'algorithme $\mathcal{B}$ (par ex. un solveur linéaire dense) qui fait plein de FLOPS mais qui termine en trois jours...

Les petits calculs ci-dessus montrent que pour atteindre la performance de crête d'une machine parallèle, il faut forcément :

- Utiliser tous les noeuds à 100% (pas de temps de communication et bon équilibrage de charge).
- Utiliser tous les coeurs à 100% (pas de surcoût et bon équilibrage de charge).
- Utiliser la vectorisation, et en particulier utiliser quasi-exclusivement des FMA.
- Exploiter le parallélisme au niveau des instructions (ILP) pour fournir aux unités d'exécutions deux FMA « prêts à exécuter » par cycle.

Obtenir de bonnes performances nécessite donc forcément des efforts de programmation non-triviaux, et/ou l'usage de librairies « sur étagères » (*off-the-shelf*) très optimisées.

Mais de toute façon, là-dessus vient se greffer le problème du *Memory Wall* abordé dans le chapitre 8. Si la mémoire n'est pas capable de fournir les données aux unités d'exécution, alors on n'a aucune chance de s'approcher de la puissance de crête. Et en réalité, beaucoup de calculs sont *memory-bound*, c'est-à-dire limités par la bande passante de la mémoire.

9.2 Modèle « *Roofline* »

Généralement, dans le monde du calcul scientifique de pointe, on cherche à optimiser un petit nombre d'applications pour *une* machine donnée (celle qu'on a sous la main).

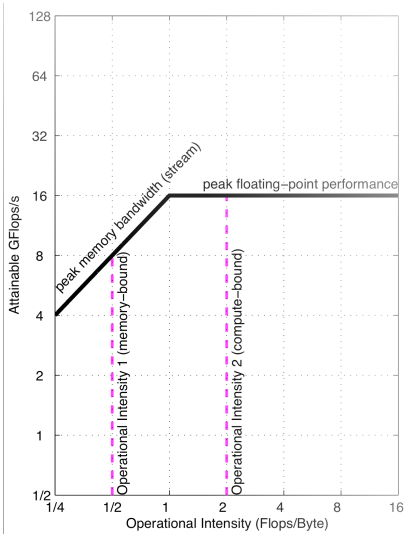

FIGURE 9.2 – Un *Roofline* dépouillé pour une machine fictive avec un seul processeur. La performance de crête d'est de 16GigaFLOPS, et la bande passante mémoire de 16Go/s. La ligne noire représente le min de la formule. Les deux lignes pointillées représentent deux applications différentes qui ont deux intensités opérationnelles différentes. L'application 1 est *memory-bound*, tandis que l'application 2 est *compute-bound*. (image : [16])

Par exemple, quand le CNRS a fait l'acquisition de `jean-zay`, le fournisseur devait s'engager à porter 6 applications importantes pour la communautés des utilisateurs, avec un niveau de performance garanti par contrat...

Quand on est confronté au problème d'optimiser *une* application particulière sur *une* machine donnée, il faut d'abord connaître les limites de la machine, mesurer les performances de l'application, puis chercher à comprendre ce qui peut les limiter. Est-ce que le problème c'est que les unités arithmétiques n'arrivent pas à faire les FLOP assez vite ? Est-ce la mémoire qui ne peut pas fournir les données assez vite ? Est-ce le système de fichier qui est trop lent ? Etc.

9.2.1 Version dépouillée

Le modèle « Roofline » est une représentation graphique simple qui aide à comprendre les éventuelles limites aux performance d'une application donnée sur une machine donnée. Il a été introduit en 2009 par Samuel Williams, Andrew Waterman, and David Patterson dans « *Roofline : an insightful visual performance model for multicore architectures* » [16].

Le modèle vise spécifiquement à comprendre si on se heurte au *Memory Wall* ou pas ; l'article (dont la lecture est assez facile et recommandée) dit en effet :

> We believe that for the recent past and foreseeable future, off-chip memory bandwidth will often be the constraining resource. Hence, we want a model that relates processor performance to off-chip memory traffic.

