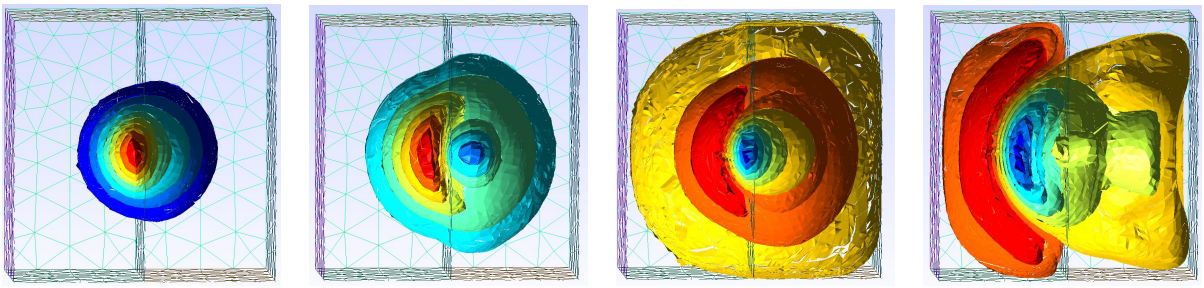
Rapport SIM3
Calcul Haute Performance

Projet : Elément finis discontinus

KERYER Benoît
Yann M. V. EPONGUE DJEUGOUE
Enseignant : MODAVE Axel



(a) t = 0.08 (b) t = 0.16 (c) t = 0.24 (d) t = 0.40

Ecole Nationale Supérieure de Techniques Avancées

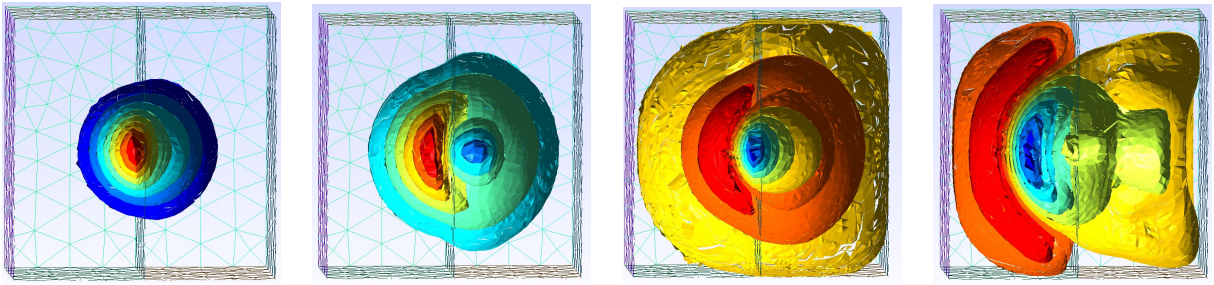
26 mai 2025

Table des matières

Table des figures

1 Introduction

L'objectif de ce projet est d'améliorer les performances d'un code pour du calcul sur un processeur multi-cœur. Le code considéré permet de simuler la propagation d'ondes scalaires en 3D. Il est basé sur une méthode d'éléments finis discontinus de type Galerkin pour des maillages composés de tétrahèdres avec des fonctions de base polynomiales de degré $N \in [1, 7]$. Dans toute la suite du document N sera le degré des fonctions de base polynomiales.



(a) t = 0.08 (b) t = 0.16 (c) t = 0.24 (d) t = 0.40

FIGURE 2 – Propagation d'une onde scalaire dans un cube composé de deux milieux physiques différents avec N = 3. Seule la moitié du cube est montrée.

2 Analyse préliminaire de la performance sur le code non-optimisé

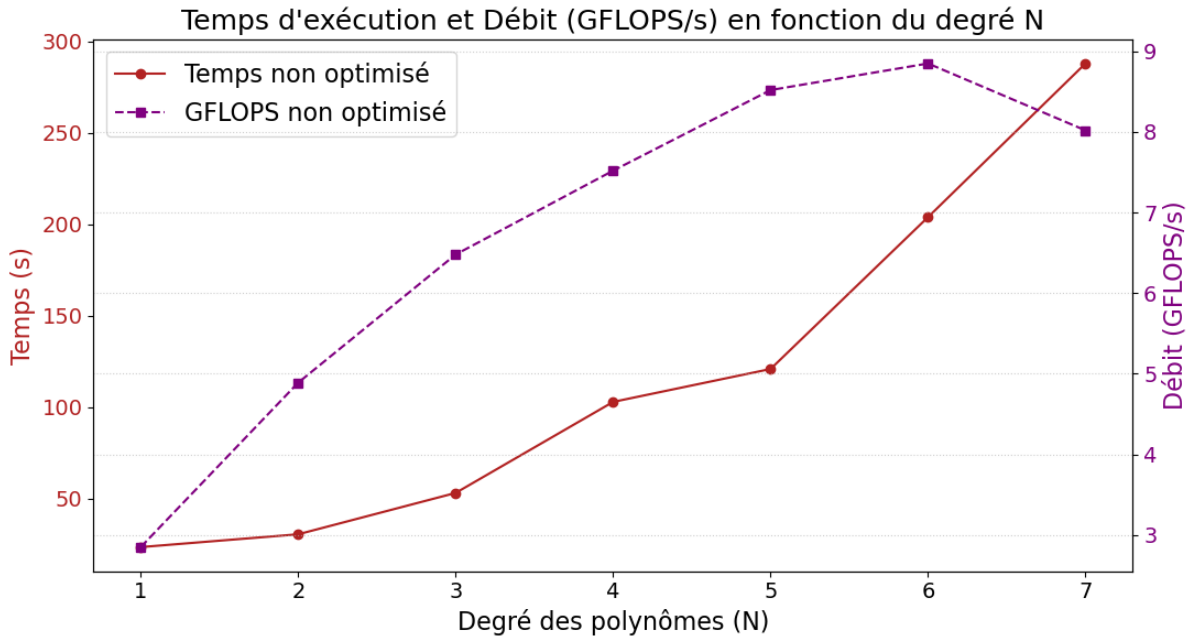


FIGURE 3 – Temps d'exécution et débit (GFLOPS/s) en fonction du degré des polynômes (N).

