Ecole Polytechnique de l'Université François Rabelais de Tours
Département Informatique
64 avenue Jean Portalis
37200 Tours, France
Tél. +33 (0)2 47 36 14 14
www.polytech.univ-tours.fr

2015-2016

Projet ASR

Évaluation de performance en vectorisation de programmes natifs vs. systèmes virtuels

Tuteurs académiques
Mathieu Delalandre

Étudiants
Jonathan Le Bonzec (DI5)
Jean-François Béchu (DI5)
Lorry Moreau (DI5)

27 décembre 2015

Pour citer ce document :
Jonathan Le Bonzec, Jean-François Béchu et Lorry Moreau, *Projet ASR: Évaluation de performance en vectorisation de programmes natifs vs. systèmes virtuels*, , Ecole Polytechnique de l'Université François Rabelais de Tours, Tours, France, 2015-2016.

```
@mastersthesis{
    author={Le Bonzec, Jonathan and Béchu, Jean-François and Moreau, Lorry},
    title={Projet ASR: Évaluation de performance en vectorisation de programmes natifs vs.
       systèmes virtuels},
    type={},
    school={Ecole Polytechnique de l'Université François Rabelais de Tours},
    address={Tours, France},
    year={2015-2016}
}
```

Document mis en forme sous LaTeX
grâce à la classe de document `polytech.cls` V1.0 (2015/11/18)

Table des matières

Table des figures

Introduction

La vectorisation est un cas particulier de parallélisation. En tant que telle, elle promet des gains de performances aux systèmes qui l'utilisent. Cette promesse est-elle valable pour tous les systèmes : virtuels comme natifs ? Dans quelle mesure ?

Dans ce contexte, nous nous proposons de comparer les performances offertes par la vectorisation sur des systèmes natifs et sur des systèmes virtuels. Nous avons fait le choix de comparer un code C++ à un code Java. Le traitement à réaliser, imposé dans le sujet, consiste à réaliser un "ou" logique entre deux opérandes de 4 Ko puis compter le nombre de bits à 1.

Nous commencerons par une étude théorique de la vectorisation avant de nous attacher à comparer les performances en vectorisation.

Cette étude s'inscrit dans le cadre des projets ASR du premier semestre de l'année 2015-2016.

1 Vectorisation : la théorie

1 Architecture des ordinateurs

On peut distinguer les différentes architectures des ordinateurs selon différents critères, comme ceux mis en évidence par la taxinomie de Flynn [WWW2]. Elle prend en compte l'organisation du flux d'instructions et de données pour distinguer quatre cas (Figure 1) :

— Single Instruction on Single Data
C'est l'architecture classique des ordinateurs telle que présentée par Von Neumann. On a un seul flux d'instructions s'appliquant à un unique flux de données, aucun parallélisme ici.

— Single Instruction on Multiple Data
Cette architecture exploite la possibilité de parallélisme au niveau de la mémoire et donc des données. On a une instruction unique qui s'applique à plusieurs données simultanément. C'est à cette architecture que l'on s'intéressera par la suite.

— Multiple Instructions on Single Data
Cette architecture propose un parallélisme au niveau des instructions en introduisant plusieurs unités de calcul travaillant réalisant donc plusieurs instructions, mais sur une même donnée.

— Multiple Instructions on Multiple Data
Cette architecture propose un parallélisme au niveau des instructions et des données. Plusieurs unités de calcul réalisent plusieurs instructions sur des données distinctes. On distingue plusieurs types : à mémoire partagée (les unité de calcul adressent la même mémoire), à mémoire distribuée (chaque unité de calcul œuvre sur sa propre mémoire) ou à mémoire hybride.

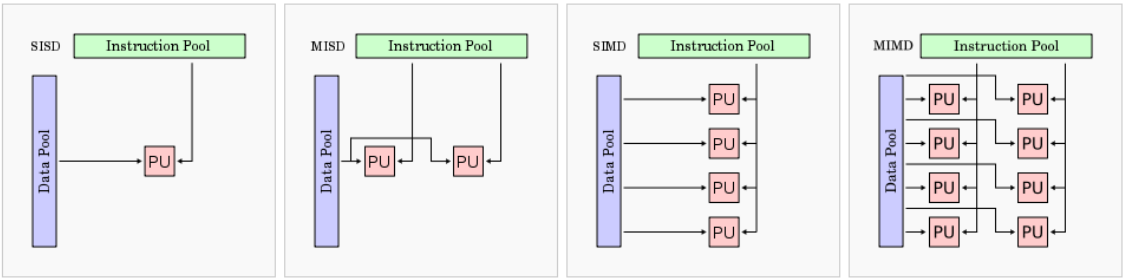


Figure 1 – *Taxinomie de Flynn*

2 Qu'est-ce que la vectorisation ?

La vectorisation est un cas particulier de la parallélisation [WWW3] rendu possible par le passage d'une architecture type SISD (Single Instruction on Single Data) à une architecture type SIMD (Single Instruction on Multiple Data).

Pour faire l'analogie avec les mathématiques, il s'agit du procédé permettant de passer d'un code scalaire à un code vectoriel.