Le modèle repose sur la notion d'*intensité opérationnelle* du calcul effectué, c'est-à-dire le nombre d'opérations effectuées par octet transféré depuis la RAM (c.a.d. transféré entre le contrôleur mémoire et les barrettes de RAM, et *non* transféré entre un coeur et la hiérarchie des caches).

l'intensité opérationnelle relie entre elles les performances de l'applications (en FLOPS) avec la bande passante « de crête » de la mémoire. En effet, on a assez logiquement :

$$\text{FLOPS} \leq \min\big([\text{FLOPS de crête}], [\text{intensité opérationnelle}] \times [\text{bande passante RAM de crête}]\big)$$

Le modèle *Roofline* représente ceci en deux dimensions (cf. figure 9.2) : l'axe des abscisses représente l'intensité opérationnelle et l'axe des ordonnées représente le nombre de FLOPS. Une grosse ligne représente la limite supérieure donnée par la formule ci-dessus : c'est une caractéristique indépassable de la machine elle-même. Sur ce graphique, la ligne horizontale représente la performance de crête, tandis que la ligne oblique représente la limite imposée par la bande passante de la mémoire. On voit que plus l'intensité opérationnelle est faible, plus les performances de l'application ont peu de chance d'atteindre le maximum potentiel. Dans l'autre sens, pour espérer atteindre les performances de crête, il *faut* avoir une intensité opérationnelle élevée.

Un premier avantage de ce schéma simple, c'est que cela donne une première idée de ce qu'on peut attendre d'une application d'intensité opérationnelle donnée sur une machine donnée.

Des techniques simples permettent parfois d'améliorer un peu l'intensité opérationnelle, comme la fusion des boucles :

```
/* Mauvais */
for (int i = 0; i < N; i++)
    A[i] = B[i] * C[i];
for (int i = 0; i < N; i++)
    D[i] = B[i] + E[i];
```

```
/* Meilleur */
for (int i = 0; i < N; i++) {
    A[i] += B[i] * C[i];
    D[i] += B[i] + E[i];
}
```

Si N est suffisamment grand, les trois tableaux ne tiendront plus dans le cache et les données vont devoir être chargées depuis la RAM. Chaque itération des deux boucles du « mauvais » code transfère 3 `double` de/vers la RAM et effectue une opérations (IO = 1/24). Par contre, le « bon » code (qui calcule la même chose) transfère 5 `double` et effectue 2 opérations (IE = 1/20). Le truc c'est que $B[i]$ n'a pas besoin d'être rechargé une deuxième fois. Les compilateurs effectuent parfois cette optimisation automatiquement.

9.2.2 Ajout des « plafonds »

Si on part d'un code peu ou pas optimisé (séquentiel, pas vectorisé, etc.), un des intérêt des schémas *Roofline* c'est qu'ils donnent une idée de ce qu'on pourrait potentiellement gagner en mettant en oeuvre tel ou tel type d'optimisation.

On peut faire figurer sur le diagramme les limites correspondant à des optimisations qu'on envisage (passer à la vectorisation, changer les algorithmes pour améliorer la localité des accès mémoire, etc.). Sur une machine fixée (par exemple, un coeur de `jean-zay`), on peut espérer faire 2.5 GFLOPS avec des opérations scalaires uniquement, mais 7.6 GFLOPS si on fait de la vectorisation AVX2, 12.8 GFLOPS avec de la vectorisation AVX512, 25.6 GFLOPS si on utilise des FMA, et enfin 51.2 si on arrive à en faire deux par cycle. Cela donne différends « plafonds » atteignable selon le niveau d'optimisation déployé, et on peut les voir sur la figure 9.3.