Les résultats montre que le code non optimisé souffre d'une augmentation rapide des temps d'exécution avec N, due à la complexité croissante des calculs. Le débit arithmétique, bien que meilleur pour $N = 3$ à $N = 6$, reste faible, indiquant une sous-exploitation des ressources matérielles. Les résultats confirment que la phase de run devient coûteuse pour les degrés élevés, avec des temps d'exécution prohibitifs pour $N \geq 4$. Le débit arithmétique, bien qu'amélioré pour les degrés intermédiaires, semble stagné par l'absence d'optimisations. La baisse pour N=7 semble venir de limitations matérielles, comme des problèmes d'accès mémoire. Pour améliorer les performances, des optimisations comme la parallélisation OpenMP, la vectorisation manuelle, ou l'utilisation de routines BLAS (comme `cblas_dgem?`) seront envisagées, surtout pour les grands N.
Ces observations justifient l'exploration d'optimisations pour réduire les temps et augmenter le débit, en particulier pour les simulations à haute résolution.
Le nombre de FLOPs théorique est donné par

$$120 \times \text{Nsteps} \times K \times \left[\frac{(N+1)(N+2)(N+3)}{6}\right]^2.$$

En réécrivant cette expression en fonction de N, avec Nsteps $\propto N^2$ et K constant, nous avons trouvé théoriquement que FLOPs $\propto N^8$, ce qui en pratique n'est pas vérifié. Cette dépendance traduit la complexité des calculs volumiques dominants dans la simulation.

Remarque sur les performances des boucles : Nous pouvons voir que la première boucle `k`, correspondant à la partie **(1) Mise à jour du terme de droite (RHS)**, s'exécute en environ **54 secondes**. En comparaison, la deuxième boucle `k`, correspondant à la partie **(2) Mise à jour du résultat et des champs**, s'exécute en environ **2,0 secondes**, soit environ 27 fois plus rapidement.

Le compteur d'opérations flottantes attribué à la deuxième boucle est d'environ $7,9 \times 10^9$, tandis que celui attribué à la première boucle est d'environ $3,9 \times 10^{11}$, soit environ 49 fois supérieur.

Nous constatons donc que la première boucle est beaucoup plus longue à s'exécuter, tandis que la deuxième réalise un nombre d'opérations flottantes bien plus important, illustrant leurs rôles distincts dans le calcul. Il sera donc plus important d'optimiser la première partie du code que la seconde (nous nous intéresserons toutefois aussi à la seconde partie du code).

3 Optimisation et analyse de performance

Nous avons optimisé la phase d'exécution du programme en modifiant uniquement `main.cpp`. Les boucles coûteuses ont été parallélisées (OpenMP), vectorisées (SIMD) ou réorganisées. Certains calculs redondants ont été pré-calculés. Chaque modification a été testée séparément. Les performances du code optimisé sont ensuite comparées à celles du code initial.

3.1 Optimisations principales

Dans la suite, nous présentons les modifications principales apportées, chacune visant à influencer la performance tout en garantissant la précision des résultats.

3.1.1 Méthodologie

Nous avons séparé le code ainsi :

- (1) Mise à jour du terme de droite (RHS)
  - (1.1) Mise à jour du vecteur de pénalité
  - (1.2) Calcul des termes volumiques
  - (1.3) Calcul des termes de surface
- (2) Mise à jour du résultat et des champs

Pour chaque partie et sous-partie, nous avons examiné les optimisations possibles et testé leur impact. Cependant, certaines étapes ont été traitées globalement, notamment le pré-calcul, car certaines variables précalculées sont partagées entre plusieurs sections.

Pour vérifier que les modifications ne compromettaient pas la validité des résultats, nous avons comparé la solution obtenue uniquement à l'état final (au temps final), par souci de simplicité. Plus précisément, les vérifications ont été réalisées dans l'ordre suivant :

- La valeur maximale de la solution finale reste inchangée.
- Exécution d'un script Python pour comparer les valeurs avec une tolérance définie.
- Vérification visuelle des résultats à l'aide de Gmsh.

Nous avons vérifié ainsi à chaque comparaison de temps d'exécution entre deux versions du code.

3.1.2 Tableau récapitulatif des modifications testées

Le tableau 1 synthétise les optimisations appliquées, notamment en parallélisation et vectorisation, ainsi que leur impact sur les performances de calcul.

Remarque : Les temps de référence (également appelés temps initiaux dans le tableau) diffèrent selon les parties, car certaines optimisations précédentes ont été conservées et cumulées au fil des modifications.

3.1.3 Explication des différentes optimisations

- **Réduction des calculs redondants**
  - **Où** : Phase Run.
  - **Comment** : Les termes comme $1/(\rho \cdot c)$ étaient recalculés à l'intérieur de certaines boucles dont ils étaient indépendants. Nous les avons factorisés et sortis de la boucle en évaluant par exemple `double inv_rho_c = 1. / (rho * c);` puis en réutilisant `inv_rho_c`.
  - **Pourquoi** : Cela évite de répéter des divisions coûteuses pour chaque itération, améliorant ainsi les performances.
  - **Résultat :** Il n'est pas représenté dans le tableau mais cela fait un gain inférieur à un pourcent. Ceci s'explique d'une part par le fait que nous avons précalculé des opérations entières qui sont relativement moins coûteuses que des opérations flottantes, et d'autre part car le temps de calcul est dominé par d'autres opérations.

- **Réorganisation de certaines boucles**

Nom de l'optimisation	Gain de temps de calcul en %
Partie 1 : Mise à jour du terme de droite (RHS)	53,15 secondes
Parallélisation sur la première boucle k dans la partie 1 du code : #pragma omp parallel for for (int k = 0; k < K; ++k)	65,19%
Partie 1.2 – Calcul des termes volumiques	Temps initial (secondes) : 17,1873
Inversion des boucles n et m	-22,76%
Partie 1.3 – Calcul des termes de surface	Temps initial (secondes) : 17,14
Inversion des boucles n et f (implémentation délicate)	-2,33%
Utilisation d'accumulateurs pour regrouper les contributions face par face, puis mise à jour finale hors de la boucle interne	-0,15%
sans vectorisation, c'est-à-dire sans #pragma omp simd reduction(+ : p_lift, u_lift, v_lift, w_lift, op_counter)	4,05%
Partie 2 – Mise à jour du RHS (avec export de la solution finale) boucle k seconde : for (int k = 0; k < K; ++k) avec boucle iField déroulé	Temps initial (secondes) : 17,8183
Vectorisation de la boucle interne n : #pragma omp simd for (int n = 0; n < Np; ++n)	3,47%
Parallélisation sur la boucle k : #pragma omp parallel for for (int k = 0; k < K; ++k) for (int k = 0; k < K; ++k) {	6,99%
Partie 2 - Boucle sur les champs : for (int iField = 0; iField < Nfields; ++iField)	Temps initial (secondes) : 19,1331
#pragma vector always	-0,92%
#pragma novector	2,21%
Test de déroulement manuel de la boucle	6,72%

