Travaux dirigés n°1

Nicolas RINCON VIJA

nicolas.rincon@ensta.fr

École Nationale Supérieure de Techniques Avancées - ENSTA
Systèmes parallèles et distribués - OS02
Palaiseau, France

Année académique 2024-2025

Table des matières

Table des figures

Remarque préalable

Ce travail a été réalisé sur la base des résultats obtenus à partir des exécutions disponibles sur le dépôt GitHub suivant : https://github.com/Nrinconv/OS02_ENSTA.

1 lscpu

La commande *lscpu* affiche des informations sur l'architecture du processeur, y compris le nombre de cœurs, la fréquence et les extensions prises en charge.

1.1 Informations CPU

```
# CPU Information

## General
- Architecture: x86_64
- CPU Modes: 32-bit, 64-bit
- Byte Order: Little Endian
- Physical Address Size: 39 bits
- Virtual Address Size: 48 bits

## Processor
- Total CPU(s): 8
- Online CPU(s): 0-7
- Vendor: GenuineIntel
- Model Name: Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
- CPU Family: 6
- Model: 142
- Stepping: 10
- Threads per Core: 2
- Cores per Socket: 4
- Socket(s): 1
- CPU Scaling MHz: 81%
- CPU Max MHz: 4000.0000
- CPU Min MHz: 400.0000
- BogoMIPS: 3999.93

## Virtualization
- Technology: VT-x

## Cache Memory (sum of all)
- L1d: 128 KiB (4 instances)
- L1i: 128 KiB (4 instances)
- L2: 1 MiB (4 instances)
- L3: 8 MiB (1 instance)

## NUMA
- NUMA Nodes: 1
- NUMA Node0 CPU(s): 0-7
```

Les CPU disponibles (0-7) représentent 8 CPU logiques grâce à l'hyper-threading activé (2 threads par cœur). Le processeur est composé de 4 cœurs physiques et peut ajuster sa fréquence entre 400 MHz et 4,0 GHz. Les caches L1, L2 et L3 sont utilisés pour améliorer le traitement des instructions. Enfin, la virtualisation VT-x permet d'accélérer les machines virtuelles.

n	secondes	MFlops
1023	12.5536	170.564
1024	9.68062	221.833
1025	11.7222	183.735

Table 1 – Performances du produit matrice-matrice naïf

2 Produit matrice-matrice

2.1 Effet de la taille de la matrice

On peut observer dans la *table*1 que les matrices dont la taille n'est pas une puissance de 2 (1023, 1025) affichent un MFlops plus élevé que celle de 1024. Cela s'explique par la gestion du cache du processeur, qui, dans mon cas, dispose de 8 MiB de cache L3 partagé.

Les tailles qui sont des puissances de 2 peuvent provoquer des conflits de cache (cache thrashing), car les accès mémoire sont trop alignés, ce qui augmente les latences. En revanche, des tailles légèrement différentes permettent une meilleure répartition des accès en mémoire, réduisant ainsi ces conflits et améliorant les performances.

Ainsi, l'optimisation du produit matrice-matrice ne dépend pas uniquement de la complexité algorithmique, mais aussi de l'alignement des données en mémoire, qui influence directement l'efficacité du cache.

2.2 Permutation des boucles

Expliquer comment est compilé le code (ligne de make ou de gcc) :

```
make TestProductMatrix.exe && ./TestProductMatrix.exe 1024
```

La commande `make TestProductMatrix.exe && ./TestProductMatrix.exe 1024` compile d'abord `TestProductMatrix.exe` en utilisant un *Makefile*, qui appelle `g++` pour générer l'exécutable en fonction des fichiers sources. Si la compilation réussit, `./TestProductMatrix.exe 1024` exécute le programme avec des matrices de taille 1024×1024. L'opérateur `&&` garantit que l'exécution ne se produit que si la compilation est réussie.