9.2.3 Quelques exemples

Produit scalaire Prenons un exemple de code de calcul simple : le produit scalaire

```
double res = 0;
for (int j = 0; j < N; j++)
    res += A[i] * B[i];
```

Il y a 2 `double` chargés par itération (16 octets) et deux opérations arithmétiques. Si les données ne tiennent pas en cache, l'intensité opérationnelle est donc 1/8. Si on regarde la droite verticale d'équation $x = 1/8$ sur la figure 9.3, on voit que ce petit bout de code est (probablement) limité par la capacité du CPU à effectuer des opérations arithmétiques en mode « scalaire » : le plafond « Peak FLOPS (scalar) » est en dessous de la bande passante de la RAM pour une intensité opérationnelle de 1/8. La limite imposée par la bande passante de la RAM est 25.2Go/s $\times 1/8 = 3.15$ GFLOPS. Si N est grand et que les données ne tiennent pas en RAM, alors on a intérêt à mettre en oeuvre une forme de vectorisation (par exemple avec des instructions AVX2), car on va y gagner un peu (passer de 2.5GFLOPS à 3.15). Par contre, ce n'est pas la peine de se casser la tête à écrire du code assembleur ultra-optimisé à la main : une fois qu'on a atteint 3.15 GFLOPS, c'est cuit !

Par contre, si les vecteurs sont petits et tiennent en cache L1, on peut espérer atteindre $300/8 = 37$ GFLOPS, mais pour ça on voit qu'il faut dépasser les plafonds « AVX2 », « AVX512 », « FMA », etc. Donc il va falloir s'assurer que le code est vraiment bien vectorisé.

Petit produit matriciel Prenons un autre exemple : on calcule $A \leftarrow A + \sum_{u<N} B_u \times C_u$, où A, B_u et C_u sont des matrices 8×8.

```
for (int u = 0; u < N; u++)
    for (int i = 0; i < 8; i++)
        for (int j = 0; j < 8; j++)
            for (int k = 0; k < 8; k++)
                A[u*64 + i*8 + j] += B[u*64 + i*8 + k] * C[u*64 + k*8 + j];
```

Chaque itération de la boucle externe charge 128 `double` depuis la mémoire (B_u et C_u), ce qui fait 1024 octets, et effectue 1024 opérations arithmétiques — pendant le produit de matrice lui-même, les données sont en cache.