TABLE 1 – Tableau résumé des optimisations et gains de performance pour N=3

- **Où** : (1.2) : *Calcul des termes volumiques* et Dans (1.3) : *Calcul des termes de surface.*
- **Comment** : Nous avons inversé l'ordre des boucles sur n et f dans le code original pour permettre une vectorisation sur n avec une dépendance constante à F_{scale} (= _Fscale[k * NFacesTet + f]) par face. Cela rend la boucle interne plus adaptée à SIMD.
- **Pourquoi** : *Partie 1.2 :* Nous pourrions envisager d'inverser l'ordre des boucles (`n` à l'extérieur et `m` à l'intérieur) afin d'améliorer les performances du calcul. En traitant l'indice `m` en premier, nous accédons aux tableaux `_Dr`, `_Ds` et `_Dt` de manière plus contiguë en mémoire, ce qui permettrait une meilleure exploitation du cache. Cette réorganisation favoriserait également la vectorisation automatique par le compilateur, ce qui rendrait l'exécution plus efficace.
  Partie 1.3 : Inverser l'ordre des boucles `n` et `f` permettrait d'améliorer la localité mémoire et l'efficacité du cache. En traitant d'abord la boucle sur `f`, on accède de manière plus contiguë aux éléments du tableau `_LIFT` indexés par `n * NFacesTet * Nfp + m`, ce qui optimise les accès mémoire pour les calculs des contributions sur chaque face. De plus, cette inversion peut permettre une meilleure vectorisation et parallélisation, car les données traitées successivement sont mieux regroupées en mémoire.
- **Résultat** :
  Partie 1.2 : Nous observons un ralentissement de l'ordre de 22 % du temps de calcul. Ce ralentissement s'explique probablement par une perte de localité mémoire lors des lectures des variables indexées par `m` (par exemple `dpdr += Dr * s_p[m];`), dont l'accès devient moins contigu après l'inversion des boucles, ce qui dégrade l'efficacité du cache. En plaçant la boucle sur `n` à l'intérieur, nous réduisons également les opportunités de vectorisation ou de parallélisme sur ces variables intermédiaires. De plus, le code se complique davantage pour conserver la bonne solution.
  Partie 1.3 : Nous observons un ralentissement de 2,33 % suite à l'inversion des boucles, ce qui reste relativement faible et peut être considéré comme négligeable. Ce léger impact peut s'expliquer par le fait que la variable $F_{\text{scale}} = \_Fscale[k * NFacesTet + f]$ n'est plus accédée de manière contiguë, ce qui ralentit l'exécution du problème.

— **Parallélisation des boucles sur les éléments k**

- **Où** : Dans les sections (1) Mise à jour du terme de droite (RHS) et (2) Mise à jour du résultat et des champs.
- **Comment** : Nous avons ajouté la directive `#pragma omp parallel for` avant les boucles sur k (de 0 à $K-1$). Chaque itération traite un élément indépendant, ce qui permet de distribuer le travail sur plusieurs threads en utilisant OpenMP. Cela exploite les capacités des processeurs multicœurs. De plus, nous avons inclus une clause `reduction(+:op_counter)` afin de garantir une accumulation correcte de l'opération `op_counter` entre les différents threads. Cette clause permet d'éviter les conflits d'accès concurrents à cette variable lors de l'exécution parallèle, tout en préservant son rôle de compteur global d'opérations.
- **Pourquoi** : Chaque itération traite un élément indépendant, ce qui permet de

distribuer le travail sur plusieurs threads en utilisant OpenMP. Cela exploite les capacités des processeurs multicœurs. Les éléments k sont indépendants les uns des autres (pas de dépendance de données entre itérations), ce qui en fait un candidat idéal pour la parallélisation.

- **Résultat** :
  - Une parallélisation appliquée uniquement à la première boucle sur k dans la Partie 1 du code permet un gain de temps significatif de **65,19 %**,car cette section est majoritairement responsable du coût de calcul total, et que sa parallélisation est très bénéfique.
  - En revanche, ajouter une seconde directive de parallélisation sur la deuxième boucle sur k (dans la partie 2 du code) ne procure qu'un **gain marginal de 6,99 %** supplémentaire. Cela peut s'expliquer par une moindre charge de travail dans cette seconde boucle.

- **Optimisation de la boucle sur les champs (`iField`)**
  - **Où :** Partie 2 – Mise à jour des résultats et des champs

```
for (int iField = 0; iField < Nfields; ++iField) {
    int id = k * Np * Nfields + n * Nfields + iField;
    _resQ[id] = a * _resQ[id] + dt * _rhsQ[id];
    _valQ[id] = _valQ[id] + b * _resQ[id];
    op_counter += 5;
}
```

  - **Comment :** Plusieurs optimisations ont été testées sur cette boucle :
    - Ajout de la directive `#pragma vector always` pour forcer la vectorisation.
    - Ajout de la directive `#pragma novector` pour désactiver explicitement la vectorisation.
    - Déroulement manuel (unrolling) de la boucle.
  - **Pourquoi :** Les itérations sur les éléments k sont totalement indépendantes (sans dépendance de données entre elles), ce qui en fait un cas idéal pour une parallélisation avec OpenMP. Cela permet de répartir efficacement le travail sur plusieurs threads et d'exploiter pleinement les capacités des processeurs multicœurs.
  - **Résultat :**

Optimisation	Variation du temps
`#pragma vector always`	**−0,92 %**
`#pragma novector`	**+2,21 %**
Déroulement manuel (unrolling)	**+6,72 %**

La boucle ne comporte que `Nfields = 4` itérations, ce qui est trop court pour que la vectorisation automatique soit pleinement exploitée. Le léger ralentissement observé avec `#pragma vector always` peut s'expliquer par une tentative de vectorisation non optimale sur un si petit nombre d'itérations. À l'inverse, désactiver la vectorisation avec `novector` permet ici un léger gain, probablement parce que cela évite une surcharge inutile liée à la tentative de

vectorisation. Enfin, le déroulement manuel de la boucle apporte le gain le plus significatif : en supprimant le coût de contrôle de boucle et en rendant les opérations explicites, il permet une meilleure optimisation par le compilateur, même sans vectorisation.

— **Réorganisation du calcul des termes de surface**

— **Où** : Partie 1.3 — Calcul des termes de surface, dans la boucle de calcul des contributions de surface (`p_lift`, `u_lift`, etc.).

— **Comment** : Deux variantes ont été testées :

— En utilisant des accumulateurs pour chaque face, et en mettant à jour `_rhsQ` en dehors de la boucle interne sur `f`.