Alors, le tableau suivant montre le temps d'exécution pour différentes configurations de boucles :

	ijk	**ikj**	**jik**
1023	1.87036 s / 1144.8 MFlops	12.5536 s / 170.564 MFlops	4.23199 s / 505.955 MFlops
1024	4.84376 s / 443.35 MFlops	9.68062 s / 221.833 MFlops	4.68583 s / 458.293 MFlops
1025	2.54318 s / 846.884 MFlops	11.7222 s / 183.735 MFlops	3.67171 s / 586.588 MFlops
	jki	**kij**	**kji**
1023	1.9145 s / 1118.41 MFlops	15.8957 s / 134.703 MFlops	1.76156 s / 1215.51 MFlops
1024	1.75817 s / 1221.43 MFlops	12.3304 s / 174.162 MFlops	1.94469 s / 1104.28 MFlops
1025	1.96474 s / 1096.22 MFlops	13.7238 s / 156.937 MFlops	1.42291 s / 1513.64 MFlops

Table 2 – Comparaison des différentes configurations de boucles

L'ordre **kji** est le plus performant, avec le plus grand nombre de MFlops et le temps d'exécution le plus réduit. Cela s'explique par la façon dont les données sont structurées en mémoire : un bon alignement permet d'optimiser l'utilisation du cache et de minimiser les conflits.

Le code suivant illustre l'ordre des boucles de la configuration **kji** :

```
for (int k = iColBlkA; k < std::min(A.nbCols, iColBlkA + szBlock); ++k)
    for (int j = iColBlkB; j < std::min(B.nbCols, iColBlkB + szBlock); ++j)
        for (int i = iRowBlkA; i < std::min(A.nbRows, iRowBlkA + szBlock); ++i)
            C(i, j) += A(i, k) * B(k, j);
```

L'algorithme le plus rapide se concentre sur l'itération "la plus courante" afin de maintenir les données bien organisées en mémoire et de prévenir les conflits de cache. Ainsi, **kji** et **jki** montrent les meilleures performances, tandis que les configurations comme **kij** sont beaucoup plus lentes.

2.3 OMP sur la meilleure boucle

```
make TestProductMatrix.exe && OMP_NUM_THREADS=8 ./TestProductMatrix.exe 1024
```

2.3.1 Résultats obtenus

Les tableaux suivants présentent les performances obtenues en fonction du nombre de threads OpenMP et de la taille des matrices.

OMP_NUM	MFlops (n=1024)	MFlops (n=2048)	MFlops (n=512)	MFlops (n=4096)
1	1829.98	1476.91	1832.56	1475.05
2	2937.84	2350.03	3098.15	2332.50
3	2867.45	2563.36	4247.64	2653.16
4	3650.54	4224.92	4917.78	3207.33
5	2899.81	2971.85	3820.08	2718.97
6	4126.58	3506.64	3541.34	3097.09
7	2652.55	3259.89	3033.41	3565.82
8	3578.44	3537.78	3095.77	3486.01

TABLE 3 – Performances en MFlops en fonction du nombre de threads et de la taille de la matrice

Le tableau suivant montre l'accélération obtenue en fonction du nombre de threads :

OMP_NUM	Speedup (n=1024)	Speedup (n=2048)	Speedup (n=512)	Speedup (n=4096)
2	1.60539	1.59118	1.69061	1.5813
3	1.56693	1.73562	2.31787	1.79869
4	1.99485	2.86065	2.68356	2.17439
5	1.58461	2.01221	2.08456	1.84331
6	2.25499	2.37431	1.93246	2.09965
7	1.4495	2.20724	1.65529	2.41742
8	1.95545	2.39539	1.68931	2.36332

TABLE 4 – Accélération en fonction du nombre de threads

La **Table 4** présente les valeurs de speedup obtenues en fonction du nombre de threads. Ces valeurs sont calculées à l'aide de la formule suivante :

$$\text{Speedup} = \frac{T_{\text{séquentiel}}}{T_{\text{parallèle}}} = \frac{\text{MFlops}_{\text{parallèle}}}{\text{MFlops}_{\text{séquentiel}}} \tag{1}$$

où :

— $T_{\text{séquentiel}}$ est le temps d'exécution avec un seul thread.