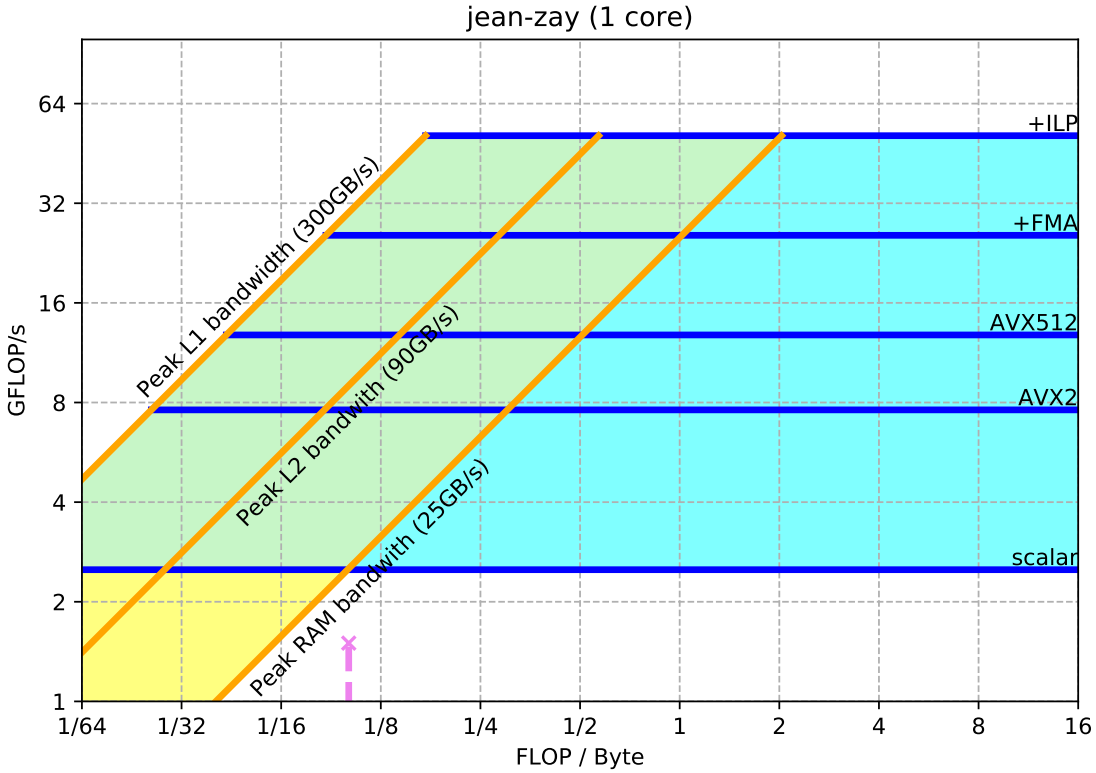

FIGURE 9.3 – Un schéma *Roofline* pour un coeur d'un processeur de `jean-zay`. On voit différents « plafonds ». La ligne pointillée violette représente les performances obtenues par un code scalaire particulier (résolution de $Ax = b$ par la méthode du gradient conjugué).

L'intensité opérationnelle est donc de 1. On voit sur la figure 9.3 qu'on peut espérer atteindre 25.2 GFLOPS, mais à condition de mettre en oeuvre une vectorisation AVX512 qui inclut des FMA.

Gradient conjugué Pour illustrer la méthode, on a fait figurer (en violet) sur la figure 9.3 les performances obtenues par un code scalaire particulier. Dans certains cas (mais pas tous !), il passe effectue essentiellement son temps à faire des produits matrice creuse-vecteur dense. Ceci a une intensité opérationnelle de ≈ 0.1 (2 FLOP pour 2 `double` et un `int` transféré). Il atteint 1.5 GFLOPS sur un coeur de `jean-zay`, ce qui est très faible.

On voit qu'on peut espérer améliorer un peu les choses en tentant d'améliorer le code, mais qu'on sera coincé par la bande-passante de la mémoire à un niveau de performance assez bas (c'est tout le temps comme ça avec les matrices creuses). Pour s'en tirer, il faudrait augmenter l'intensité opérationnelle, probablement en réorganisant les données.

9.3 Modèle I/O simplifié

Au vu de ce qui précède, un moyen important d'améliorer l'efficacité des calculs consiste à augmenter l'intensité opérationnelle du code, c'est-à-dire à faire le même travail en chargeant moins de données depuis la RAM. Pour étudier tout ceci dans un cadre un peu plus précis, on considère le modèle de calcul suivant : un processeur est relié à une (petite) mémoire *rapide*, elle-même reliée à une (grosse) mémoire *lente*. Par exemple, « RAM vs. disque » ou bien « cache L1 vs RAM ». Le processeur ne peut lire que ce qui se trouve en mémoire rapide, mais c'est « gratuit ». On compte le nombre d'opérations arithmétiques réalisées par le processeur et la quantité de données transférées de/vers la mémoire lente. Dans ce modèle, on note N la taille du problème à traiter et M la taille de la mémoire rapide ; on note ν et μ le nombre de FLOP et de `double` transférés depuis la mémoire lente, respectivement ; enfin on note t_ν le temps nécessaire pour accomplir un FLOP et t_μ le temps pour transférer un `double` (avec l'idée que $\nu \lll \mu$).

Ce modèle est dit « simplifié » car dans le vrai modèle I/O, la mémoire rapide stocke des blocs de taille B, et les données ne peuvent être transférées entre les deux mémoires que par bloc. Le vrai modèle a été introduit à la fin des années 1980 par Aggarwal et Vitter pour étudier la complexité de certains algorithmes courants (tri, transposition, FFT, ...) qui traitent des données trop grosses pour tenir en RAM. Le transfert « par bloc » est réaliste : les caches transfèrent des lignes de cache, les disques transfèrent des secteurs, etc.

Dans ce modèle, l'intensité opérationnelle est précisément $\nu/8\mu$. Pour simplifier les choses, on pose $q = \nu/\mu$ (nombre de FLOP par `double` transféré). C'est un critère clef d'efficacité des algorithmes (plus il est élevé, mieux c'est).