— En enlevant la vectorisation, c'est-à-dire la directive `#pragma omp simd reduction(...)` sur la boucle la plus interne de cette partie (for (int m = f * Nfp ; m < (f + 1) * Nfp ; m++)).

— **Pourquoi** : L'objectif est de :

— Réduire les écritures répétées dans `_rhsQ` à chaque itération de `m`, en les regroupant à la fin de la boucle sur `f`.

— Tester l'impact de la vectorisation forcée sur les accumulateurs, car dans certains cas, elle peut ne pas améliorer les performances et même les dégrader si la boucle est trop courte ou les dépendances trop fortes.

— **Résultat et explication du résultat** :

— Avec accumulateurs : légère perte de performance de l'ordre de **-0,15 %**. Cela signifie que la gestion des accumulateurs ne suffit pas à compenser le surcoût de calcul ou d'organisation mémoire.

— Sans vectorisation (pas de `#pragma omp simd`) : **gain de 4,05 %**. Cela peut paraître contre-intuitif, mais s'explique probablement par une surcharge introduite par la directive `simd` dans un contexte où le compilateur ne peut pas vectoriser efficacement (boucles trop courtes, dépendances implicites, etc.). Supprimer la directive laisse plus de liberté au compilateur pour optimiser autrement.

— **Vectorisation de la mise à jour des champs**

— **Où** : Partie 2 — Mise à jour du RHS, cad dans la deuxième boucle sur `k`, qui effectue les opérations finales sur `_resQ` et `_valQ`.

— **Comment** : La directive `#pragma omp simd` a été ajoutée avant la boucle sur `n` pour activer explicitement la vectorisation de cette boucle interne. Celle-ci effectue les calculs indépendants champ par champ, avec un déroulement explicite de la boucle sur `iField` (équivalent à `Nfields = 4`).

— **Pourquoi** : La vectorisation SIMD permet de traiter plusieurs valeurs en parallèle au sein d'un même cœur CPU. Dans cette boucle, les calculs sur les indices `n` sont indépendants et répétés de manière similaire sur plusieurs champs, ce qui en fait un bon candidat à la vectorisation.

— **Résultat et explication du résultat** :

— Gain de performance de **3,47 %** grâce à la directive `#pragma omp simd`. Ce gain, bien que modéré, reste significatif. Il peut s'expliquer par le fait que la Partie 2 de la phase *Run* représente une portion relativement peu coûteuse du temps de calcul global, limitant ainsi l'impact absolu de l'optimisation.

3.2 Optmisation détaillée de la vectorisation de la boucle for (int m = f * Nfp ; m < (f + 1) * Nfp ; m++)

Nous allons nous focaliser sur cette boucle. Pour cela, comme toujours, nous avons vérifié à chaque étape que la solution finale restait la même, soit visuellement via GSMH, soit en utilisant un script Python pour comparer les écarts des valeurs des différents champs. La figure 4 montre différents résultats obtenus avec diverses instructions de vectorisation appliquées à la boucle étudiée.

Version	Instruction pragma	% gain temps	% gain débit
OMP SIMD + reduction + simdlen(2)	omp simd reduction(+:p_lift, u_lift, v_lift, w_lift, op_counter) simdlen(2)	-3.07%	-3.05%
OMP SIMD + reduction + simdlen(4)	omp simd reduction(+:p_lift, u_lift, v_lift, w_lift, op_counter) simdlen(4)	-6.30%	-5.92%
OMP SIMD + reduction	omp simd reduction(+:p_lift, u_lift, v_lift, w_lift, op_counter)	-0.22%	-0.36%
OMP SIMD	omp simd	-1.84%	-1.56%
OMP SIMD forcée	vector always	0.17%	0.36%
OMP SIMD interdite	novector	-19.70%	-16.52%
Vectorisation automatique (aucune consigne)		17,82 secondes	5,57 Gflops

FIGURE 4 – Comparaison des performances pour différentes instructions de vectorisation appliquées à la boucle étudiée pour N=3. La convention utilisée ici, est qu'un pourcentage positif de gain de temps est gain de temps, alors qu'un pourcentage négatif de gain de temps est une perte de temps.

Remarque : Le temps indiqué correspond à la phase *run* du programme. Nous aurions pu prendre le temps seulement de l'exécution de la boucle, mais dans ce cas-là, nous n'aurions pas vu l'impact de la boucle sur le code complet. Par exemple, si nous nous apercevions d'un gain de 20% dans la boucle mais que cela ne représente que 0.1% de temps gagné dans la phase run alors l'amélioration ne vaudrait pas le coût. Aussi cela permet de prendre en compte des effets indirects.

Pour les différences de performance inférieures à 2 %, on peut raisonnablement attribuer ces variations à la variabilité des mesures. En effet, l'écart-type observé sur les gains de temps est d'environ 2 %, ce qui signifie que des variations plus petites que cette valeur peuvent simplement provenir des fluctuations normales liées à la précision de la mesure, plutôt qu'à une réelle amélioration ou dégradation du code. Voici nos principales explications de nos résultats :

— La vectorisation automatique, sans directive explicite, donne de bonnes performances puisque le compilateur optimise la boucle seul pour utiliser les instructions SIMD.

— Interdire la vectorisation (avec l'option `novector`) ralentit fortement le code (environ -19,7%) car la boucle ne profite plus de la vectorisation offerte par les instructions SIMD.

— L'utilisation de `#pragma omp simd` seule, sans clause `reduction`, entraîne une légère perte de performance (environ -1,8%). Cependant, cette différence étant inférieure à 2 %, elle peut être attribuée à la variabilité naturelle des mesures. Ce léger recul est probablement dû au fait que les accumulations (`+=`) ne sont pas optimisées pour la vectorisation dans ce cas.

— L'ajout de la clause `reduction` permet au compilateur de gérer correctement les accumulations lors de la vectorisation, réduisant quasiment à zéro la perte de performance (-0,22 %). Cette variation est également dans la marge d'erreur de mesure. Nous pourrions en conclure que c'est ce que fait le compilateur sans qu'on lui donne de consigne.

— Forcer la longueur des vecteurs SIMD avec `simdlen(2)` ou `simdlen(4)` limite la taille des paquets SIMD. Si cette taille n'est pas optimale (4 au lieu de 2) pour le processeur, la performance diminue (de -3 % à -6 %), ce qui dépasse la variabilité de mesure et indique un impact réel. Nous allons chercher à optimiser la valeur de simdlen dans la section suivante.

— Forcer systématiquement la vectorisation avec `vector always` apporte un léger gain (+0,17 %), mais cette amélioration est inférieure à l'écart-type observé (2 %), il est donc raisonnable de considérer ce gain comme négligeable. Ce léger gain pourrait s'expliquer par le fait que le compilateur a fait moins de vérifications avant de vectoriser.