À titre d'exemple, prenons l'ajout un à un de tous les éléments d'un tableau A à tous les éléments d'un tableau B vers un tableau C. Comme présenté en Figure 2, un code scalaire effectuerait l'opération sur chaque ligne du tableau successivement, alors qu'un code vectoriel effectuerait l'opération sur plusieurs lignes simultanément.

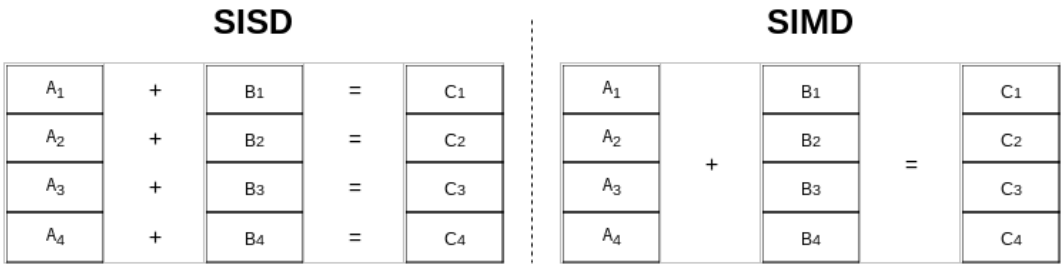


Figure 2 – *Comparaison SISD SIMD*

On définit le vecteur comme l'espace mémoire utilisé pour réaliser les opérations en parallèle. Ici, la taille du vecteur est de 4. En pratique, on essaiera au maximum d'utiliser des vecteurs de la taille des registres disponibles pour notre architecture. En effet, la vectorisation est réalisée au niveau physique grâce au registres.

Prenons pour exemple l'opération c[i] = a[i] OR b[i], avec a, b et c des tableaux d'entiers. En Figure 3, on observe le déroulement de l'opération non vectorisée : successivement, chaque cellule des tableaux opérandes est associée à une partie de registre puis l'opération est réalisée. A l'inverse, en Figure 4, on observe son homologue vectorisé : plusieurs cellules sont copiées de manière à utiliser la totalité du registre, et l'on réalise une seule opération sur tout le registre.

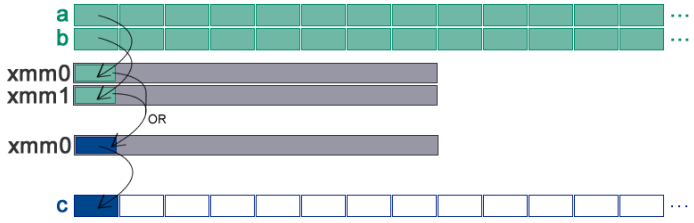


Figure 3 – *Opération logique non vectorisée*

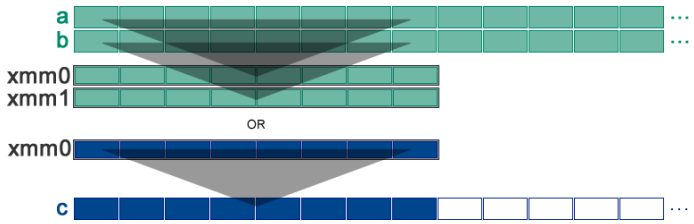


Figure 4 – *Opération logique vectorisée*

Avec les machines actuelles, la taille des registres est bien souvent de 128 bits ou plus (512 avec le jeu d'instructions AVX par exemple).

3 Vectorisation automatique

Comme le précise [WWW3], "la vectorisation automatique est un sujet de recherche majeur en informatique qui consiste à trouver des méthodes permettant à un compilateur d'effectuer cette vectorisation de manière autonome". La vectorisation automatique existe notamment sur le compilateur gcc, et est activable grâce à un flag ou aux paramètres d'optimisation.

Il est important de bien comprendre comment fonctionne la vectorisation automatique, afin de développer de manière à profiter de cette fonctionnalité. Notons que la vectorisation automatique, comme toutes les optimisations, ne doit en aucun cas modifier le comportement du programme. De même, toutes les dépendances doivent être respectées, et la précision d'entier doit être maintenue.

Tout d'abord, il faut comprendre les dépendances entre les différentes variables et les réaligner si nécessaire. Ensuite, il faut organiser correctement les implémentations d'instructions des candidats.

3.1 Graphe de dépendances

Ainsi, le compilateur commence par créer le graphe de dépendances, permettant d'identifier quelles instructions dépendent de quelles autres instructions. Il s'agit d'examiner chaque instruction et d'examiner chaque jeu de données auquel elle accède et détecter si ces données sont utilisées par une autre instruction. Il y a dépendance dès lors que deux instructions accèdent au même emplacement mémoire. Il va de soi que si le graphe indique la dépendance entre deux instructions, il sera impossible de les vectoriser. Le graphe de dépendance contient toutes les dépendances locales ayant une distance inférieure strictement à la taille du vecteur. Toutes les autres dépendances non cycliques ne devraient pas invalider la vectorisation puisqu'il n'y aura pas d'accès concurrent dans la même instruction vectorielle. Ainsi, si la taille des registres est de 128 bits, et que l'on a un tableau d'éléments de taille 32 bits, on a un vecteur de taille 128/32 = 4. Cela signifie que sur notre registre, on pourra aligner 4 éléments consécutifs de notre tableau. Prenons cet exemple :

```
for (i = 0; i < 128; i++) {
  a[i]=a[i-16]; // 16>4, Peut être supprimé du graphe de dépendance
  a[i]=a[i-1]; // 1<4, Reste sur le graphe de dépendance
}
```

