Amélioration de la Vitesse d'Exécution de la Génération de Bruit Blanc par la Parallélisation et la Vectorisation

FOSSI Cédric

20 Janvier 2024

Table des matières

1 Utilisation de OpenMP

La FFT, dans sa forme récursive originale, bien que conceptuellement simple, présente des défis pour la parallélisation efficace. En effet, la nature récursive de l'algorithme entraîne des dépendances de données qui limitent les opportunités de parallélisation. Pour surmonter ces contraintes, nous avons transformé la FFT en une version itérative.

La fonction `fft` séquentielle est donc récursive comme donné dans l'énoncé tandis que la fonction `fftp` qui utilise l'optimisation avec OpenMP va utiliser une approche itérative. Cette transformation va donc reposer sur deux étapes clés : le *bit reversal* et le calcul itératif des étapes de la FFT.

Dans la première étape, nous réorganisons les données d'entrée en utilisant un schéma de permutation de renversement de bits, une opération qui s'utilise bien à la parallélisation grâce à l'indépendance des opérations pour chaque élément. Ensuite, dans la boucle principale de la FFT itérative, nous appliquons les opérations de la FFT de manière itérative plutôt que récursive. Cette approche itérative permet donc d'exploiter pleinement les capacités de parallélisation offertes par des technologies comme OpenMP, en réduisant les dépendances de données et en permettant le traitement simultané de multiples éléments du tableau. On a donc une exécution plus rapide et plus efficace, particulièrement bénéfique pour les grands ensembles de données typiques dans les applications de traitement du signal et d'image.

```
// Fonction pour réorganiser les éléments d'un tableau selon le schéma de permutation de renversement de bits
Comment Code
void bit_reverse_copy(const double complex *src, double complex *dest, u64 n) {
    u64 log2n = log2(n); // Calcul du logarithme en base 2 de n pour déterminer le nombre de niveaux dans l'arbre FFT

    // Utilisation de OpenMP pour la parallélisation de la boucle
    #pragma omp parallel for
    for (u64 k = 0; k < n; ++k) {
        u64 rev = 0;// Variable pour stocker l'indice inversé
        // Boucle pour calculer l'indice inversé (bit-reversal) de k
        for (u64 i = 0; i < log2n; ++i) {
            if (k & (1 << i))
                rev |= 1 << ((log2n - 1) - i);
        }
        dest[rev] = src[k]; // Copie de l'élément à la position inversée dans le tableau de destination
    }
}
```

La fonction `bit_reverse_copy` effectue ici une réorganisation efficace des éléments d'un tableau de nombres complexes en utilisant le schéma de permutation de renversement de bits. En parcourant chaque indice du tableau d'entrée, elle calcule l'indice inversé (bit-reversal) en inversant les bits, puis copie l'élément correspondant dans un tableau de destination. La parallélisation avec OpenMP améliore les performances en permettant le traitement simultané des indices, optimisant ainsi la préparation des données pour des algorithmes tels que la transformation de Fourier rapide (FFT). On fera ainsi appel à la fonction dans `iterative_FFT`.

Les bénéfices de l'utilisation d'OpenMP dans ce code résident dans l'accélération des boucles qui peuvent être traitées de manière indépendante. La parallélisation est notamment appliquée lors de la réorganisation des éléments du tableau (`bit_reverse_copy`) et lors de la boucle principale de la FFT itérative (`iterative_FFT`). Cette approche permet de réduire le temps d'exécution en effectuant simultanément des calculs sur plusieurs éléments du tableau, profitant ainsi des architectures multicœurs.

En résumé, l'utilisation d'OpenMP dans ce code améliore l'efficacité des calculs liés à la FFT en exploitant la parallélisation, ce qui peut être particulièrement bénéfique pour

des ensembles de données volumineux dans les applications de traitement du signal.

2 Utilisation de la vectorisation

Nous avons aussi appliqué des techniques de vectorisation et de parallélisation pour améliorer les performances du programme. Ce rapport détaille nos choix stratégiques concernant où et comment ces techniques ont été mises en œuvre, ainsi que les raisons pour lesquelles certaines parties du code n'ont pas été vectorisées.

2.1 Choix de Vectorisation

2.1.1 Génération de Bruit Blanc

Nous avons choisi de vectoriser cette partie du code car les calculs pour chaque élément sont indépendants les uns des autres. L'utilisation des instructions AVX nous a permis de traiter quatre éléments simultanément, ce qui a significativement accéléré le processus.

