Rapport Projet ASR
Vectorisation de convolution de matrices

Proposé et encadré par : M. Delalandre Mathieu
Réalisé par : Vinet Dylan et Lezé–Robert Corentin

Table des matières :

1 Introduction :

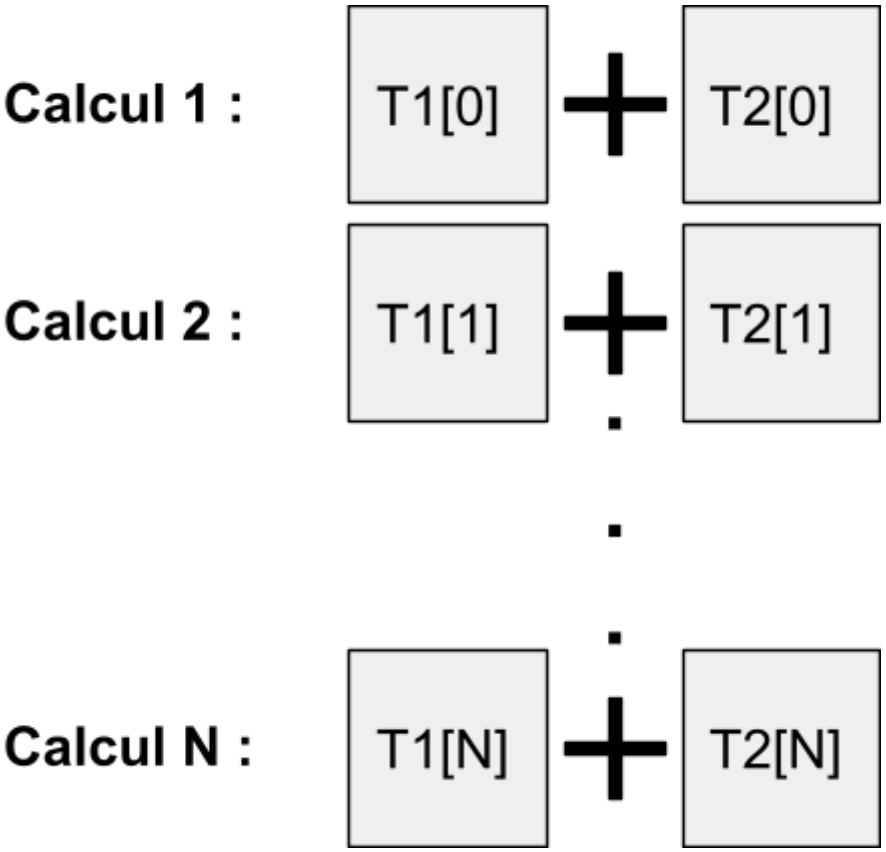
Dans ce rapport, nous proposons une solution pour le projet “Station TV” mis en place au DI. Le but de ce projet est d’enregistrer en continu plusieurs chaines TV française afin d’effectuer divers traitements sur les données récupérées. Les diffusions enregistrées peuvent servir à faire du fact checking par exemple, ou bien de la reconnaissance d’image, etc. Tous ces projets nécessitent de faire beaucoup de traitement d’image, et donc beaucoup de convolution de matrices. Le but de notre projet est donc de mettre en place un programme vectorisé qui effectue des convolutions de matrices afin d’obtenir un gain de performances. Nous allons également effectuer des mesures de temps d’exécution sur les versions normales, automatiquement vectorisés et vectorisés manuellement du programme pour évaluer le gain de performances.

2 Introduction à la vectorisation :

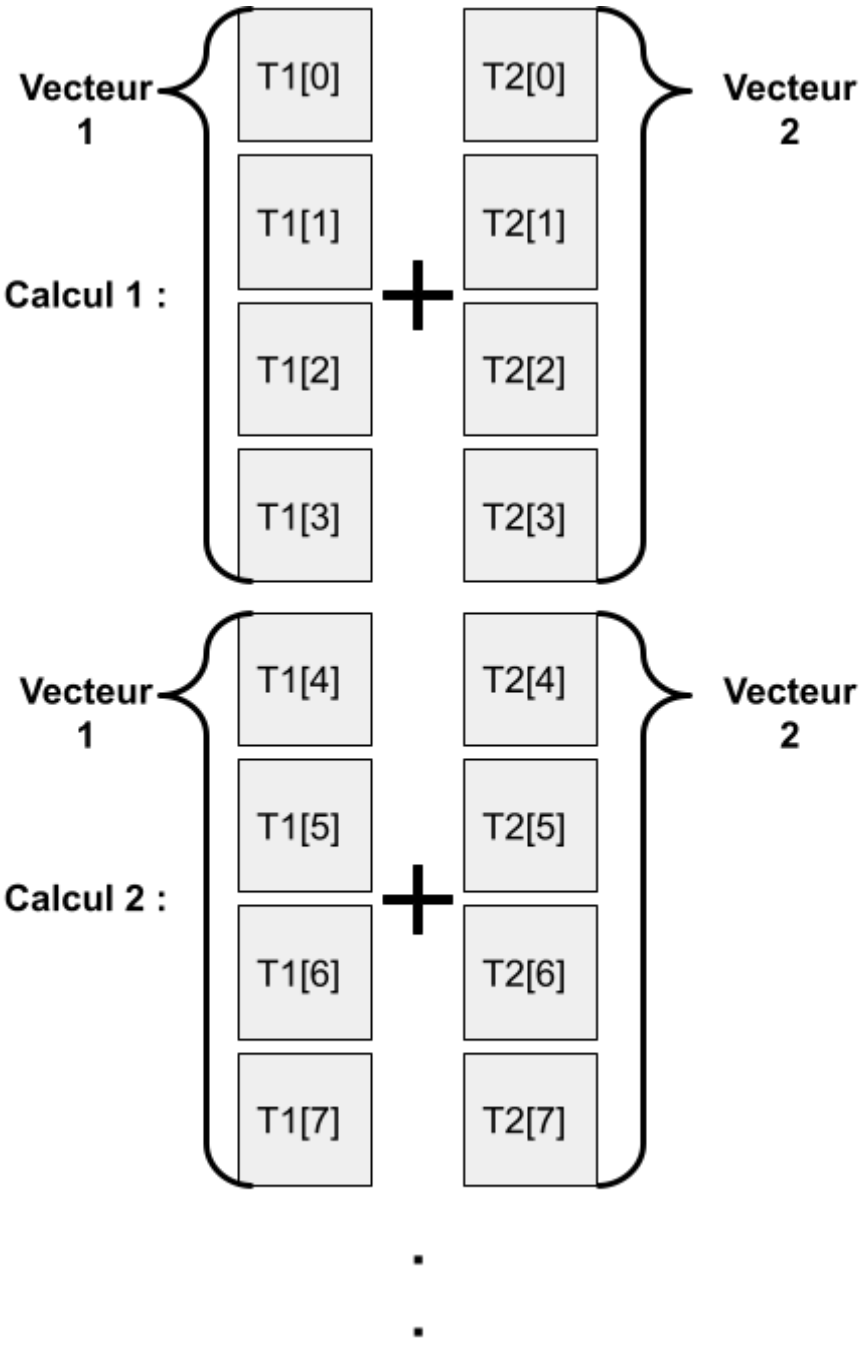
2.1 Le concept de la vectorisation :