La première instruction fait intervenir une distance de dépendance supérieure à la taille de notre vecteur, et peut donc être supprimée du graphe de dépendance sans problème. La seconde en revanche, fait intervenir une distance de dépendance trop courte : elle reste sur le graphe. Dans la pratique, cet algorithme aurait un comportement indéterminé : à la première itération on essaie d'accéder à a[-16], qui n'est potentiellement pas alloué. Cela n'enlève rien à la valeur d'illustration de l'exemple.

3.2 Clustering

En utilisant le graphe, le compilateur peut maintenant regrouper les composantes fortement connexes (CFC) et isoler les déclarations vectorisables. Imaginons un fragment de programme contenant trois groupes de déclaration à l'intérieur de boucles : (CFC1 + CFC2), CFC3 et CFC4, dans cet ordre, avec seulement CFC3 vectorisable. Le programme final contiendra ainsi trois boucles, une pour chaque groupe, avec seulement celle du milieu vectorisée. Il est impossible d'assembler la première et la dernière, sous peine de violer l'ordre d'exécution des instructions, qui est une garantie nécessaire.

3.3 Détection des idiomes

On peut optimiser davantage la vectorisation en se basant sur les expressions idiomatiques. Si on prend l'expression suivante, a[i] = a[i] + a[i+1], on remarque que les données du membre droit sont d'abord récupérées puis stockées sur le membre gauche. Il est donc impossible que les données changent au cours de l'assignation. L'auto-dépendance par scalaires peut être vectorisée par l'élimination de variables.

3.4 La vectorisation automatique à l'usage

En pratique, le compilateur ne procède à ces optimisations qu'à la demande : soit en demandant un niveau d'optimisation de 2 ou plus, soit en utilisant le flag "-ftree-vectorize". Reste à alors à s'assurer des déductions qu'il a pu faire quant à la vectorisation des instructions.

La première étape pour observer la vectorisation automatique consiste à étudier les logs de compilation avec le flag "-ftree-vectorizer-verbose". On peut ainsi confirmer ce qui a été considéré comme vectorisable et ce qui ne l'a pas été.

Mais pour bien comprendre comment le compilateur vectorise le code, le plus efficace reste d'observer les fichiers avant assemblage.

Nous avons pu observer que le compilateur gère en général tous les cas. Ainsi, la plupart des fonctions qu'il a détecté comme vectorisables commencent par une multitude de tests sur l'alignement des données, leur dépendance ... Dans certains cas, il sera en mesure de modifier certaines données présentant des obstacles à la vectorisation pour ensuite revenir au traitement vectorisé. Dans d'autres cas, il utilisera tout simplement une autre version de la fonction n'exploitant pas la vectorisation afin d'éviter le moindre ennui.

Il est possible de donner davantage d'informations à propos des données au compilateur pour qu'il se dispense de ces tests et, par la même occasion, pour s'assurer que les données seront bien exploitées par la version vectorisée du code compilé. Toutefois, il est assez difficile de faire disparaitre la totalité des tests : le compilateur souhaitant rester vigilant, il reste strict à l'égard des tests.

4 Vectorisation manuelle

On l'a vu, le processus de vectorisation automatique est complexe. Il en résulte qu'il est souvent difficile d'obtenir le traitement souhaité après compilation.

4.1 Méthodes explicites

Dans ce but, certains compilateurs proposent des outils permettant de recourir à une vectorisation explicite. Comme le mentionnent Hwancheol Jeong et ses collaborateurs [3], les compilateurs proposent trois approches :

- Utiliser directement les instructions machines propres à un traitement vectorisé dans le code C ou C++ en recourant à du code assembleur inline. Cette méthode, très bas niveau, est également la plus flexible et puissante. En revanche, elle est complexe à utiliser.
- Utiliser des fonctions proposées par le compilateur et représentant ces mêmes instructions machines dédiées au calcul vectorisé. On les appelle "fonctions intrinsèques". On conserve ainsi la syntaxe C ou C++ et la simplicité de programmation qui l'accompagne (comparée à une programmation en langage assembleur). En revanche, on perd la finesse évoquée au point précédent et les promesses d'optimisation optimale qui l'accompagnait.
- Utiliser les classes dédiées. L'exercice se veut encore plus abordable, au détriment de la performance du traitement une fois compilé.

4.2 Jeux d'instructions et performances

Dans tous les cas, le développeur est limité par les instructions gérées par la machine cible. On distingue deux grands jeux d'instructions :

- SSE qui vient avec des registres de 128 bits. Plusieurs versions viennent enrichir les instructions proposées.
- AVX qui vient avec des registres de 256 bits et des instructions gérant 3 opérandes (A = B + C), à l'inverse des instructions classiques (A = A + B).

Quelle que soit la méthode retenue, et comme on l'a déjà dit, le but est d'exploiter au maximum la capacité des registres pour diviser au maximum le nombre d'itérations nécessaires pour traiter une quantité de données constante.