3.2.1 Simdlen optimale

Sur une boucle (voir Listing 1) nous avons cherché à comprendre quel serait le simdlen optimal : ce paramètre peut être vu comme fixant le nombre d'itérations qu'une boucle doit regrouper en une seule instruction SIMD. En théorie, le "simdlen" optimal dépend des capacités de notre processeur. La machine de l'ENSTA a un jeu d'instructions AVX2, peut traiter 4 doubles à la fois (car il a des registres de 256 bits). Nous pensions donc que choisir simdlen=4 serait optimal, car cela correspondrait exactement à ce que le processeur peut gérer en une seule instruction, maximisant ainsi la vitesse.

```
1
2  #pragma omp simd reduction(+:p_lift,u_lift,v_lift,w_lift,op_counter)
       simdlen(SIMDLEN)
3  for (int m = f * Nfp; m < (f + 1) * Nfp; ++m) {
4      double tmp = _LIFT[n * NFacesTet * Nfp + m];
5      p_lift += tmp * s_p_flux[m];
6      u_lift += tmp * s_u_flux[m];
7      v_lift += tmp * s_v_flux[m];
8      w_lift += tmp * s_w_flux[m];
9      op_counter += 8;
10 }
```

Listing 1 – Boucle de calcul des flux (code X)

Mais en pratique, les résultats que nous avons obtenus ne montrent pas que 4 est la valeur optimale, comme le montre le graphe 5 [1].

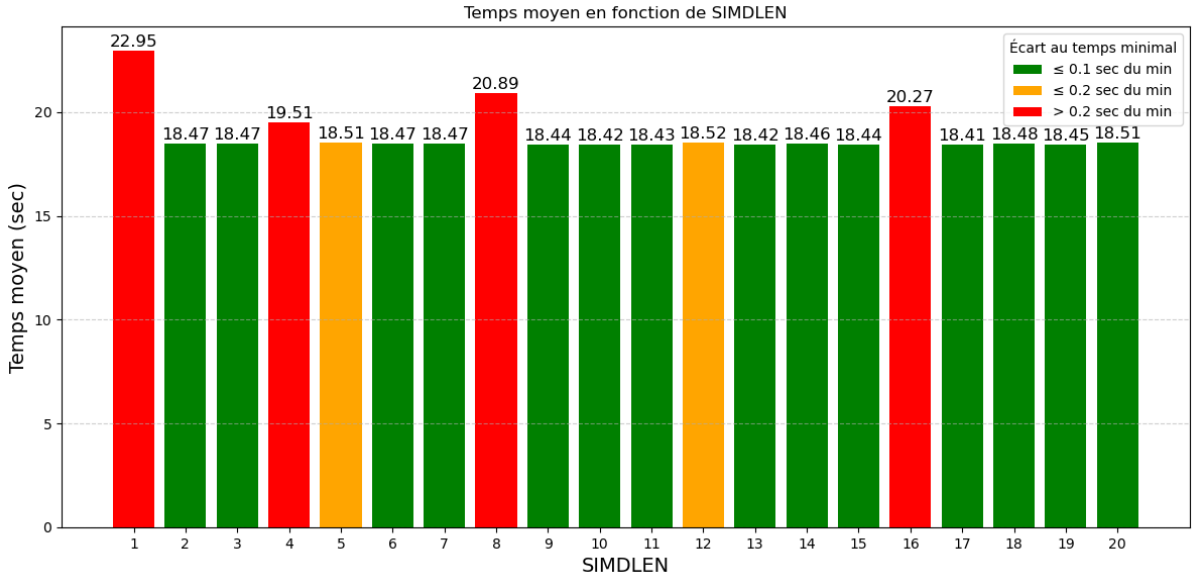


FIGURE 5 – Résultats expérimentaux des temps moyens en fonction de Simdlen

Nous observons tout d'abord que pour Simdlen=1 on a le temps le plus long, cela s'explique par le fait que pour Simdlen=1 nous sommes dans le cas sans vectorisation. D'ailleurs, pour cette valeur, le rapport de compilation nous indique qu'il n'a pas vectorisé la boucle.

Ensuite, on remarque que pour les puissances de 2, nous avons des temps plus longs d'environ 2 secondes que pour les autres valeurs de Simdlen. Plusieurs explications à cela :

— **Mauvaise longueur du vecteur** : Dans le rapport de compilation, on peut lire que, pour les valeurs qui ne sont pas des puissances de 2, la taille du vecteur n'est pas acceptable et est finalement prise comme égale à 2.

— **"Rigidité vectorielle"** : D'après les rapports de compilation, lorsque `simdlen` vaut 4, 8 ou 16, le compilateur reste bloqué à traiter systématiquement les mêmes paquets de 4 (ou leurs multiples) doubles, sans pouvoir commencer à charger les données plus tôt pour éviter l'attente liée à la mémoire. Nous pensons que cela est dû au fait que le compilateur doit conserver de nombreuses sommes partielles avant de les additionner, ce qui alourdit le travail interne. À cela s'ajoutent des lectures non alignées (plus lentes), répétées à chaque paquet, ainsi que des accès toujours concentrés dans les mêmes zones du cache, provoquant des ralentissements. En somme, cette rigidité et ces pénalités liées à la mémoire surpassent les bénéfices de la vectorisation, d'où un ralentissement global.

4 Implémentation alternative avec BLAS

Le but de cette partie est d'optimiser le code initiale mais cette fois-ci en implémentant plutôt les routines BLAS de la librairie MKL. cette implémentation sera amélioré en

1. Le temps moyen a été calculé sur 3 itérations, avec un écart-type faible (moins de 0,2 s en général).Le degré du polynome est de 3

utilisant la vectorisation et la parallélisation avec OpenMP.

4.1 Méthodologie

4.1.1 Etape 1 : détection des parties du code vectorisable avec les routines BLAS

Les bouts code où nous avons observé les opérations matrices vecteurs se trouve dans la partie `2) RUN` du code.