La vectorisation est une méthode qui permet d'effectuer des calculs terme à termes simultanés pour accélérer les calculs en les effectuant des opérations entre deux vecteurs contenant plusieurs valeurs plutôt que simplement entre deux valeurs.

Prenons en exemple deux jeux de données de temps T1 et T2. On souhaite additionner la valeur T1[0] avec T2[0], puis T1[1] avec T2[1] etc… Pour des jeux de donnés de taille N, cela implique de faire N calculs :



Vectoriser ce calcul avec un vecteur de taille 4 permettrait de faire uniquement N/4 calculs, car au lieu d'additionner les temps un à un, on les additionnerait 4 par 4, en utilisant deux vecteurs :



C’est cette méthode que nous allons appliquer aux convolutions de matrices afin de les accélérer.

2.2 SIMD :

Pour appliquer la vectorisation, nous allons appliquer du SIMD (Single Instruction Multiple Data). Dans notre cas “Multiple data” parce que nous faisons nos opérations sur toutes les valeurs des vecteurs en même temps et “Single Instruction” car nous ne faisons l’opération qu’en un seul calcul.

Nous allons plus spécifiquement utiliser de l'AVX. Ce sont des instructions processeurs Intel disponibles en C et C++ permettant de stocker des valeurs dans des vecteurs comme illustré précédemment et d'effectuer des calculs groupés.

2.3 AVX :

Il existe trois grandes générations d'AVX :

- L'AVX
- L'AVX2
- L'AVX512

L'AVX simple se fait avec des vecteurs de 128 bits, l'AVX2 avec des vecteurs de 256 bits et l'AVX512 avec des vecteurs de 512 bits. Le sujet de notre projet était à l'origine d'effectuer la vectorisation en AVX512, en revanche, aucun de nous deux n'ayant une machine personnelle pouvant compiler de l'AVX512 et n'ayant pas accès facilement à une machine permettant d'effectuer la compilation au DI non plus, nous avons décidé de passer en AVX2.

Toutes les générations d'AVX s'utilisent globalement de la même manière. Au-delà de la taille des vecteurs qui change, de nouvelles instructions sont ajoutées à chaque génération afin de simplifier la manipulation des vecteurs et d'effectuer des calculs plus complexes, plus rapidement.

Les instructions AVX sont assez bas niveau, car elles permettent de demander au processeur d'effectuer une instruction précise. Effectivement, c'est normalement au compilateur de choisir quelle instruction utiliser, puis comment l'utiliser pour effectuer le calcul demandé. Dans le cas de l'AVX, il faut quand même utiliser un compilateur car même si on indique quelle instruction utiliser, on n'indique pas comment effectuer le calcul. Coder en encore plus bas niveau reviendrait à s'affranchir du compilateur et coder directement en assembleur.

2.4 Fonctionnement de l'AVX2 :

Comme les générations d'AVX fonctionnent de manière similaire, nous allons expliquer son fonctionnement en AVX2 afin d'éviter la redondance.

2.4.1 Inclusion :

Pour pouvoir coder en AVX, il suffit d'inclure "immintrin.h" pour utiliser n'importe lequel des générations. Il y avait avant plusieurs fichiers headers à inclure en fonction de la génération et des fonctionnalités que l'on souhaitait utiliser, mais cela est désormais très déconseillé. Effectivement, beaucoup d'instructions n'existent que dans une seule génération, mais sont utilisés partout, il vaut mieux donc tout inclure.

2.4.2 Les registres de vecteur :

Pour utiliser AVX, il n'est pas possible de simplement déclarer un vecteur en C et de demander au processeur d'effectuer des calculs avec. Il faut utiliser les types de données

fournies par Intel afin de charger les valeurs dans un registre du processeur pour qu'il puisse effectuer les calculs directement sur le vecteur.

Il existe 3 type de vecteurs de 256 bits (Les autres générations d'AVX suivent cette même logique, mais avec des vecteurs de taille différente) :

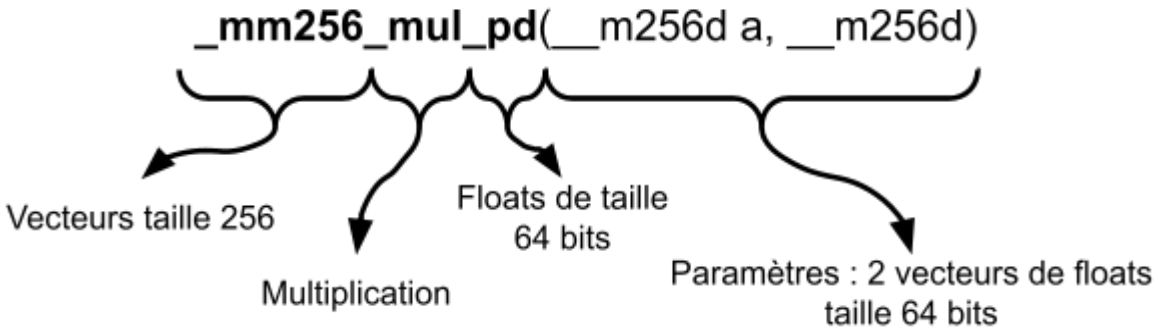
- **__m256** : Les 256 bits sont séparés en 8 flottants de 32 bits.
- **__m256i** : Les 256 bits sont séparés en entiers de tailles non spécifiées (Peuvent être 4 de 64 bits / 8 de 32 bits / 16 de 16 bits / 32 de 8 bits).
- **__m256d** : Les 256 bits sont séparés en 4 flottants de 64 bits (des doubles).