2.1.2 Normalisation

Cette étape a également été vectorisée pour des raisons similaires. Chaque élément du tableau peut être traité de manière indépendante, rendant la vectorisation particulièrement efficace ici.

2.1.3 Non-Vectorisation de Certaines Parties

Nous avons décidé de ne pas vectoriser certaines parties, comme les transformations de Fourier (FFT et iFFT). La raison principale est la complexité des dépendances entre les données dans ces opérations, rendant la vectorisation moins efficace et plus difficile à mettre en œuvre.

2.2 Performance et Vectorisation

Il est important de noter que la vectorisation ne garantit pas toujours une amélioration des performances. Les raisons en sont multiples :

— **Dépendances de Données** : Si les opérations sur les données sont interdépendantes, la vectorisation peut s'avérer complexe ou inefficace.

— **Coût de Gestion des Instructions SIMD** : L'utilisation d'instructions SIMD peut introduire des coûts supplémentaires, tels que l'alignement des données ou la gestion des cas où la taille des données n'est pas un multiple de la largeur du vecteur SIMD.

— **Surcharge de Gestion des Threads** : Dans un environnement parallèle, la gestion des threads peut introduire une surcharge qui contrebalance les gains potentiels de la vectorisation.

2.3 Suppression des Directives `#pragma omp` dans la Vectorisation

Dans certaines parties vectorisées de notre code, nous avons choisi de retirer les directives `#pragma omp` pour les raisons suivantes :

— **Redondance avec la Vectorisation** : Lorsque les instructions SIMD sont utilisées, elles traitent déjà plusieurs données en parallèle. Ajouter un parallélisme au niveau des threads peut ne pas apporter d'avantages supplémentaires.

— **Complexité et Coût de Gestion des Threads** : La gestion des threads dans un contexte déjà vectorisé peut complexifier le code et introduire des coûts de performance supplémentaires.

2.4 Conclusion

En résumé, notre approche de vectorisation a été appliquée de manière ciblée pour optimiser les performances là où c'était possible et efficace. Nous avons pris en compte les complexités et les limitations potentielles de cette approche pour équilibrer les gains de performance avec la maintenabilité et la lisibilité du code.

3 Résultats

3.1 Tableau des valeurs des temps selon la puissance de 2

Puissance 2	Temps vecto	Temps parallele	Temps sequentiel	max
2	0.006643	0.007872	0.000841	4.62205e-113
4	0.003419	0.005275	0.000187	1.30967e-58
8	0.004229	0.005889	0.000233	4.29282e-32
16	0.002511	0.003446	0.000362	1.45165e-18
32	0.004675	0.009168	0.000165	3.64137e-12
64	0.008641	0.004604	0.000378	5.92499e-09
128	0.003132	0.006312	0.000179	3.28585e-07
256	0.005824	0.007731	0.000556	1.8087e-06
512	0.003140	0.003880	0.000642	1.22254e-05
1024	0.006153	0.009948	0.001132	4.01416e-05
2048	0.006453	0.005346	0.001253	0.000186282
4096	0.005475	0.005475	0.002231	0.000703947
8192	0.010669	0.012527	0.002981	0.00273294
16384	0.011345	0.013037	0.007427	0.012123
32768	0.016902	0.017461	0.015543	0.0481664
65536	0.030617	0.027254	0.026402	0.192026
131072	0.048466	0.055273	0.072122	0.809406
262144	0.123623	0.133052	0.147753	3.43184
524288	0.206809	0.212088	0.368172	16.6069
1048576	0.410373	0.411005	1.144366	60.5386
2097152	0.858596	0.835712	2.119032	177.218
4194304	1.658998	1.649653	4.184248	389.611
8388608	3.553408	3.549367	8.041762	747.426
16777216	7.315671	6.919970	13.520095	1622.22
33554432	20.080917	20.216329	32.618072	3216.44

FIGURE 1 – Tableau des valeurs des temps selon la puissance de 2

Voici un aperçu des résultats que nous avons obtenus en enregistrant toutes les valeurs pour les trois programmes fournis. Un constat notable est que le résultat maximal reste

constant, ce qui est attendu étant donné que les trois programmes devraient produire la même sortie, même si les temps d'exécution varient. Nous avons ensuite créé un graphique pour visualiser l'évolution des temps en fonction de la puissance de 2.

Il est intéressant de noter qu'à partir de 262 144 (2^{18}), nous observons un écart de temps croissant entre le code séquentiel et les deux autres solutions. Cette observation met en évidence l'efficacité de l'optimisation réalisée via OpenMP et la vectorisation, en particulier lorsque le volume de données augmente.