— Nous avons premièrement le bout de code qui calcul les dérivées :

```
// Compute mat-vec product for surface term
for (int n = 0; n < Np; ++n) {
    double dpdr = 0, dpds = 0, dpdt = 0;
    double dudr = 0, duds = 0, dudt = 0;
    double dvdr = 0, dvds = 0, dvdt = 0;
    double dwdr = 0, dwds = 0, dwdt = 0;
    for (int m = 0; m < Np; ++m) {
        double Dr = _Dr[n + m * Np];
        dpdr += Dr * s_p[m];
        dudr += Dr * s_u[m];
        dvdr += Dr * s_v[m];
        dwdr += Dr * s_w[m];
        double Ds = _Ds[n + m * Np];
        dpds += Ds * s_p[m];
        duds += Ds * s_u[m];
        dvds += Ds * s_v[m];
        dwds += Ds * s_w[m];
        double Dt = _Dt[n + m * Np];
        dpdt += Dt * s_p[m];
        dudt += Dt * s_u[m];
        dvdt += Dt * s_v[m];
        dwdt += Dt * s_w[m];
    }
```

A partir de ce code, nous remarquons clairement que (par exemple) nous pouvons définir le tableau `dpdr` de taille $Np*1$ et l'obtenir grâce au produit matrice-vecteur : `dpdr` = `_Dr` $*$ `s_p`.

Nous faisons un constat similaire si on définit les tableaux `dpds`, `dpdt`, `dudr`, `duds`, `dudt`, `dvdr`, `dvds`, `dvdt`, `dwdr`, `dwds` et `dwdt` de taille $Np * 1$

— Ensuite nous avons le code où sont calculés les termes de surface avec la matrice `_LIFT`

```
 for (int n = 0; n < Np; ++n) {
    for (int f = 0; f < NFacesTet; f++) {

// Compute mat-vec product for surface term
        double p_lift = 0.;
        double u_lift = 0.;
        double v_lift = 0.;
        double w_lift = 0.;
```

```
            for (int m = f * Nfp; m < (f + 1) * Nfp; m++) {
              double tmp = _LIFT[n * NFacesTet * Nfp + m];
              p_lift += tmp * s_p_flux[m];
              u_lift += tmp * s_u_flux[m];
              v_lift += tmp * s_v_flux[m];
              w_lift += tmp * s_w_flux[m];
            }
// Load geometric factor
            ...
    }
}
```

A partir de ce code, nous remarquons clairement que (par exemple) nous pouvons définir le tableau `p_lift` de taille $Np * 1$ et l'obtenir grâce au produit matrice-vecteur : `p_lift` = `_LIFT` $*$ `s_p_flux`.

Nous faisons le même constat en définissant les tableaux `u_lift` `v_lift` `w_lift` de taille $Np * 1$

4.1.2 Etape 2 : Implémentation de cblas_dgemv

D'après ce qui précède, nous constatons que nous avons à faire à des opérations matrice-vecteur et pour de tels opérations, c'est la routine `cblas_dgemv()` qui est adéquat.

— **Calcul des dérivées**

```
/* Tableaux temporaires pour les derivees */
vector<double> dpdr(Np, 0.0), dpds(Np, 0.0), dpdt(Np, 0.0);
vector<double> dudr(Np, 0.0), duds(Np, 0.0), dudt(Np, 0.0);
vector<double> dvdr(Np, 0.0), dvds(Np, 0.0), dvdt(Np, 0.0);
vector<double> dwdr(Np, 0.0), dwds(Np, 0.0), dwdt(Np, 0.0);

/* Parametres pour cblas_dgemv */
const double alpha = 1.0, beta = 0.0;
const int incx = 1, incy = 1;

/* Calcul des derivees pour p */
cblas_dgemv(CblasRowMajor, CblasTrans, Np, Np, alpha, _Dr.data(), Np,
    s_p.data(), incx, beta, dpdr.data(), incy);
cblas_dgemv(CblasRowMajor, CblasTrans, Np, Np, alpha, _Ds.data(), Np,
    s_p.data(), incx, beta, dpds.data(), incy);
cblas_dgemv(CblasRowMajor, CblasTrans, Np, Np, alpha, _Dt.data(), Np,
    s_p.data(), incx, beta, dpdt.data(), incy);

/* Calcul des derivees pour u */
cblas_dgemv(CblasRowMajor, CblasTrans, Np, Np, alpha, _Dr.data(), Np,
    s_u.data(), incx, beta, dudr.data(), incy);
cblas_dgemv(CblasRowMajor, CblasTrans, Np, Np, alpha, _Ds.data(), Np,
    s_u.data(), incx, beta, duds.data(), incy);
cblas_dgemv(CblasRowMajor, CblasTrans, Np, Np, alpha, _Dt.data(), Np,
    s_u.data(), incx, beta, dudt.data(), incy);

/*calcul similaire des derivees pour v et W*/
```

```
22
23          ...
```

Listing 2 – Calcul des dérivées via des produits matrice-vecteur avec `cblas_dgemv`

— **Calcul termes de surface avec la matrice `_LIFT`**

```
1
2  // Tableaux temporaires pour les termes lift
3  vector<double> p_lift(Np, 0.0);
4  vector<double> u_lift(Np, 0.0);
5  vector<double> v_lift(Np, 0.0);
6  vector<double> w_lift(Np, 0.0);
7
8  // Calculer les termes lift en utilisant `cblas_dgemv`
9  cblas_dgemv(CblasRowMajor, CblasNoTrans, Np, Nfp * NFacesTet, alpha,
       _LIFT.data(), Nfp * NFacesTet, s_p_flux.data(), incx, beta,
       p_lift.data(), incy);
10 cblas_dgemv(CblasRowMajor, CblasNoTrans, Np, Nfp * NFacesTet, alpha,
       _LIFT.data(), Nfp * NFacesTet, s_u_flux.data(), incx, beta,
       u_lift.data(), incy);
11 cblas_dgemv(CblasRowMajor, CblasNoTrans, Np, Nfp * NFacesTet, alpha,
       _LIFT.data(), Nfp * NFacesTet, s_v_flux.data(), incx, beta,
       v_lift.data(), incy);
12 cblas_dgemv(CblasRowMajor, CblasNoTrans, Np, Nfp * NFacesTet, alpha,
       _LIFT.data(), Nfp * NFacesTet, s_w_flux.data(), incx, beta,
       w_lift.data(), incy);
```

Listing 3 – Application de cblas_dgemv au calcul des termes de surface avec la matrice

4.1.3 Etape : Optimisations des boucles - parallélisation et vectorisation

Explications des optimisations

Voici comment nous avons optimisé les boucles de la partie 2) RUN du code , étape par étape :

1. **Conservation de certaine optimisation faites dans la partie 3.**
   Nous avons conservé les optimisations décrit plus haut telles que : Réduction des calculs redondants et parallélisation des boucles sur les éléments k
2. **Optimisation des accès mémoire**
   - **Où** : Dans tout le code, notamment dans les boucles sur n et nf.
   - **Comment** : Les tableaux comme _valQ, _rhsQ, et les tableaux temporaires (s_p, dpdr, etc.) sont déjà accédés de manière contiguë dans le code d'origine, ce qui est optimal pour la localité des données. Nous avons conservé cette structure.
   - **Pourquoi** : Les accès mémoire contigus réduisent les "cache misses" et améliorent l'efficacité des caches du processeur.