2.4.3 Convention de nommage :

Avant d'expliquer comment charger les valeurs dans les registres, il est bon d'expliquer la manière dont est construite la convention de nommage des fonctions AVX. Chaque fonction est divisée en 3 parties :

- Le préfixe qui indique la taille du vecteur sur lequel nous allons utiliser la fonctionnalité.
- Le corps qui indique la fonction à utiliser (addition, multiplication, charger des données, etc).
- Le suffixe qui indique le type de vecteur utilisé (Les données vues plus haut, donc flottant de 32 bits, entiers de 8 bits etc).

Voici, par exemple, à quoi ressemble une multiplication sur des vecteurs 256 bits de flottants 64 bits :



À noter que les flottants sont indiqués par "ps" et "pd" pour "Packed Singles" et "Packed Doubles" et que les entiers sont notés "epi" suivi de leur taille ("epi32" par exemple) pour "Extended Packed Int".

2.4.4 Charger les vecteurs :

Nous allons également directement voir les instructions qui permettent de charger ces valeurs dans les registres. Il existe deux méthodes.

La première est le "set" qui est une fonction qui prend directement en paramètre les valeurs à charger dans le registre. Pour charger par exemple 8 flottants 32, on peut utiliser "set" ainsi :

```
__m256 a = _mm256_set_ps(1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0);
```

Cette méthode ne nécessite pas de déclarer les variables au préalable. En revanche, il est également possible de charger une liste déjà existante avec la seconde méthode qui est le “load”. Par exemple, si on définit une liste de flottants “d”, il est possible de le charger ainsi :

```
__m256 e = _mm256_loadu_ps(d);
```

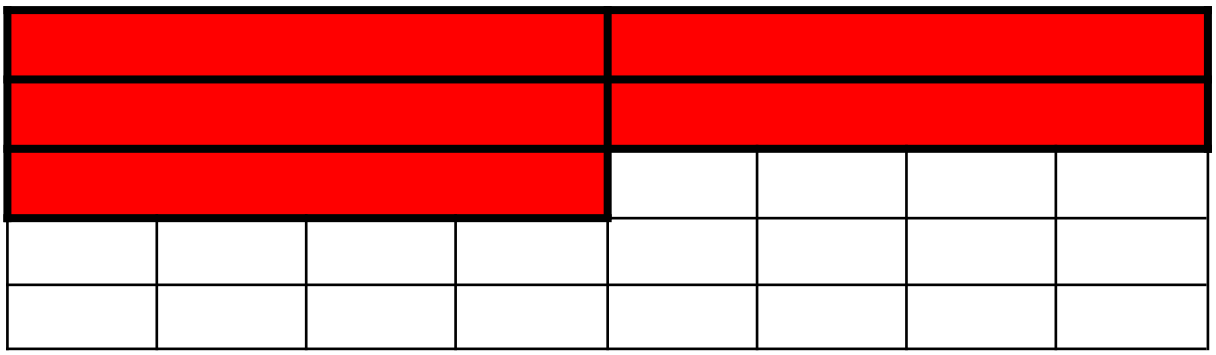
Ces vecteurs sont désormais utilisables dans des instructions AVX. On remarque en revanche que nous avons utilisé un “loadu” et non un “load”. C’est parce que nous n’avons pas aligné la mémoire de la liste d.

2.4.5 Alignement de mémoire :

Afin d’optimiser encore plus la vectorisation, il est possible d’aligner la mémoire afin d’éviter les zones de vides inutiles. Pour aligner la mémoire, il faut allouer les pointeurs des variables sur des adresses mémoires multiples de 8 pour que tous les blocs soient bien imbriqués sans laisser d’espaces.

Voici à quoi peut ressembler les blocs d’une mémoire non alignée dans laquelle on alloue des blocs de 4 :

De l’espace perdu (bloc gris) se forme. En revanche, si on aligne les adresses mémoire sur des multiples de 8, voici à quoi ressemble la mémoire :



Aucun espace n’est perdu.

Cet alignement doit être fait au moment du malloc ou bien, il sera important d’ajouter un “u” dans le corps des fonctions qui se comporte différemment en fonction de l'alignement afin de préciser que la mémoire n’est pas alignée.

2.4.6 Opérations :

Nous n'allons pas décrire toutes les opérations possibles en AVX, mais il est bon de savoir qu'on peut y retrouver les opérations arithmétiques de base. Voici la syntaxe des opérations (en suivant la convention de nommage expliqué précédemment) :

- __m256 c = _mm256_add_ps(__m256 a, __m256 b);
  - Pour l'addition
- __m256 c = _mm256_mul_ps(__m256 a, __m256 b);
  - Pour la multiplication
- __m256 c = _mm256_sub_ps(__m256 a, __m256 b);
  - Pour la soustraction
- __m256 c = _mm256_div_ps(__m256 a, __m256 b);
  - Pour la division

Pour les opérations arithmétiques, il existe également une fonction qui effectue la multiplication puis l'addition à la suite, ce qui nous aurait été très utile pour les convolutions. Malheureusement, cette fonction n'existe qu'en AVX512.

Il est également possible de faire des comparaisons telles que :
__m256 c = _mm256_and_ps(__m256 a, __m256 b);

Finalement, il est possible de réarranger les données d'un vecteur avec "permute" qui réarrange l'ordre des données en prenant un vecteur d'indices en paramètres. Ou bien "shuffle" qui réarrange tout aléatoirement.

2.4.7 Récupération des données :

Une fois l'opération vectorisée effectuée, il faut récupérer les données dans des variables C / C++ classiques afin de pouvoir les exploiter dans le reste du programme.

Pour cela, nous allons tout simplement utiliser la fonction "store". Par exemple, pour stocker un vecteur résultat "c" dans une liste "d", voici la syntaxe :

```
_mm256_store_ps(d, c);
```

2.5 Compilation :

Pour compiler de l'AVX, nous avons utilisé GCC. En fonction du processeur et de son architecture, toutes les instructions ne sont pas forcément activées.

Le premier moyen de s'assurer que GCC compilera l'AVX, est de spécifier l'architecture du processeur utilisé avec "-march=" suivi de l'architecture. Le plus simple est "-march=native" qui va prendre l'architecture de la machine utilisée et activer toutes ses instructions dont l'AVX2 s'il est supporté. Il est également possible de spécifier une autre architecture pour utilisation sur une autre machine ou sur un groupe de machines qui tombent dans la même catégorie (graniterapids / rocketlake / icelake / skylake etc).