Mais quelle jeu d'instructions choisir ? Intuitivement, AVX semble le plus prometteur. Toutefois, Hwancheol Jeong et ses collaborateurs ont procédé à plusieurs expérimentations pour répondre rigoureusement à cette question. Elles consistaient à comparer les performances de trois codes réalisant la même opération mais écrits en C++, en assembleur inline utilisant SSE et en assembleur inline utilisant AVX.

Dans un contexte général, pour une simple opération vectorisable, on aura assez peu de différence entre le recours au jeu d'instructions SSE et AVX. En revanche, tous deux sont bien plus performant que le C++ classique (1.7 fois pour l'opération étudiée). En activant la vectorisation automatique et les optimisations liées, cette démarcation disparait et le code C++ vectorisé automatiquement s'aligne sur les performances du code assembleur, qu'il soit SSE ou AVX.

On peut se demander pourquoi il est serait nécessaire de s'embêter avec une vectorisation manuelle. Pour des opérations simples que le compilateur sait vectoriser, le gain est en effet inexistant. La différence se fait sentir lorsque le traitement devient plus complexe et que le compilateur ne parvient pas à réaliser les optimisations que l'on peut réaliser manuellement.

Par exemple, si on réutilise des données au cours d'un traitement, le compilateur ne sera pas capable de relever cet aspect et on aura ainsi de piètres performances en C++ comparé aux résultats obtenus en assembleur (de l'ordre d'un facteur 30 sans optimisation, 15 avec optimisation). Pour ce même traitement, AVX s'avère également 2 à 3 fois plus performant que SSE.

5 Conclusion

En conclusion, la compilation proposée par les compilateurs modernes gère relativement bien la vectorisation automatique pour des traitements simples. Dans de tels cas, il n'est pas nécessaire de descendre à un plus bas niveau pour obtenir de meilleures performances.

En revanche, pour des traitements complexes que le compilateur ne sait pas traduire de manière optimale en instructions vectorisées, il peut être nécessaire d'utiliser l'une des trois méthodes de vectorisation explicite. Le choix de la méthode se fera selon les familiarités de chacun et le degré d'optimisation visé. Quant au jeu d'instructions à exploiter, on a noté qu'AVX était au moins aussi performant que SSE, sinon plus.

2 Mise en œuvre

1 Contexte

Le but de ce projet est de comparer la mise en oeuvre de la vectorisation et ses performances sur des systèmes natifs et sur des systèmes virtuels. Nous avons fait le choix de comparer un code C++ à un code Java, tous deux devant effectuer le même traitement : réaliser un “ou” logique entre deux opérandes prenant la forme de deux mots de 4 Ko, et retourner le nombre de bits à 1 dans ce résultat (aussi appelé poids de Hamming plus loin).

On peut noter le tout : |A+B|.

Avec :

— “+” pour la comparaison OU bit à bit.
— “| . . . |” pour le comptage des bits à 1.

1.1 De la pertinence du contexte

Le problème retenu est un cas particulier d’un problème bien plus répandu : le comptage de bits. A en croire [1], il s’agit d’un problème historique récurrent, notamment en cryptographie.

C’est également un bon moyen de s’assurer qu’un générateur de nombres pseudo aléatoires est aussi aléatoire que possible en vérifiant que la part de bits à 1 dans un très grand mot est proche de cinquante pour cent. Il s’agit d’un des aspects retenus par le National Institute of Standards and Technology (NIST) pour certifier de tels générateurs. Puisque l’on traite de grandes quantités de données, un algorithme de comptage efficace est plus que souhaitable.

Il existe bien d’autres usages à cette fonction, sobrement nommée "popcount", et, en conséquence, de nombreuses approches ont été proposées au fil des années.

1.1.1 Comptage itératif

Il s’agit simplement de masquer le mot en question, bit par bit, d’effectuer le décalage nécessaire à chaque fois et de réaliser une addition au sein d’un compteur global. Cette méthode est très intuitive mais excessivement lourde.

1.1.2 Comptage parallèle

Le comptage parallèle consiste en une succession de "et" logiques, de décalages et d'additions sur des fragments du mot dont on veut déterminer la population.

Pour bien comprendre, appuyons nous sur un exemple avec un fragment de 8 bits. Le traitement s'effectue comme un arbre binaire représenté en Figure 1 et dont les 8 bits seraient les 8 feuilles. A la racine, on a le résultat final.