4.2 Présentations des résultats

Les temps de calculs obtenus ci-dessous on été obtenus en ignorant les phases d'initialisation et l'enregistrement des solutions (OutputStep=0).

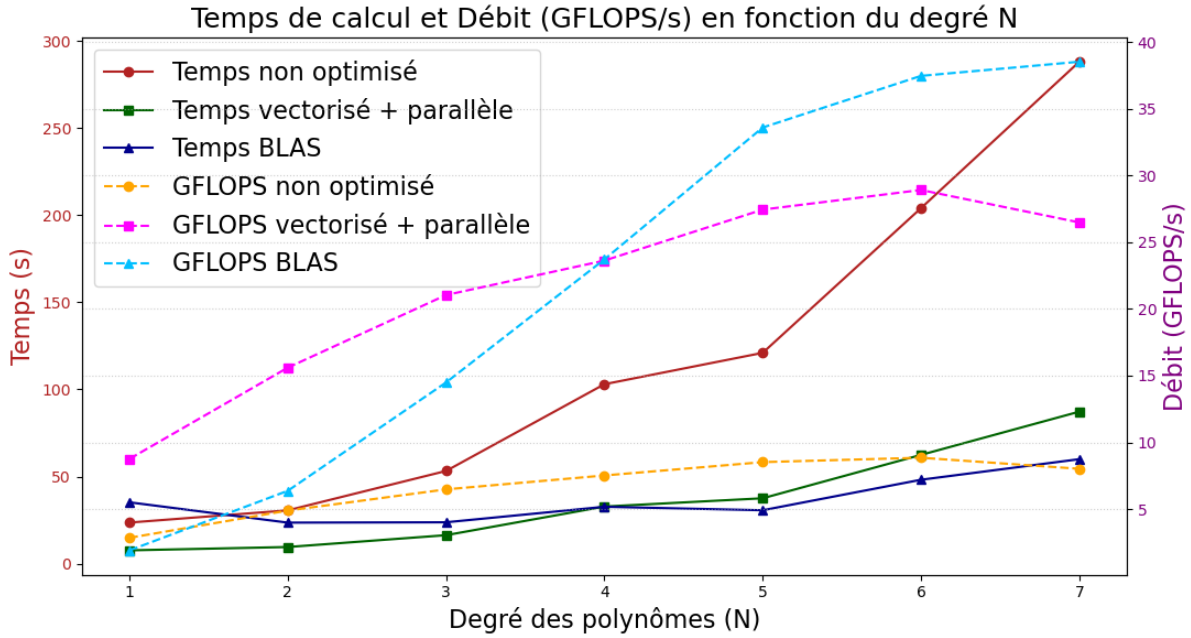


FIGURE 6 – Débit arithmétique et temps d'exécutionen fonction du degré N

4.3 Analyse et interprétation des performances

Les tableaux 4 et 5 (cf Annexes ou bien la figure 6)présentent les performances en termes de temps d'exécution (en secondes) et de débit arithmétique (en GFLOPS/s) pour trois versions d'un code de simulation basé sur une méthode spectrale d'éléments discontinus, en fonction du degré des polynômes N (de 1 à 7). Les trois versions étudiées sont : (1) un code non optimisé, (2) un code optimisé par vectorisation manuelle et parallélisation avec OpenMP, et (3) un code optimisé utilisant la bibliothèque BLAS (notamment la routine `cblas_dgemm` pour les multiplications matricielles).

Temps d'exécution (Tableau 4 cf Annexes)

Le code non optimisé montre des temps d'exécution élevés, croissant rapidement avec N, de 23,64 s pour $N = 1$ à 288 s pour $N = 7$. Cette augmentation est due à la complexité croissante des calculs. La version optimisée par vectorisation manuelle et parallélisation réduit significativement les temps d'exécution, avec des gains d'environ 3x à 3,3x (par exemple, 7,7 s vs 23,64 s pour $N = 1$, et 87,26 s vs 288 s pour $N = 7$). Cette amélioration est attribuable à l'utilisation efficace des processeurs multi-cœurs via OpenMP et à l'optimisation des boucles pour exploiter les instructions SIMD.

La version optimisée avec BLAS montre des résultats contrastés. Pour $N = 1$, elle est moins performante (35,25 s) que le code non optimisé, en raison des surcoûts d'initialisation des routines BLAS pour de petites matrices. Cependant, à partir de $N = 2$, les temps diminuent et deviennent comparables à ceux de la version vectorisée pour $N = 4$ (32,7 s

vs 32,82 s). Pour $N \geq 5$, BLAS surpasse légèrement ou nettement la version vectorisée (par exemple, 60 s vs 87,26 s pour $N = 7$), grâce à ses optimisations internes pour les grandes matrices.

Débit arithmétique (Tableau 5 cf Annexes)

Le débit arithmétique, mesuré en GFLOPS/s, reflète l'efficacité du code à exploiter les ressources matérielles. Le code non optimisé atteint un débit relativement faible, de 2,85 GFLOPS/s pour $N = 1$ à un maximum de 8,85 GFLOPS/s pour $N = 6$, avec une légère baisse à 8,02 GFLOPS/s pour $N = 7$, probablement due à une saturation des ressources ou à des inefficacités dans la gestion mémoire.

La version optimisée par vectorisation manuelle et parallélisation atteint des débits bien plus élevés, de 8,76 GFLOPS/s pour $N = 1$ à 28,91 GFLOPS/s pour $N = 6$, avec une légère baisse à 26,48 GFLOPS/s pour $N = 7$. Ces gains (environ 3x à 3,5x) sont dus à une meilleure exploitation des cœurs multiples et des instructions vectorielles.

La version BLAS présente un débit inférieur pour $N = 1$ (1,91 GFLOPS/s), confirmant son inefficacité pour les petites tailles de matrices. Cependant, pour $N \geq 2$, le débit croît rapidement, atteignant 38,52 GFLOPS/s pour $N = 7$, surpassant la version vectorisée pour $N \geq 5$. Cela démontre que les routines BLAS, comme `cblas_dgemm`, sont particulièrement efficaces pour les calculs intensifs impliquant de grandes matrices, grâce à une gestion optimisée du cache et à des algorithmes vectorisés.