— $T_{\text{parallèle}}$ est le temps d'exécution avec n threads.

— $\text{MFlops}_{\text{séquentiel}}$ et $\text{MFlops}_{\text{parallèle}}$ représentent les performances mesurées en millions d'opérations en virgule flottante par seconde.

Cette formule permet d'évaluer l'efficacité de la parallélisation : un speedup idéal devrait être proportionnel au nombre de threads. Cependant, en raison des limitations de la mémoire cache et des conflits d'accès mémoire, l'augmentation du speedup n'est pas toujours linéaire.

2.3.2 Analyse des performances

Les résultats montrent que la performance du produit matrice-matrice s'améliore avec le nombre de threads, mais pas de manière linéaire. Plusieurs facteurs expliquent cette tendance :

— Gain initial avec l'augmentation des threads : Jusqu'à 4 threads, le calcul est bien parallélisé et les performances augmentent efficacement.

— Saturation et baisse d'efficacité : Au-delà de 5 à 8 threads, les gains de performance deviennent minimes et, dans certains cas, on observe même une baisse des performances. Cela s'explique par la concurrence pour l'accès à la mémoire, ce qui entraîne des latences supplémentaires et un temps d'attente accru entre les threads.

— Influence de la taille de la matrice : Pour les petites matrices (512), l'accélération est plus efficace car les données tiennent mieux en cache, limitant les accès à la RAM. En revanche, pour les grandes matrices (4096), le débit mémoire devient un facteur limitant, ce qui ralentit l'exécution malgré l'augmentation du nombre de threads.

Conclusion

— Le nombre optimal de threads dépend de la taille de la matrice.

— Le meilleur équilibre entre parallélisation et gestion mémoire semble être atteint avec 4 à 6 threads et à 7 threads et plus, la saturation de la bande passante mémoire et la surcharge de synchronisation réduisent l'efficacité.

— Pour améliorer davantage les performances, on pourrait explorer :

  — L'optimisation avec des blocs de calcul pour limiter les accès mémoire.

  — L'utilisation d'instructions SIMD pour améliorer le débit des opérations.

  — Des techniques de répartition de charge dynamique pour mieux exploiter les ressources.

Code Extrait du code utilisé :

```
namespace {
void prodSubBlocks(int iRowBlkA, int iColBlkB, int iColBlkA, int szBlock,
                   const Matrix& A, const Matrix& B, Matrix& C) {

  #pragma omp parallel for
  for (int k = iColBlkA; k < std::min(A.nbCols, iColBlkA + szBlock); ++k)
    for (int j = iColBlkB; j < std::min(B.nbCols, iColBlkB + szBlock); ++j)
      for (int i = iRowBlkA; i < std::min(A.nbRows, iRowBlkA + szBlock); ++i)
        C(i, j) += A(i, k) * B(k, j);
}
const int szBlock = 32;
}
```

2.3.3 Analyse des courbes de speedup

L'analyse des courbes révèle plusieurs tendances :

— Croissance initiale (threads 2 à 4) : Toutes les tailles de matrices bénéficient d'un gain significatif, avec un speedup proche de 3 pour $n = 512$ et $n = 2048$.