Le second moyen est de spécifier la sous-famille d'instructions directement. Dans notre cas, spécifier "-mavx2" nous permettra de compiler.

Chaque architecture à ses propres performances associées.

2.6 Autovectorisation :

Par défaut, le compilateur va essayer d'autovectoriser le code fournir, nous avons donc spécifié "#pragma loop(no_vector)" au-dessus de nos boucles effectuant les calculs matriciels. Cela nous a permis de désactiver l'autovectorisation pour effectuer nos mesures.

Nous voulions également mesurer l'efficacité de l'autovectorisation, pour cela rien à rajouter dans le code. Nous nous sommes tout de même assurés que le code suivait les règles fournies par Intel, qui permettent de vérifier que le code est bien vectorisable :

- La boucle doit être composée de code ligne à ligne. C'est-à-dire qu'il faut éviter de faire des sauts dans le code tels que les "switch".
- La durée de la boucle doit être déterminable avant de rentrer dedans. Autrement dit, pas de boucle dont la taille varie en fonction de ce qui est calculé dedans.
- Il ne doit pas y avoir de dépendance d'une boucle à une autre. Par exemple, il ne faut pas que ce qui est calculé dans l'itération 1 soit nécessaire pour calculer l'itération 2.

Nos programmes de convolution et multiplication de matrices n'ont pas enfreint ces règles, nous avons donc pu mesurer nos performances.

3 Organisation / outils :

Pour ce projet, nous devons programmer en C ou C++, mais avec une préférence pour le C. Nous avons choisi d'utiliser l'IDE Visual Studio Code pour 2 raisons :

1) Nous étions tous les deux familier avec cette IDE.
2) On peut installer des extensions sur cette IDE. L'une d'elle, nommée LiveShare, permet de pouvoir travailler à plusieurs en simultané sur un même projet, de la même manière qu'un Google Doc.

Pour l'utilisation des résultats du programme, nous avons choisi Google Sheet, qui donne plus de contrôle sur la création des histogrammes que Excel.

4 Résultats :

4.1 Convolutions de matrices

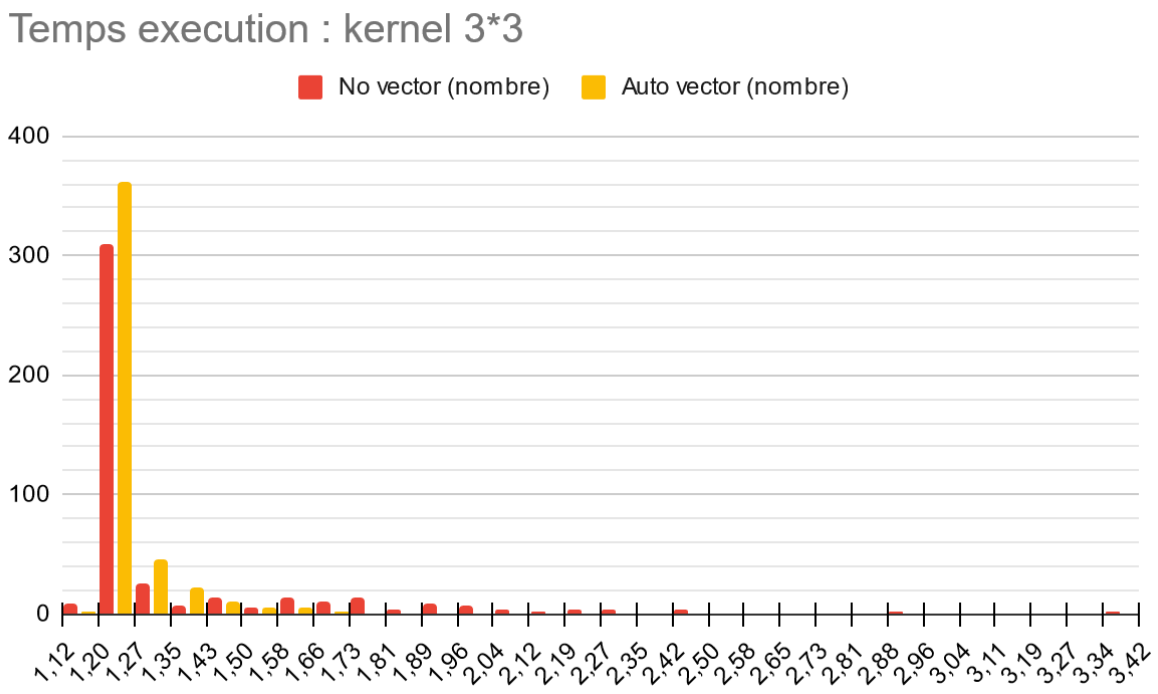
Avant de voir les données récolté, il faut savoir que :

- Les matrices utilisés sont de taille 200*200, avec des valeurs entières de 0 à 255
- Il y a, au total, 455 itérations, et une itérations équivaut à 455 convolutions
- Différents tailles de Kernel ont été utilisés