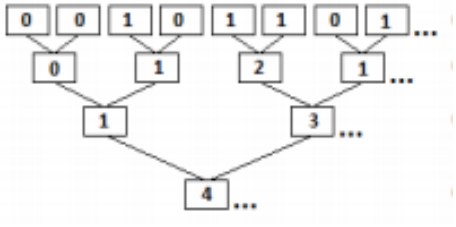


Figure 1 – *Comptage parallèle vu comme un arbre*

On réalise en fait le traitement suivant, appliqué à un exemple en Figure 2 :

	Commentaire	*Bits*							
	Fragment original	0	0	1	0	1	1	0	1
1 bit	Masque haut de taille 1	1	0	1	0	1	0	1	0
	Fragment masqué (et) haut	0	0	1	0	1	0	0	0
	Fragment masqué (et) haut décalé de 1	0	0	0	1	0	1	0	0
	Masque bas de taille 1	0	1	0	1	0	1	0	1
	Fragment masqué (et) bas	0	0	0	0	0	1	0	1
	Addition des deux parties	0	0	0	1	1	0	0	1
2 bits	Masque haut de taille 2	1	1	0	0	1	1	0	0
	Fragment masqué (et) haut	0	0	0	0	1	0	0	0
	Fragment masqué (et) haut décalé de 2	0	0	0	0	0	0	1	0
	Masque bas de taille 2	0	0	1	1	0	0	1	1
	Fragment masqué (et) bas	0	0	0	1	0	0	0	1
	Addition des deux parties	0	0	0	1	0	0	1	1
4 bits	Masque haut de taille 4	1	1	1	1	0	0	0	0
	Fragment masqué (et) haut	0	0	0	1	0	0	0	0
	Fragment masqué (et) haut décalé de 4	0	0	0	0	0	0	0	1
	Masque bas de taille 4	0	0	0	0	1	1	1	1
	Fragment masqué (et) bas	0	0	0	0	0	0	1	1
	Addition des deux parties – résultat final	0	0	0	0	0	1	0	0

Figure 2 – *Comptage parallèle*

— On est d'abord au niveau des feuilles où l'on considère les bits individuellement. Le but est de réunir les bits deux par deux pour le niveau suivant puisqu'on a un arbre binaire. Pour cela, on prend le bit haut (à l'aide d'un masque et d'un "et logique") que l'on décale d'un rang pour qu'il soit aligné avec le bit de poids faible. On réalise alors l'addition des deux pour obtenir le noeud père. On notera que ces opérations logiques peuvent être réalisées sur les 8 bits simultanément.
— On est ensuite au premier niveau, où l'on considère 4 ensembles de deux bits que l'on veut réunir en 2 ensembles de 4 bits. Pour cela, on prend les 2 bits hauts que l'on décale de deux rangs pour qu'ils soient alignés avec les 2 bits de poids faible. On réalise alors l'addition des deux ensembles pour obtenir le noeud père.
— On est finalement au deuxième niveau, où l'on considère 2 ensembles de quatre bits que l'on veut réunir en 1 ensemble de 8 bits. Pour cela, on prend les 4 bits hauts que l'on

décale de quatre rangs pour qu'ils soient alignés avec les 4 bits de poids faible. On réalise alors l'addition des deux ensembles pour obtenir la racine, notre résultat final.

1.1.3 Comptage proposé par le NIST

Pour chaque octet, au lieu de stocker les 8 bits de manière contiguë, on stocke chaque bit dans son propre octet. On occupe ainsi 8 fois plus de mémoire, mais le comptage se résume ensuite en une simple addition de tous ces octets.

1.1.4 Comptage à l'aide d'une lookup table

Le recours à une lookup table permet de pré-calculer et stocker le comptage de population pour des portions de longueur définie. Il suffit ensuite de couper le mot en morceaux de la taille choisie, et, pour chacun d'eux, de récupérer la valeur en lisant la table. Autrement dit, on remplace des traitements lourds par une simple lecture en mémoire. Une addition dans un accumulateur global permet d'avoir le résultat en fin de comptage.

La lookup table doit stocker toutes les valeurs de population pour l'ensemble des valeurs représentables sur le nombre de bits retenu. Autrement dit, avec une lookup table pour des fragments de 8 bits on stockera 256 valeurs, pour des fragments de 16 bits on stockera 65 536 valeurs et pour des fragments de 32 bits on stockera plus de 4 milliards de valeurs. En revanche, plus les fragments sont grands, plus le traitement sera rapide puisqu'on fera moins d'itérations pour parcourir le mot complet (on nuancera plus tard). En somme, plus la table est grande, plus on occupera de mémoire et plus le comptage sera rapide. Il faut donc trouver un bon équilibre.

Il nous faut cependant nuancer ce propos en prenant en compte la notion de cache processeur. Il est de taille réduite, et ne peut pas stocker énormément de valeurs. A chaque fois qu'on récupère une valeur depuis la mémoire secondaire, elle est stockée dans ce cache au détriment d'une autre valeur qui en sort. Si on se ressert de cette valeur par la suite et qu'elle est toujours en cache, on s'épargne un accès mémoire couteux. Or, on voit qu'avec plusieurs milliards de valeurs, un tel phénomène est hautement improbable. De ce point de vue, plus la table sera petite, plus le cache processeur sera source de gain de temps.

On l'a dit, les accès mémoires sont couteux, pas seulement pour récupérer la valeur stockée dans la table. Obtenir les fragments du mot est également couteux. Pour réduire ce problème, les auteurs évoquent une approche un peu moins intuitive. Au lieu de récupérer fragment par fragment, nous allons en obtenir plusieurs en un seul accès mémoire, les stocker en registre processeur et les traiter ensuite un par un. Par exemple, au lieu de faire 8 accès mémoire pour obtenir successivement des fragments de 8 bits, on pourra lire les 64 bits en une seule lecture, les stocker en registre et traiter 8 bits par 8 bits sans nouvel accès mémoire.