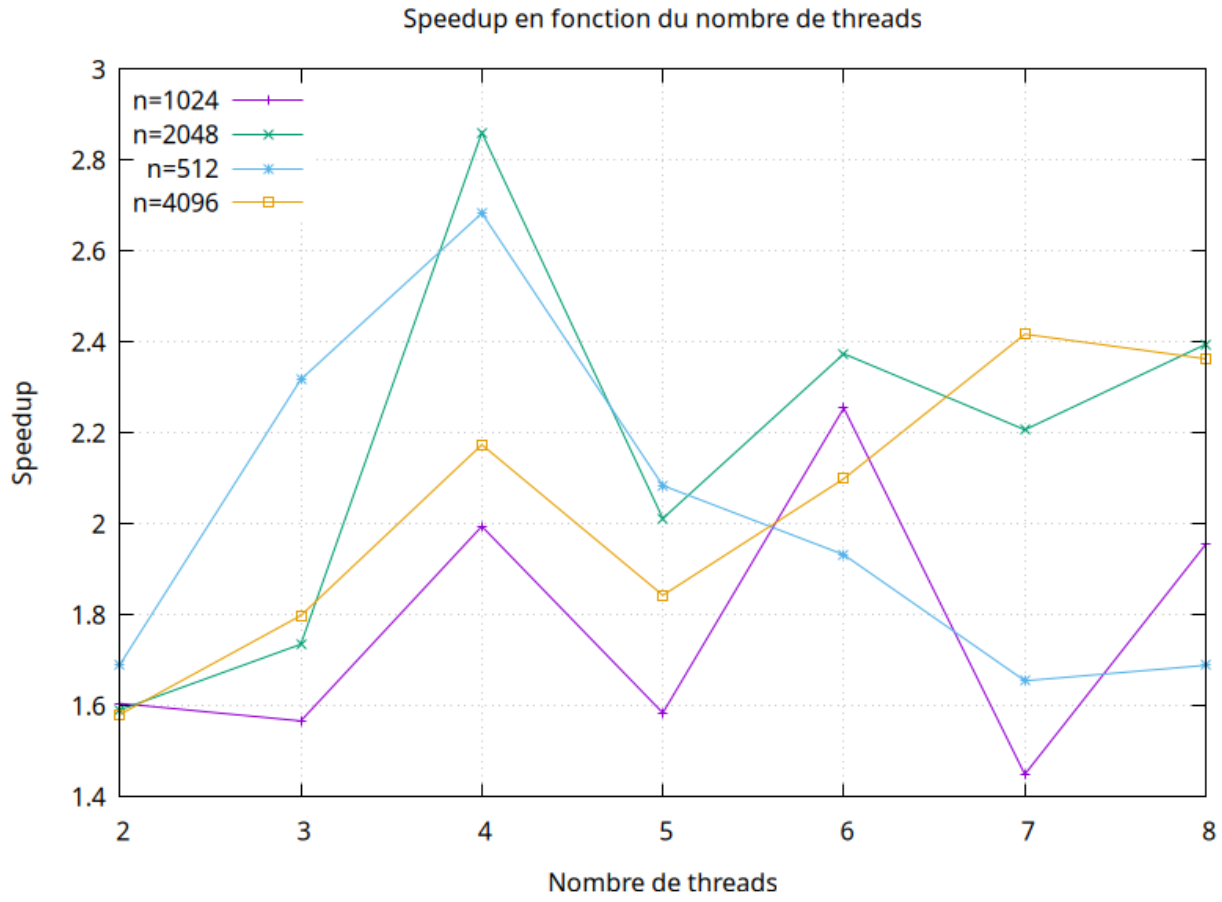

FIGURE 1 – Courbes de speedup en fonction du nombre de threads

— Diminution et instabilité après 4 threads : On observe une baisse marquée du speedup après 4 threads, en particulier pour $n = 512$. Cette instabilité est liée aux conflits d'accès mémoire et à la surcharge de synchronisation.

— Impact de la taille de la matrice : Les grandes matrices ($n = 4096$) maintiennent une progression plus stable, tandis que les petites matrices souffrent davantage du coût de synchronisation des threads.

Conclusion

— L'optimisation est efficace jusqu'à 4-6 threads, mais au-delà, la gestion mémoire et la synchronisation deviennent des facteurs limitants.

— Les grandes matrices ($n = 4096$) montrent une meilleure scalabilité, tandis que les petites matrices ($n = 512$) sont plus sensibles aux fluctuations.