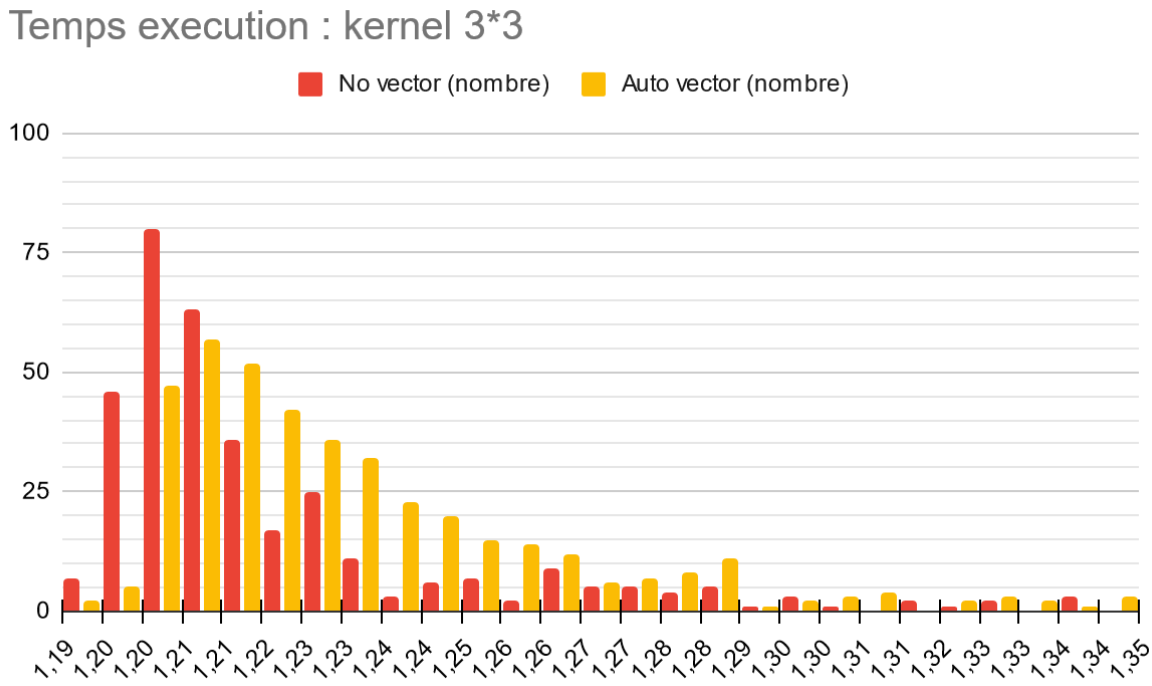
	MOYENNE TEMPS			***MEDIANE TEMPS***		
	Kernel 3*3	Kernel 11*11	Kernel 29*29	Kernel 3*3	Kernel 11*11	Kernel 29*29
non vectorisé	1,386	1,322	1,329	1,219	1,3	1,299
auto-vectorisé	1,257	1,319	1,32	1,23	1,302	1,299

On remarque que la moyenne de temps d'exécution est toujours inférieure lorsqu'on auto-vectorise, mais la médiane de temps est un peu plus élevée. Cependant, l'écart entre les deux est plutôt faible.

On a ensuite pris les données de la convolutions avec le kernel 3*3 pour en faire un histogramme.



On remarque qu'il y a beaucoup "d'artefact" du côté non vectorisé, des durées très extrêmes comparé à la majorité observé. Mais globalement, à cette échelle, les durées sont très proches.



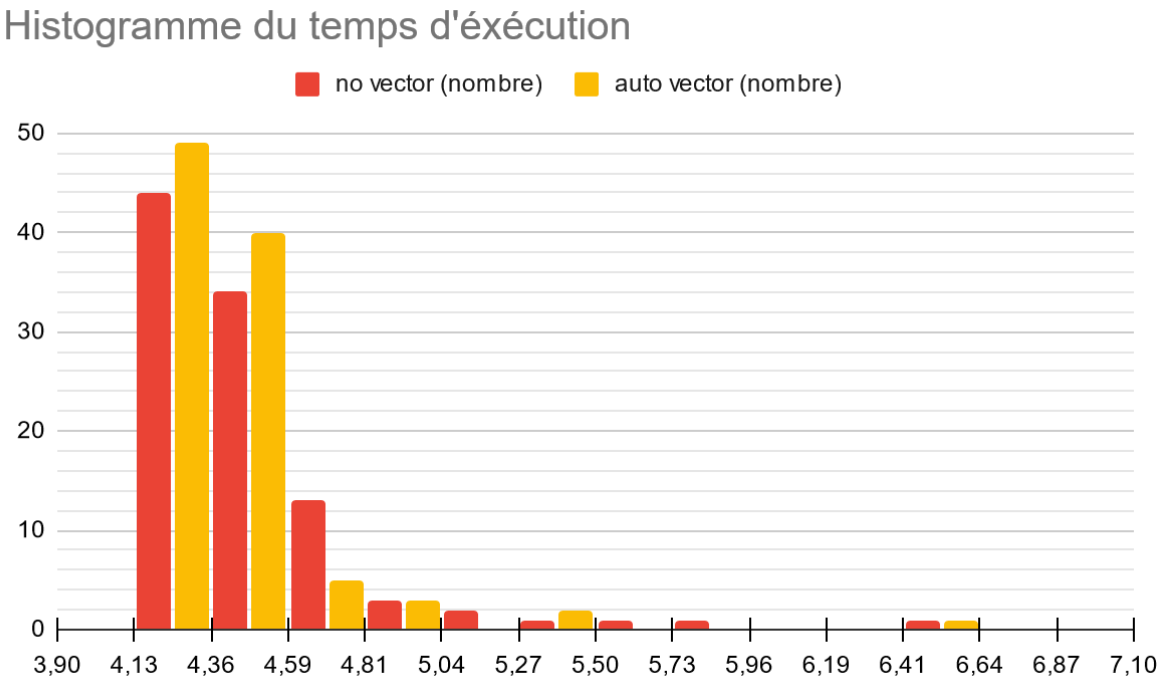
Lorsque l'on zoom sur les données, on remarque que c'est finalement l'exécution non vectorisée qui est le plus rapide le plus souvent. On peut conclure que, bien que la version non vectorisée est techniquement plus rapide, la version auto-vectorisée à l'avantage d'être plus constante sur sa durée. On remarque le même comportement quelle que soit la taille du kernel.