1.1.5 Comptage exploitant les registres SSE

Il ne s'agit pas vraiment d'une méthode à part entière, mais plutôt de donner une nouvelle dimension aux méthodes déjà évoquées.

Qu'il s'agisse du comptage parallèle ou par lookup table, on a vu que la taille des registres pouvait avoir un rôle déterminant. Or, avec le jeu d'instructions SSE viennent des registres de 128 bits. C'est autant de place dont on peut disposer dans le cadre d'une "lecture en une fois" avec une lookup table, ou pour stocker les feuilles du comptage parallèle.

On réduit ainsi le temps de traitement en limitant les accès mémoire et le nombre global d'itérations pour compter la population du mot complet (dans le second cas).

1.1.6 Instruction dédiée

On le verra plus loin, mais, compte tenu de la récurrence du problème, les fondeurs ont finalement décidé de proposer une instruction machine dédiée au comptage de bits au sein des processeurs proposant le jeu d'instructions SSE dans sa version 4.2 et supérieures. On assiste là à l'évolution classique de la résolution d'un système : la recherche d'algorithmes performants suivie d'une implémentation matérielle qui les surpasse pour répondre bien plus efficacement au besoin.

1.2 Point de départ : le cas Java

On peut intuitivement suspecter le code exécuté par une machine virtuel d'être plus lent : l'exécution est virtualisée, donc un acteur supplémentaire intervient dans l'exécution du programme. On peut éventuellement réduire une partie de cette charge, dans certaines conditions et en considérant des VM à même de gérer les hotspots.

Toutefois, la machine virtuelle Java n'est pas capable de gérer au delà de 64 bits, limitant ainsi l'exploitation des registres. De plus, elle n'est pas capable de gérer la vectorisation.

A la suite d'un premier essai avec une implémentation intuitive, on constate un temps d'exécution légèrement supérieur à 7 µs sur la machine de test. En passant avec une approche par look-up table (voir ci-après), on atteint les 4 µs.

A quel point peut-on améliorer ces performances avec un code natif en exploitant les possibilités offertes par la vectorisation ?

Tous les résultats mentionnés et le protocole de mesure sont détaillés au chapitre 3.

2 Algorithme naïf

La première approche adoptée a été la plus évidente : utiliser les fonctions dédiées à la comparaison (ou logique) et au comptage de bits (builtin popcount). Cette dernière accepte, au maximum, des opérandes de 64 bits. De manière à l'exploiter au maximum, on lui passera donc de telles opérandes et on parcourera les 4 Ko par morceaux de 64 bits.

Notons que l'implémentation réelle de la fonction "builtin popcount" est laissée à l'appréciation du compilateur selon les flags de compilation définis. Dans notre cas, et pour l'instant, la fonction prend la forme d'un ensemble de décalages et d'opérations logiques.

```
1  virtual void perform(){
2      uint64_t i, nb = 0, iterations = DATA_LENGTH / sizeof(uint64_t);
3
4      for(i = 0; i < iterations; i++) {
5          uint64_t comp = a[i] | b[i];
6          nb += __builtin_popcountl(comp);
7      }
8
9      result = nb;
10 }
```

Ce premier algorithme offre une exécution en 1.6 µs qui nous servira de référence.

3 Algorithme avec LUT, sans vectorisation

Dans cette version de l'algorithme nous introduisons une table de correspondance (ou lookup table). La LUT se présente comme un simple tableau (Figure 3) où les index sont les mots de 16 bits et où les cases sont les résultats de l'opération popcount sur l'index correspondant. La LUT dispose donc de 65 536 cases, c'est à dire le nombre de représentations différentes avec 16 bits. Autrement dit, elle occupe 125 Mo en mémoire. Plus les blocs en entrée sont grands, plus le traitement sera rapide, mais il faudra également une table plus grande. En ce sens, c'est la plus grande puissance de deux raisonnable : avec 32 bits, la table aurait plus de quatre milliards d'entrées, et occuperait donc près de 16 Go.

Index	Valeur
0	0
1	1
2	1
3	2
...	...
1141	6
...	...
65535	16

Figure 3 – *Look-up table utilisée*

On voit, par exemple, que le popcount de 3 (index) vaut 2 (valeur). En effet la valeur binaire de 3 est 0b11. On a bien deux bits à 1.

Une fois notre LUT initialisée on coupe nos deux mots de 4 Ko en fragments de 16 bits. Une boucle va ensuite s'occuper d'exécuter un OU entre chaque fraction de 16 bits.

Dans chaque tour de boucle, après avoir obtenu le résultat R (16 bits) de l'instruction OU, on utilise la table de correspondance. L'algorithme cherche la valeur V à l'index R de la LUT. Cette valeur V correspond au nombre de bits à 1 dans R.

Enfin, le tour le boucle se termine par l'ajout, dans un compteur global, du résultat de l'opération précédente. Ce compteur contiendra, à la fin de l'algorithme, quand l'exécution de la boucle sera terminée, le résultat de |A+B|.

A titre d'exemple, l'exécution d'un tour de boucle est représentée en Figure 4.