— Des optimisations comme la gestion intelligente du cache, la répartition dynamique des tâches et l'utilisation d'instructions SIMD pourraient améliorer les performances.

2.4 Produit par blocs

```
make TestProductMatrix.exe && ./TestProductMatrix.exe 1024
```

La **Table 5** montre les performances en MFlops en fonction de la taille des blocs.

szBlock	MFlops (n=1024)	MFlops (n=2048)	MFlops (n=512)	MFlops (n=4096)
32	1176.35	1149.51	1599.32	1288.56
64	1493.48	1667.65	1184.68	1557.57
128	1687.88	1914.26	2206.11	1670.23
256	2335.24	1951.86	2417.48	1865.26
512	2110.14	2173.35	2613.79	1620.18
1024	1809.54	1870.1	2592.93	1470.7

TABLE 5 – Performance du produit par blocs

2.4.1 Comparaison avec le produit matrice-matrice scalaire

Afin d'évaluer l'impact de l'optimisation par blocs, nous comparons les performances obtenues avec celles de la multiplication scalaire. Les résultats mettent en évidence plusieurs tendances :

- **Amélioration significative des performances avec le blocage :** Par rapport à la version scalaire, l'utilisation d'un blocage permet une nette augmentation des MFlops, atteignant jusqu'à un facteur de 2 à 3 par rapport à la version naïve.
- **Influence de la taille des blocs :**
  - Pour les petites valeurs de `szBlock` $(32, 64)$, l'amélioration reste modérée.
  - La performance atteint un maximum pour `szBlock` $= 512$, où l'exploitation du cache est la plus efficace.
  - Une baisse des performances est visible pour `szBlock` $= 1024$, ce qui indique que des blocs trop volumineux dépassent la capacité du cache, ce qui mène à un nombre supérieur d'accès à la mémoire RAM.
- **Impact de la taille de la matrice :**
  - Pour les petites matrices ($n = 512$), l'amélioration est particulièrement marquée car le cache est utilisé de manière plus efficace.
  - Pour les grandes matrices ($n = 4096$), bien que l'amélioration soit notable, la bande passante mémoire devient un facteur limitant.

Conclusion L'optimisation par blocs réduit significativement le temps d'exécution en minimisant les accès mémoire inutiles. Cependant, la taille optimale des blocs dépend fortement de la taille de la matrice et de l'architecture matérielle. Une taille de bloc comprise entre 256 et 512 semble offrir le meilleur compromis entre exploitation du cache et réduction des latences mémoire.

Code Extrait du code utilisé in Matrix operator* :

```
const int szBlock = 128; //32,64,512..
...
// Itération sur les blocs
    for (int iRowBlkA = 0; iRowBlkA < A.nbRows; iRowBlkA += szBlock) {
        for (int iColBlkB = 0; iColBlkB < B.nbCols; iColBlkB += szBlock) {
            for (int iColBlkA = 0; iColBlkA < A.nbCols; iColBlkA += szBlock) {
                prodSubBlocks(iRowBlkA, iColBlkB, iColBlkA, szBlock, A, B, C);
            }
        }
    }
    return C;
```

2.5 Bloc + OMP

La **Table 6** compare les performances obtenues en utilisant OpenMP pour paralléliser la multiplication matrice-matrice par blocs.

szBlock	OMP_NUM	MFlops (n=1024)	MFlops (n=2048)	MFlops (n=512)	MFlops (n=4096)
1024	1	1705.14	1707.3	2369.28	1267.83
1024	8	1887.34	1796.62	2496.04	1559.7
512	1	1868.92	2068.58	2850.17	1670.23
512	8	5222.42	1913.14	2398.35	1769.76

TABLE 6 – Bloc + OpenMP

2.5.1 Analyse de la parallélisation du produit matrice-matrice par blocs avec OpenMP

On observe :

— Impact du nombre de threads : L'utilisation de 8 threads améliore significativement les performances, en particulier pour `szBlock = 512` où l'on atteint 5222.42 MFlops pour $n = 1024$, ce qui représente une accélération notable par rapport à la version avec un seul thread.