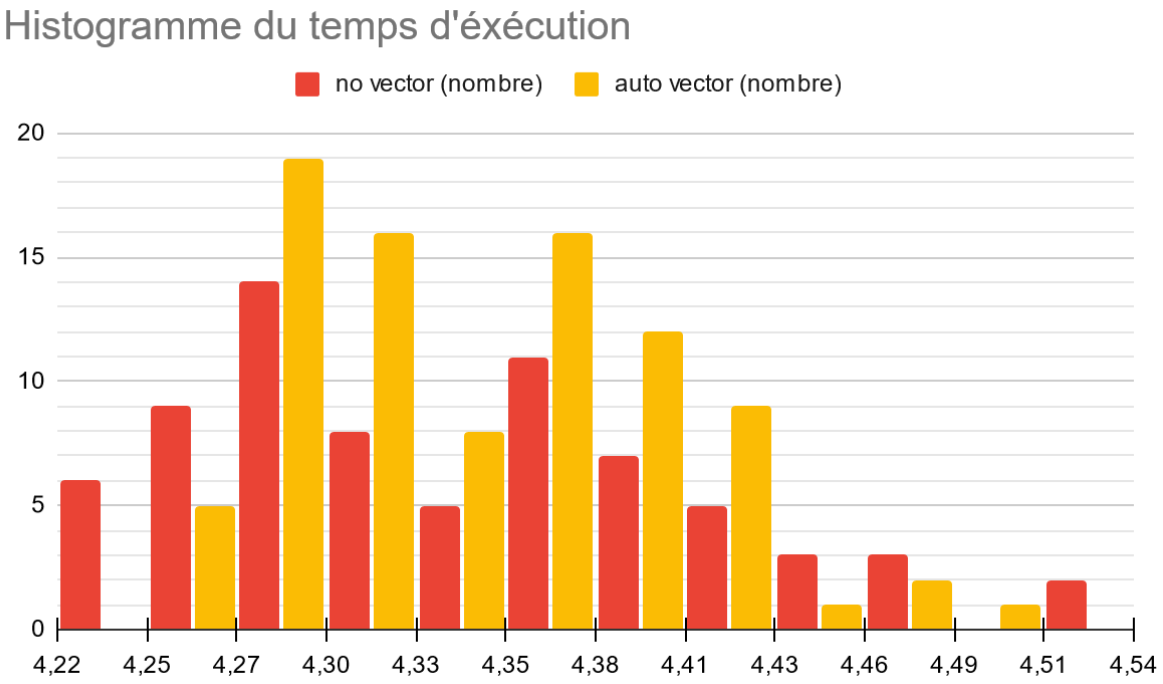
4.2 Multiplication de matrice

Lorsqu'on a commencé à s'attaquer à AVX-512, on nous a demandé de commencer par des multiplications de matrice plutôt que la convolution, ce qui nécessite de refaire un jeu de données.

- Les matrices sont de taille 256*256, avec des valeurs entières entre 0 et 255
- Il y a, au total, 100 itérations, et 1 itération équivaut à 100 multiplication
- Le produit calculé est celui de la matrice avec lui-même

	MOYENNE	*MÉDIANE*
non vectorisé	4,489	4,3715
auto-vectorisé	4,424	4,358



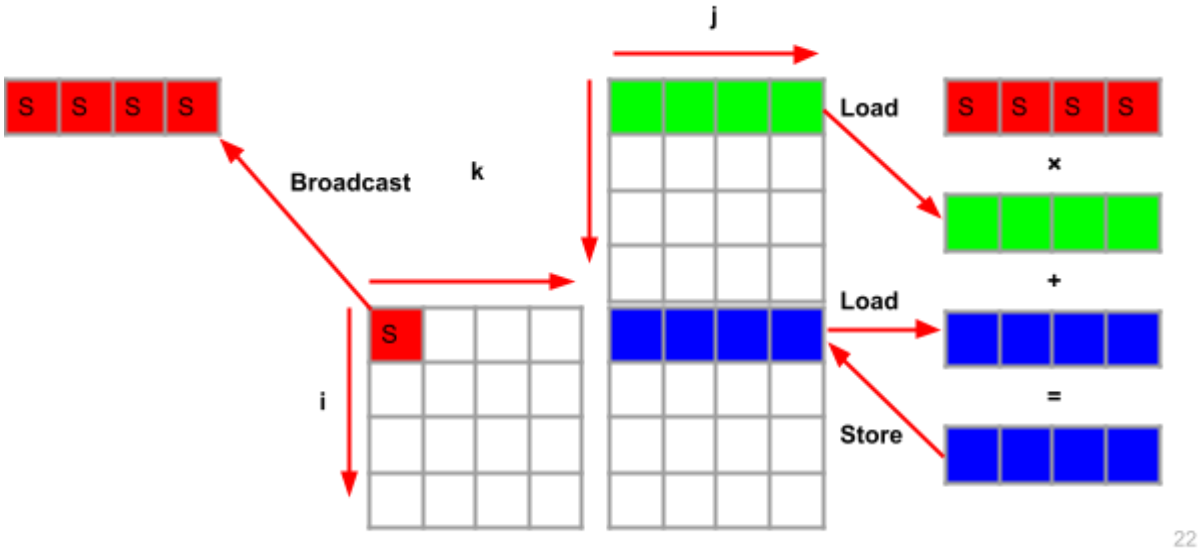


Avec ces données, on observe le même comportement entre les versions non vectorisées et auto-vectorisée que pour les convolutions :

- Les moyennes et médiane entre les deux sont très proches
- La version non vectorisée est techniquement plus rapide, mais possède un peu plus d'artéfact
- La version auto-vectorisée est plus constante.

On n’a pas eu le temps malheureusement de finir notre version manuellement vectorisée de la multiplication de matrice. Cependant, nous savons comment l’implémenter.

D’habitude, on parcourt d’abord les lignes de la matrice A puis les colonnes de la matrice B et enfin k. Ici, on va itérer d’abord la valeur k avant j. Ensuite, on prend la valeur A[i, k] que l’on broadcaste (C'est-à-dire que l’on remplit un vecteur de doublons de cette valeur), puis que l’on multiplie à la ligne B[k]. On additionne le produit à la ligne i de la matrice de résultat.



Toutefois, il faut gérer les différents cas en fonction de la taille de la matrice. Si la matrice est plus grande que la taille des vecteurs de l’AVX, alors il faudra itérer sur j par pas de la taille du vecteur. De plus, il faudra gérer aussi le cas où la taille de la matrice n’est pas un multiple de la taille du vecteur.