— a = 0x1AF5
— b = 0x821C

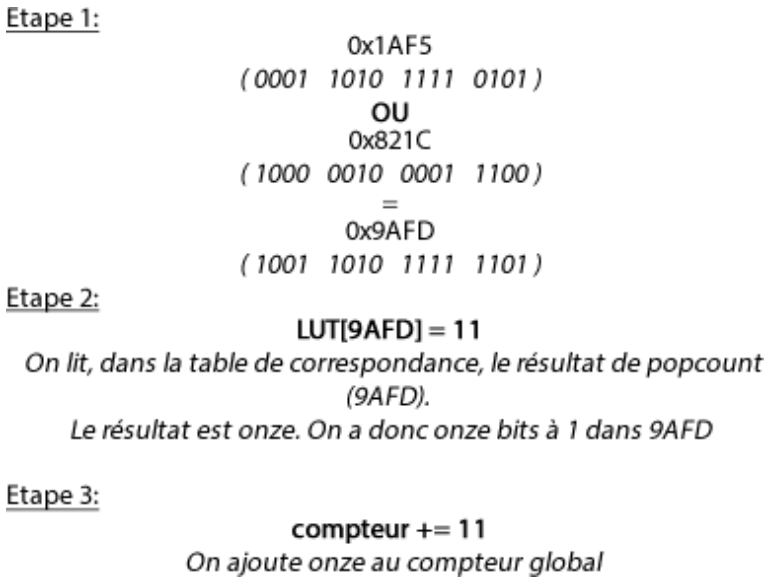


Figure 4 – *Exemple d'utilisation de la LUT*

4 Algorithme avec LUT et vectorisation du "OU"

La table de correspondance a été introduite pour accélérer le temps de traitement en évitant de (re)calculer le "builtin popcount" à chaque itération. Nous pouvons aller plus loin en vectorisant le code. Seule l'instruction OU s'y prête puisque l'incrémentation du compteur n'est pas vectorisable. Nous allons ici nous aider du compilateur et de sa fonctionnalité de vectorisation automatique.

Pour cela nous avons besoin d'isoler le OU et l'incrémentation dans deux blocs différents. Cela a pour effet de diviser l'algorithme en 2 boucles indépendantes.

Nous effectuons, dans un premier temps, un OU sur chaque segment de 16 bits en parcourant entièrement les mots A et B de 4Ko. Une boucle est dédiée à cette procédure. Le résultat des OU sur chaque segment est stocké dans un tableau. Ce tableau fait donc 4Ko et sert d'intermédiaire avec la seconde partie de l'algorithme.

Dans un second temps nous récupérons le résultat du popcount stocké préalablement dans la LUT, comme expliqué dans l'algorithme précédent. Une boucle est également utilisée dans cette seconde partie. A chaque tour de boucle nous récupérons 16 bits supplémentaires du tableau résultant de l'étape 1 et nous déterminons le nombre de bits à 1 dans chaque segment de 16 bits.

Dans cette version de l'algorithme le compilateur est capable de repérer notre première boucle OU et de la vectoriser. Il exécute les OU sur des segments de 128 bits plutôt que sur des segments de 16 bits.

```
1  virtual void perform() {
2      size_t i;
3
4      result = 0;
5
6      uint16_t * __restrict al_a = (uint16_t * __restrict) __builtin_assume_aligned(a, 16);
7      uint16_t * __restrict al_b = (uint16_t * __restrict) __builtin_assume_aligned(b, 16);
8      uint16_t * __restrict al_comp = (uint16_t * __restrict) ↩
           __builtin_assume_aligned(comp, 16);
9
10     // vectorizable
```

```
11      for(i = 0; i < LUT_ITERATIONS; i++) {
12          al_comp[i] = al_a[i] | al_b[i];
13      }
14
15      // not vectorizable
16      for(i = 0; i < LUT_ITERATIONS; i++) {
17          result += lut[al_comp[i]];
18      }
19  }
```

5 Algorithme avec LUT et vectorisation du "OU" et de l'addition

A l'étape précédente, le compilateur n'était pas en mesure de vectoriser la seconde boucle pour deux raisons :

- Elle comporte un accès en mémoire selon un index variable qui n'est pas vectorisable.
- Le compteur est incrémenté, ce qui suppose de connaître sa valeur antérieure et empêche ainsi toute parallélisation.

On ne peut (pour l'instant), rien faire contre le premier point qui reste la base de la LUT. Mais on peut essayer de transformer l'incrémentation en une série d'additions vectorisables.

Pour cela, on commence par casser de nouveau la deuxième boucle en deux autres boucles : la première stockant les valeurs lues de la LUT dans un tableau, la seconde étant chargée d'additionner toutes les valeurs ainsi stockées.

Pour permettre la vectorisation des additions, il nous faut deux opérandes indépendantes. On stockera donc la moitié des valeurs obtenues par la LUT dans un tableau, et la seconde moitié dans un autre.

L'addition consiste alors en une répétition du schéma suivant (Figure 5) :

- Une première boucle parcourt la première moitié des tableaux A et B, ajoutant à chaque itération deux résultats et les stockant dans A'.
- Une deuxième boucle parcourt la deuxième moitié des tableaux A et B, ajoutant à chaque itération deux résultats et les stockant dans B'.