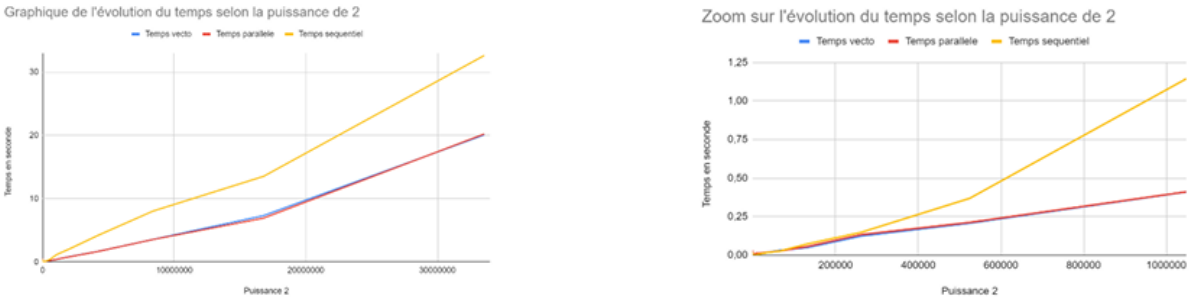

FIGURE 2 – Graphiques

4 Utilisation du code et choix des options

4.1 Utilisation du Makefile et Contraintes de Taille des Données

Nous avons créé un Makefile pour simplifier l'utilisation du code fourni dans ce projet. Le Makefile compile les différents fichiers source (`fft.c`, `fftp.c`, `fftp_vecto.c`) et génère les exécutables correspondants (`fft`, `fftp`, `fftp_vecto`). Pour compiler le code et exécuter avec la taille par défaut de 33 554 432 (nous pouvons ajuster en changeant la valeur dans le Makefile grâce aux différentes puissances de 2 en commentaires juste au-dessus), il suffit d'utiliser la commande `make execute`.

4.2 Limitations de Taille des Données

La taille de 33 554 432 a été choisie comme valeur par défaut pour l'exécution sur notre ordinateur. Au-delà de cette taille, le processus est interrompu car la manipulation de nombres complexes entraîne un dépassement de capacité mémoire. Pour des tailles de données plus importantes, nous utiliserons Grid5000, qui est plus adapté pour ces cas-là. Bien que nous ayons effectué des tests sur Grid5000 pour des nombres plus grands, nous avons décidé de maintenir la taille de 33 554 432 pour des raisons de praticité.

4.3 Options d'Optimisation `-O3` et AVX

Nous avons intégré l'optimisation de niveau 3 (`-O3`) dans le processus de compilation. Cela permet au compilateur de beaucoup optimiser les performances du code généré.

De plus, nous utiliserons aussi l'option de compilation avec l'ensemble d'instructions AVX (Advanced Vector Extensions) (`-mavx`). Cette option tire parti des fonctionnalités d'optimisation vectorielle des architectures modernes pour accélérer les calculs, notamment dans la manipulation de nombres complexes.

4.4 Exécution avec Taille Personnalisée

L'exécution du code avec une taille personnalisée peut être réalisée en spécifiant la taille désirée à l'aide de l'option `-size`. Par exemple, la commande suivante générera un fichier audio avec une taille de 1024 :

```
./fft --size 1024 --output son.wav
```

4.5 Gestion des Problèmes de Taille

Nous reconnaissons que des problèmes peuvent survenir lorsque la taille des données devient trop importante, en raison de la consommation accrue de mémoire par les nombres complexes. Pour résoudre ce problème, la valeur maximale de taille de données qui fonctionne sur notre système a été identifiée et utilisée comme limite.

4.6 Création d'un Makefile Complet

Afin de simplifier davantage le processus de compilation et d'exécution, nous avons mis en place un Makefile complet. En utilisant la commande `make execute`, le Makefile compile automatiquement le code et exécute le programme avec la taille par défaut de 33 554 432.

5 Conclusion

Nous avons donc constaté une amélioration significative de nos temps d'exécution grâce à l'application des deux méthodes traitées, à savoir l'utilisation d'OpenMP et la vectorisation. Bien que ces approches aient démontré leur efficacité, il est important de noter que nous n'avons malheureusement pas pu approfondir notre analyse en effectuant la *Roofline Analysis* en raison de contraintes temporelles. Ce serait un aspect intéressant à explorer à l'avenir pour une compréhension plus approfondie des performances de notre système.