Synthèse

Les résultats montrent que les optimisations par vectorisation manuelle et parallélisation offrent des améliorations significatives pour tous les degrés N, avec des gains constants en temps et en débit. L'utilisation de BLAS, bien que moins performante pour les petits N, devient avantageuse pour $N \geq 4$, où elle rivalise ou surpasse la version vectorisée. Ces observations suggèrent que :

— Pour des simulations avec des degrés faibles ($N \leq 3$), la vectorisation manuelle et la parallélisation sont préférables en raison de leur simplicité et de leur efficacité immédiate.

— Pour des degrés élevés ($N \geq 4$), l'utilisation de BLAS est recommandée, car elle exploite mieux les ressources matérielles pour les calculs matriciels de grande taille.

Ces conclusions peuvent guider le choix des optimisations en fonction des besoins spécifiques de la simulation, en tenant compte de la taille des données et des ressources matérielles disponibles.

5 Conclusion

Dans ce projet, nous avons cherché à améliorer les performances d'un code de simulation de la propagation d'ondes scalaires en 3D, reposant sur une méthode d'éléments finis discontinus de type Galerkin, pour des maillages tétraédriques avec des polynômes de degré $N \in [1, 7]$. Notre analyse initiale a mis en évidence que la phase d'exécution – en particulier la mise à jour du terme de droite (RHS) – représentait la majorité du temps de calcul, avec des limitations liées aux accès mémoire et à un manque d'optimisations pour les degrés élevés ($N \geq 4$).

Nous avons exploré deux approches d'optimisation : (1) une optimisation manuelle via la parallélisation avec OpenMP, la vectorisation SIMD, le pré-calcul et la réorganisation des boucles, et (2) une implémentation s'appuyant sur les routines BLAS (notamment `cblas_dgemv`), également avec parallélisation et vectorisation. Les deux approches ont permis d'obtenir des gains significatifs, bien que dépendants du degré N.

Grâce à l'optimisation manuelle, nous avons obtenu une vitesse trois fois plus grande que pour le cas non optmisé en prenant par exemple N=3, en grande partie grâce à une parallélisation efficace des boucles sur les éléments k (avec un gain de 65,19 % sur la partie RHS) et à une vectorisation ciblée. Cela dit, certaines tentatives, comme l'inversion des boucles dans les sections 1.2 et 1.3, ont conduit à des ralentissements (jusqu'à $-22{,}76\,\%$) dus à une perte de localité mémoire. Quant à la vectorisation de la boucle sur les termes de surface (section 1.3), elle a donné des résultats mitigés, avec un gain de seulement 4,05 % sans directive SIMD, ce qui montre que forcer la vectorisation peut parfois être contre-productif, notamment pour des boucles courtes ou avec des accès mémoire peu efficaces. Pour aller plus loin, nous aurions pu affiner notre étude en calculant le nombre réel d'opérations flottantes effectuées par le code optimisé (ce qui n'était pas exigé dans le cadre du projet). Il aurait également été pertinent d'explorer d'autres pistes d'optimisation, comme la vectorisation et la parallélisation de boucles intermédiaires, ou encore l'usage de fonctionnalités avancées d'OpenMP.

L'implémentation basée sur BLAS s'est révélée moins performante pour $N = 1$, en raison de surcoûts liés à l'initialisation. En revanche, elle a surpassé notre approche manuelle pour $N \geq 4$, atteignant un débit de 38,52 GFLOPS/s pour $N = 7$, contre 26,48 GFLOPS/s avec l'optimisation manuelle. Ce résultat s'explique par l'efficacité des routines BLAS dans les calculs matriciels de grande taille, qui tirent parti du cache et des instructions vectorielles.

Comme **axe d'amélioration d'optimisation avec BLAS de la librairie MKL**, on pourrait essayer d'introduire la fonction `Cblas_dgemm( )`. Ainsi au lieu de 16 appels de la fonction `Cblas_dgemv( )`, on pourrait plutôt avoir 4 appels de la fonctions `Cblas_dgemm( )`, ce qui optimiserait de façon accrue les performances.

En résumé, pour les simulations de faible degré ($N \leq 3$), nous recommandons une optimisation manuelle à base d'OpenMP et de SIMD, qui offre un bon compromis entre "simplicité" de mise en œuvre et performances. Pour les degrés plus élevés ($N \geq 4$), l'utilisation des bibliothèques BLAS est à privilégier, car elle permet d'exploiter pleinement les ressources matérielles pour les calculs intensifs. Ces conclusions nous donnent des pistes claires pour orienter le choix des optimisations selon les besoins spécifiques des simulations et l'architecture matérielle disponible. Enfin, il serait intéressant d'envisager des pistes futures mêlant approches hybrides, combinant BLAS pour les calculs matriciels, techniques manuelles pour d'autres parties du code, et des optimisations spécifiques à certaines architectures comme AVX-512.

6 Annexes

Dégré des polynômes	temps (s) du code non optimisé
$N = 1$	23,64
$N = 2$	30,64
$N = 3$	53,25
$N = 4$	103
$N = 5$	121
$N = 6$	204
$N = 7$	288

TABLE 2 – temps d'exécution du code initiale en fonction du dégré des polynômes

Dégré des polynômes	code non optimisé (GFLOPS/s)
$N = 1$	2.85
$N = 2$	4,89
$N = 3$	6,48
$N = 4$	7,52
$N = 5$	8,52
$N = 6$	8,85
$N = 7$	8,02

TABLE 3 – Tableau des performances en débit arithmétique (GFLOPS/s) en en fonction du dégré des polynômes

Dégré des polynômes	temps (s) du code non optimisé	temps (s) du code non optimisé par vectorisation manuelle et parallélisation	temps (s) du code optimé en utilisant BLAS
$N = 1$	23,64	7,7	35,25
$N = 2$	30,64	9,61	23,6
$N = 3$	53,25	16,4	23,8
$N = 4$	103	32,82	32,7
$N = 5$	121	37,57	30,7
$N = 6$	204	62,46	48,2
$N = 7$	288	87.26	60

TABLE 4 – Tableau des performances en temps (s) en fonction du dégré des polynômes

Dégré des polynômes	code non optimisé (GFLOPS/s)	par vectorisation manuelle et parallélisation (GFLOPS/s)	code optimé en utilisant BLAS (GFLOPS/s)
$N = 1$	2.85	8,76	1,91
$N = 2$	4,89	15.6	6,37
$N = 3$	6,48	21,04	14,49
$N = 4$	7,52	23,62	23,71
$N = 5$	8,52	27,44	33.59
$N = 6$	8,85	28,91	37.46
$N = 7$	8,02	26.48	38.52

TABLE 5 – Tableau des performances en débit arithmétique (GFLOPS/s) en en fonction du dégré des polynômes