Si toutes les données sont dans la mémoire rapide, alors le temps d'exécution minimal indépassable est $\nu \times t_\nu$. Mais en fait, il va y avoir des transferts, donc le temps réel est :

$$T = \nu \times t_\nu + \mu \times t_\mu = \nu \times t_\nu \left(1 + \frac{t_\mu}{t_\nu} \times \frac{1}{q}\right)$$

Autant la quantité q est une caractéristique de l'algorithme, autant $\frac{t_\mu}{t_\nu}$ est une caractéristique de la machine, qu'on nomme son « équilibre » (*machine balance*). C'est un critère clef d'efficacité de la machine. Plus la machine est équilibrée ($t_\mu \approx t_\nu$), plus elle est facile à programmer et plus il est facile d'atteindre de bonnes performances. A contrario, plus la machine est déséquilibrée ($t_\mu \ggg t_\nu$), plus c'est difficile.

La performance d'un algorithme donné sur une machine donnée dépend de l'adéquation entre l'équilibre de la machine et l'intensité opérationnelle de l'algorithme : les machines déséquilibrées (beaucoup de FLOPS, peu de Go/s) sont moins tolérantes aux algorithmes « peu intenses ».

De manière générale, lorsque q est faible (petite constante), les algorithmes ont tendance à être *memory-bound*. Mais on peut parfois les améliorer sensiblement en les réorganisant complètement.

Produit Matrice-Vecteur Considérons le produit matrice-vecteur $y \leftarrow y + Ax$. Supposons que la mémoire rapide est suffisamment grande pour contenir trois vecteurs de taille N ; alors on peut écrire :

```
/* charger x et y en mémoire rapide */
for (int i = 0; i < N; i++) {
    /* charger la i-ème ligne de A en mémoire rapide */
    for (int j = 0; j < N; j++)
        y[i] += A[i*N + j]*x[j];
```

```
}
/* écrire y en mémoire lente */
```

Cet algorithme s'exécute correctement dans le modèle IO simplifié, il fait $\nu = 2N$ opérations arithmétiques et transfère $\mu = (3+N)N$ flottants depuis la mémoire lente. On a donc $q \approx 2$ et l'intensité opérationnelle est faible ($1/4$). Dans la pratique, et cet algorithme est quasiment tout le temps limité par la bande passante de la mémoire — ceci dit, si on se reporte à la figure 9.3 on voit qu'il faut éventuellement se battre un peu pour l'atteindre, car il n'est pas dit que ce code C naïf y parvienne.

Produit Matrice-Matrice Le produit matriciel est un problème intéressant, car le nombre d'opérations arithmétiques qu'il nécessite ($2N^3$) peut devenir sensiblement plus grand que la taille des données manipulées ($2N^2$) — ce qui n'est pas le cas du produit scalaire ou du produit matrice-vecteur. On pourrait donc espérer avoir une bonne intensité opérationnelle, et donc des algorithmes efficaces sur bon nombre de machines.

Reprenons pour la n-ème fois le code :

```
for (int i = 0; i < N; i++) {
    /* charger la i-ème ligne de A en mémoire rapide */
    for (int j = 0; j < N; j++)
        /* charger C[i,j] en mémoire rapide */
        /* charger la j-ème colonne de B en mémoire rapide */
        for (int k = 0; k < N; k++)
            C[i*N + j] += A[i*N + k] * B[k*N + j];
        /* écrire C[i,j] en mémoire lente */
}
```

Dans ce code, on a $\nu = 2N^3$ et $\mu = N^3 + 3N^2$. Par conséquent, on a $q \approx 2$ et l'intensité opérationnelle n'est pas meilleure que celle du produit matrice-vecteur.

Mais on peut sensiblement améliorer les choses en faisant le produit « par bloc ». On considère nos matrices $N \times N$ comme des matrices $\frac{N}{B} \times \frac{N}{B}$ de blocs de taille $B \times B$.

```
for (int i = 0; i < N/B; i++)
    for (int j = 0; j < N/B; j++) {
        /* charger le bloc C[i,j] en mémoire rapide */
        for (int k = 0; k < N/B; k++) {
            /* charger le bloc A[i,k] en mémoire rapide */
            /* charger le bloc B[k,j] en mémoire rapide */
            C[i, j] += A[i, k] * B[k, j];
        }
        /* écrire le bloc C[i,j] en mémoire lente */
    }
```