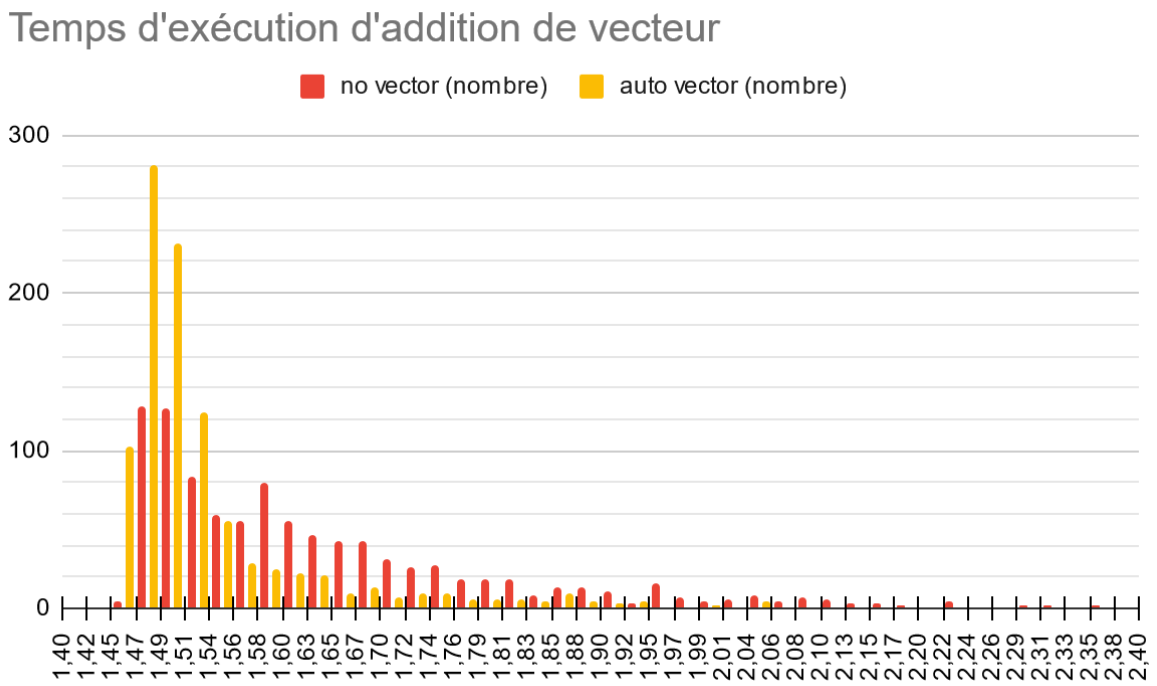
4.3 Addition de vecteur

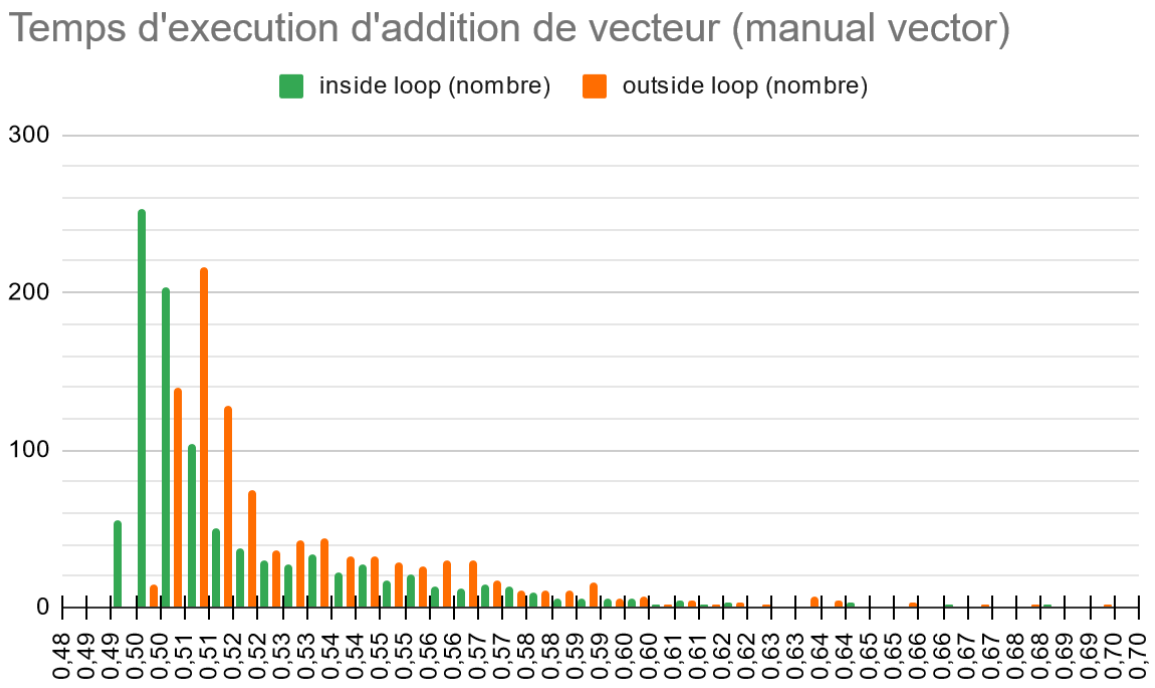
N'ayant pas réussi à manuellement vectoriser la multiplication de matrice, nous avons essayé de vectoriser une opération plus simple : l'addition de vecteur.

- Les vecteurs sont de taille 8 et contiennent des flottants.
- Il y a, au total, 1000 itérations, et 1 itération correspond à 100 000 000 d'addition de vecteur.
- On va vectoriser de deux manières :
    - Une première version où on charge les valeurs des vecteurs dans les registres processeurs à chaque itération (inside loop). Cela ressemble plus à ce que nous aurions fait pour de la multiplication de matrices où chaque vecteur est un bout de la matrice et doit donc être rechargé.
    - Une où on ne convertit qu'avant la boucle (outside loop).
- Nous comparons ces résultats à une version non vectorisée (Ce coup-ci, nous désactivons l'autovectorisation dans GCC au lieu de mettre des flags dans le code) et une version autovectorisée.

		MOYENNE	*MEDIAN*
No vector		1,646933	1,5945
Auto vector		1,541973	1,501
Manual vector	**inside loop**	0,518553	0,504
	outside loop	0,529669	0,515

On remarque ici que, malgré la sous-version, le temps moyen et le temps médian de la version manuellement vectorisée est 3 fois plus rapide que les versions non vectorisées et auto-vectorisées.





On peut remarquer ici l'énorme écart qu'il y a : la pire durée d'exécution pour une version manuellement vectorisée reste 2 fois plus rapide environ que les exécutions non vectorisées ou auto-vectorisées. On remarque aussi que le temps d'exécution manuellement vectorisé est très constant, mais que la version inside loop est plus efficace.

5 Problèmes rencontrés :

5.1 Documentation :

Lorsqu'on a commencé à lire la documentation de l'AVX, on a eu beaucoup de difficultés. Premièrement parce que la documentation n'était pas forcément à jour, ce qui nous a envoyé sur beaucoup de fausses pistes. Beaucoup des résultats trouvés étaient très indigeste et nous avons mis beaucoup de temps à comprendre. On a ensuite essayé d'implémenter des programmes "Hello World" trouvés pour essayer de se donner des bases pour pouvoir débuter, mais aucun ne fonctionnait.