— Comparaison avec la version scalaire parallélisée : Bien que la parallélisation améliore le temps d'exécution, le produit par blocs est plus efficace que la version scalaire parallélisée. Cela s'explique par une meilleure gestion du cache, limitant les accès à la mémoire RAM et améliorant la localité des données.

— Influence de la taille des blocs :

  — Pour `szBlock = 1024`, le gain de performance est moins important, ce qui confirme que cette taille dépasse la capacité du cache et entraîne une saturation de la bande passante mémoire.

  — Pour `szBlock = 512`, les performances sont optimales avec OpenMP, ce qui montre qu'une taille de bloc adaptée permet une exploitation efficace des ressources du processeur.

Conclusion L'optimisation par blocs combinée à OpenMP permet une accélération significative du produit matrice-matrice. Cependant, la taille du bloc joue un rôle clé. Une valeur trop grande comme 1024 entraîne une saturation mémoire, réduisant l'efficacité du parallélisme. Une taille de bloc de 512 semble être le meilleur compromis entre gestion du cache et accélération parallèle.

Code Extrait du code utilisé in Matrix operator* :

```
const int szBlock = 512; // ou 1024
...
// Itération sur les blocs
    #pragma omp parallel for
    for (int iRowBlkA = 0; iRowBlkA < A.nbRows; iRowBlkA += szBlock) {
        for (int iColBlkB = 0; iColBlkB < B.nbCols; iColBlkB += szBlock) {
            for (int iColBlkA = 0; iColBlkA < A.nbCols; iColBlkA += szBlock) {
                prodSubBlocks(iRowBlkA, iColBlkB, iColBlkA, szBlock, A, B, C);
            }
        }
    }
    return C;
```

2.6 Comparaison des performances entre BLAS et Bloc + OpenMP

La Table 7 présente une comparaison entre l'algorithme optimisé de BLAS et l'implémentation Bloc + OpenMP en termes de temps d'exécution et de performance (MFlops).

2.6.1 Analyse des résultats

Les résultats montrent que BLAS est beaucoup plus rapide que l'implémentation Bloc + OpenMP. Cette grande différence s'explique par plusieurs points :

n	BLAS		Bloc + OpenMP (8 threads)	
	Temps (s)	MFlops	Temps (s)	MFlops
1024	0.0319	67111.7	0.3732	5753.46
2048	0.1813	94739.3	8.0643	2130.43
4096	2.1830	62957.7	75.5130	1820.07

TABLE 7 – Comparaison des performances entre BLAS et Bloc + OpenMP

— **BLAS est mieux optimisé** BLAS est une bibliothèque spécialisée qui utilise des techniques avancées pour accélérer les calculs, notamment en organisant les données de manière plus efficace et en exploitant les capacités du processeur.

— **Bloc + OpenMP est limité par la mémoire** Lorsque la taille de la matrice augmente, OpenMP doit gérer beaucoup plus d'accès à la mémoire, ce qui ralentit l'exécution. BLAS, de son côté, optimise ces accès pour minimiser le temps perdu.

— **Le parallélisme OpenMP n'est pas suffisant** Bien qu'OpenMP permette d'accélérer l'exécution avec plusieurs threads, il ne peut pas compenser l'efficacité des optimisations internes de BLAS.

Conclusion BLAS est largement supérieur à l'implémentation Bloc + OpenMP en raison de ses optimisations avancées. Bien que notre implémentation puisse être améliorée, elle reste limitée par la gestion de la mémoire et l'organisation des calculs. Pour obtenir de meilleures performances, il faudrait utiliser des techniques similaires à celles de BLAS, comme une meilleure gestion des données et des instructions plus adaptées au matériel.