Chaque transfert de bloc nécessite B^2 transfert de `double`. Il y $2(N/B)^2(1+N/B)$ transferts de bloc en tout, donc l'intensité opérationnelle est :

$$IO = \frac{1}{1/N + 1/B} \xrightarrow[N\to+\infty]{} B$$

Donc, plus la taille du bloc augmente, plus l'intensité opérationnelle augmente, et meilleures les performances devraient être. Par contre, pour que ça marche, il faut que trois blocs tiennent en mémoire rapide, donc que $3B^2 \leq M$, autrement dit que $B \leq \sqrt{M/3}$. C'est donc cette taille de bloc-là qu'on a intérêt à choisir. Pour obtenir les meilleures performances possibles, il faut un ajustement (« *tuning* ») de la taille du bloc à la taille du cache.

Un algorithme qui a besoin de connaître la taille de la mémoire rapide (du cache) pour s'exécuter avec les performances optimales est dit « *cache-aware* ». Il existe aussi des algorithmes « *cache-oblivious* » qui s'exécutent bien quelle que soit la taille du cache, et même s'ils ne la connaissent pas ! Mais en pratique ils sont parfois plus lent que des algorithmes *cache-aware* bien réglés.

9.4 Bibliothèques de calcul scientifique

Un certain nombre d'opérations mathématiques sont récurrentes dans le calcul scientifique : algèbre linéaire dense, algèbre linéaire creuse, transformée de fourrier rapide, etc.

Il existe un certain nombre de codes, *open-source* ou commerciaux, qui ont été lourdement optimisés depuis des années, et qu'il serait dommage de refaire soi-même en moins bien.

BLAS Les *Basic Linear Algebra Subroutines* (BLAS) sont les plus importants. Il s'agit d'une interface implantée par plusieurs bibliothèques pour faire de l'algèbre linéaire dense. La bibliothèque contient trois types de fonctions :

Niveau 1 Opérations sur des vecteurs ou entre deux vecteurs (produit scalaire, $x \leftarrow \alpha x + \beta y$, ...).

Niveau 2 Opérations matrice-vecteur ($y \leftarrow y + Ax$, $y \leftarrow T^{-1}x$ où T est triangulaire, ...).

Niveau 3 Opérations matrice-matrice ($C \leftarrow C + AB$, ...).

L'intensité opérationnelle des niveaux 1 et 2 est plutôt faible, mais comme on l'a vu celle des fonctions du niveau 3 est plus élevée. Dans les implantations des BLAS de bonne qualité, *les fonctions du niveau 3 s'approchent très près des performances de crête de nombreuses machines.* On a donc tout intérêt à les utiliser !

Les BLAS ont initialement été écrits en Fortran 77, donc un langage où les noms de fonctions étaient limités à 6 caractères. C'est ainsi que les fonctions des BLAS ont des noms barbares : `DGEMM` (Double precision GEneral Matrix-Matrix multiply), `STRMV` (Single precision TRiangular Matrix-Vector multiply), `CAXPY` (Complex $A \times x$ Plus y), ...

Il existe des BLAS *open-source* de bonne qualité, comme ATLAS ou OpenBLAS. Il y a aussi des BLAS commerciaux, tels que MKL (Intel), CuBLAS (NVIDIA), etc.

La plupart des opérations intéressantes en algèbre linéaire (résolution de systèmes, etc.) peuvent être implantées de telle manière à exploiter l'efficacité des BLAS. La librairie LAPACK offre toutes les factorisations de matrice possibles, et elle est efficace tant que le BLAS utilisé l'est.

Un *benchmark* standard des grosses machines parallèles, LINPACK, résoud un gros système parallèle de manière distribuée en utilisant MPI et LAPACK. Il atteint souvent une fraction importantes des performances de crête des meilleures machines.

FFT La transformée de Fourier rapide est une opération récurrente d'un certain nombre de calculs, et il existe au moins une librairie qui atteint de bonnes performances sur presque toutes les machines : FFTW (the *Fastest Fourier Transform in the West*).

La FFT nécessite $\mathcal{O}(N \log N)$ opérations arithmétiques sur des données de taille N, donc là-aussi on peut espérer une intensité opérationnelle qui augmente et s'éloigne des petites constantes lorsque N grandit.