Ce qui a rendu cela difficile, c'était aussi le fait que les générations d'AVX étaient mélangées dans les documentations. Mélanger l'AVX et l'AVX2 n'est pas un souci, en revanche comme durant la période 2016-2018, Intel mettait le support AVX512 sur tous ses processeurs, beaucoup de ressources en ligne se sont mises à mélanger les trois. De plus, certaines nouvelles instructions AVX512 portent le nom "_mm256" comme de l'AVX2.

Enfin, la convention de nommage de l'AVX est assez nébuleuse. Au-delà des soucis de compréhension que cela nous a causés, cette convention de nommage a complètement brouillé les moteurs de recherche. Effectivement, tout le long du projet, nous avons eu beaucoup de mal à avoir des résultats pertinents avec des termes tels que "vpdpbusd".

5.2 AVX512 :

Après avoir fini de programmer les versions non vectorisée et auto-vectorisée, on a commencé à regarder le fonctionnement de l'AVX512, et on a pris beaucoup de temps. On a essayé pendant cette période d'écrire des programmes l'utilisant mais rien ne fonctionnait. La raison est que nos machines, et les ordinateurs de l'école, ont des processeurs qui ne sont pas compatibles avec AVX512. Ayant compris ça trop tard, il a fallu passer à l'AVX-2 qui lui est compatible.

Ce problème s'est mélangé au fait que le sujet est assez difficile à apprendre (notamment à cause de la documentation). Effectivement, nous avions bien pensé à regarder si nos ordinateurs étaient compatibles avant de faire de l'AVX512. Sauf qu'au moment où nous avons vérifié, nous n'avions probablement pas une compréhension du SIMD assez poussée pour comprendre que l'AVX512 spécifiquement poserait un problème. De plus, "gcc" ne donnera jamais d'erreur si on lui demande de compiler en AVX512 alors que le processeur ne le supporte pas, le fichier exécutable généré ne fera juste rien. Nous avons donc perdu beaucoup de temps à essayer de compiler sans comprendre pourquoi nos programmes ne marchaient pas.

5.3 Le retrait de l'AVX512 :

Nous en avons brièvement parlé plus tôt, mais toutes les ressources en ligne et notre compréhension se sont retrouvées changées par cette période de 2016-2018. Intel s'est

mis, en 2016 à sortir des processeurs compatibles avec l'AVX512 mais a subitement décidé de rendre le support de l'AVX512 exclusif à ses processeurs de serveurs haute performance en 2018. De ce que nous avons compris, Intel a décidé que l'AVX512 les forçait à employer une architecture particulière de processeur pour supporter les registres de 512 bits. Mais cette architecture n'était pas bonne du tout pour les processeurs pour ordinateurs personnels. Le retrait de l'AVX512 n'a jamais été clairement annoncé, ce qui nous a donné de fausses attentes. L'idée que nos machines récentes ne pouvait pas supporter les dernières instructions ne nous est pas venue à l'esprit immédiatement.

5.4 Les entiers :

Là où les flottants ont deux types distincts en AVX, le type de vecteur qui contient des entiers ne contient pas d'informations sur leur taille. Nous avons mis du temps à comprendre comment les charger (car il n'existe pas une fonction pour chaque taille d'entiers), mais nous avons a priori réussi. Nous pensons qu'il faut charger un vecteur de 256 bits sans spécifier de type. C'est l'opération à effectuer (par exemple une addition) qui lira le vecteur en fonction de ce qu'on lui indique. Pour une addition de deux vecteurs de 16 entiers de taille 16 bits, l'addition découpera le vecteur en 16 et retrouvera elle-même les valeurs avec cette syntaxe :
_mm256_add_epi16

Malheureusement, il reste encore à récupérer le résultat dans une variable exploitable, mais nous n'y avons pas réussi. Il existe des fonctions "store" pour les epi32 et epi64 exclusivement en AVX512, nous n'avons donc pas pu les utiliser. Nous pensons qu'il est possible de récupérer les bits à la main et de les trier et de les recaler pour former les valeurs, mais nous n'avons malheureusement pas eu le temps de nous en occuper.

Conclusion

Si cette tendance s'observe sur les opérations de multiplication de matrice ou la convolution, il y a un grand intérêt pour manuellement vectoriser avec AVX. Mais pour cela il faut réfléchir différemment les opérations, notamment celle de convolution.

Une autre question à se poser est de savoir quelle version d'AVX utilisé. La station TV est compatible avec l'AVX 512, qui possède de meilleures performances mais est beaucoup moins portable que l'AVX 2.

Bibliographie :

Vérifier que boucle bien vectorisable :
https://www.intel.com/content/www/us/en/developer/articles/technical/requirements-for-vectorizable-loops.html

Explications :
https://www.intel.com/content/www/us/en/developer/articles/technical/vectorization-essential.html

https://en.wikichip.org/wiki/x86/avx512_vnni

Exemples d'avx en c :
https://devblogs.microsoft.com/cppblog/accelerating-compute-intensive-workloads-with-intel-avx-512/

https://www.intel.com/content/www/us/en/developer/articles/code-sample/intel-advanced-vector-extensions-512-intel-avx-512-new-vector-neural-network-instruction.html

https://github.com/kshitijl/avx2-examples/blob/master/examples/01-hello.c

Cours :
https://www.uio.no/studier/emner/matnat/ifi/IN3200/v19/teaching-material/avx512.pdf

https://inst.eecs.berkeley.edu/~cs61c/sp19/pdfs/lectures/lec18.pdf

https://learn.microsoft.com/en-us/dotnet/api/system.runtime.intrinsics.x86.avx?view=net-7.0

https://github.com/Triple-Z/AVX-AVX2-Example-Code/blob/master/Initialization_Intrinsics/src/load.c4

Vidéos :
https://www.youtube.com/watch?v=I3efQKLgsjM

https://www.youtube.com/watch?v=AT5nuQQO96o

Options de compilation :
https://gcc.gnu.org/onlinedocs/gcc/x86-Options.html

Documentation :
https://www.intel.com/content/www/us/en/developer/library.html?query=&currentPage=1&externalFilter=guid%3Aetm-309fa72139794adfb4272f0509026587%3B&s=Newest

https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#cats=Set&techs=AVX_ALL&ig_expand=6292,6369,6378&text=set1

https://www.felixcloutier.com/x86/vpbroadcast