Ce schéma est répété plusieurs fois, divisant ainsi le nombre d'opérandes par deux à chaque répétition. De plus, et c'est là le but recherché, en ajoutant des données issues de tableaux distincts, la vectorisation est possible.

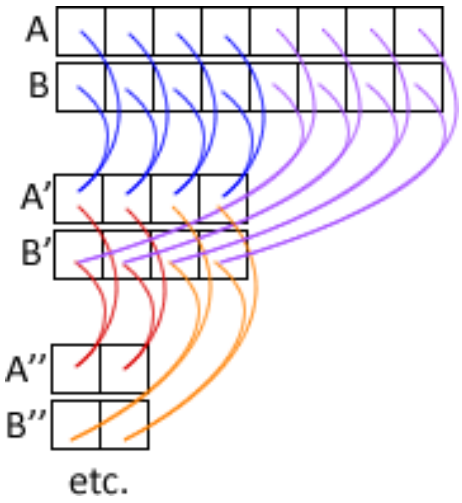


Figure 5 – *Gestion des additions vectorisable*

Avec cette approche, on vectorise l'addition et on gagne ainsi du temps d'exécution. Selon nos mesures, le gain se limite toutefois à 0,10 µs sur la machine de test. En effet, si nous parallélisons les additions, de nombreuses structures de boucle font leur apparition et viennent alourdir le code. De plus, comme on va le voir, la majeure partie du temps d'exécution est de toute façon occupée par l'accès mémoire (la LUT).

```
virtual void perform() {
    size_t i;
    size_t j;

    result = 0;

    uint16_t * __restrict al_a = (uint16_t * __restrict) __builtin_assume_aligned(a, 16);
    uint16_t * __restrict al_b = (uint16_t * __restrict) __builtin_assume_aligned(b, 16);
    uint16_t * __restrict al_comp = (uint16_t * __restrict) ↩
            __builtin_assume_aligned(comp, 16);
    uint16_t * __restrict al_nbA = (uint16_t * __restrict) __builtin_assume_aligned(nbA, 16);
    uint16_t * __restrict al_nbB = (uint16_t * __restrict) __builtin_assume_aligned(nbB, 16);
    uint16_t * __restrict al_nbC = (uint16_t * __restrict) __builtin_assume_aligned(nbC, 16);
    uint16_t * __restrict al_nbD = (uint16_t * __restrict) __builtin_assume_aligned(nbD, 16);

    // vectorizable
    for(i = 0; i < LUT_ITERATIONS; i++) {
        al_comp[i] = al_a[i] | al_b[i];
    }

    // not vectorizable
    for(i = 0, j = 0; i < LUT_ITERATIONS; i+=2, j++) {
        al_nbA[j] = lut[al_comp[i]];
        al_nbB[j] = lut[al_comp[i+1]];
    }

    //Vectorizable
    #define INNER(FA,FB,TA,TB,SZE) for(i = 0; i < SZE / 2; i++) TA[i] = FA[i] + ↩
            FB[i];for(i = SZE / 2; i < SZE; i++) TB[i-SZE / 2] = FA[i] + FB[i];
    INNER(al_nbA,al_nbB,al_nbC,al_nbD,1024);
    INNER(al_nbC,al_nbD,al_nbA,al_nbB,512);
    INNER(al_nbA,al_nbB,al_nbC,al_nbD,256);
    INNER(al_nbC,al_nbD,al_nbA,al_nbB,128);
    INNER(al_nbA,al_nbB,al_nbC,al_nbD,64);
    INNER(al_nbC,al_nbD,al_nbA,al_nbB,32);
    INNER(al_nbA,al_nbB,al_nbC,al_nbD,16);

    // not vectorizable
    for(i = 0; i < 8; i++) {
        result += al_nbC[i] + al_nbD[i];
    }
}
```

6 Algorithme exploitant le jeu d'insctructions SSE 4.1

Par curiosité, nous avons voulu observer ce qui était le plus coûteux dans l'algorithme précédent. A l'issue de ce test, il apparaît qu'une microseconde est dédiée à l'accès à la lookup table tandis que quelques dixièmes de microsecondes sont consacrée au "ou logique" et à l'addition. Malgré le gain conséquent offert par la LUT, elle demeure la partie la plus lourde de l'algorithme. Comme

en témoigne l'article [1], le jeu d'instructions SSE promet des gains sensibles de performance, d'abord parcequ'il gère de plus grands registres. Bien que cet aspect soit déjà utilisé lors de la compilation automatique, il peut être intéressant d'utiliser les instructions avec plus de précision que ne peut le faire le compilateur en l'absence d'informations.

Quelques recherches nous ont permis de découvrir une autre approche [WWW1], orientée bas niveau, et reposant sur des instructions machines spécifiques : "PSHUFB" et "PSADW". Un nouvel algorithme a donc pu être écrit en utilisant essentiellement des fonctions compilateurs intrinsèques (fonctions C équivalentes à des instructions machines), de manière à rester dans la philosophie bas niveau. Étudions cette solution particulière en détails et examinons en quoi le recours aux instructions précédemment citées ainsi qu'au jeu d'instructions SSE4 qui les propose est nécessaire.

La présence du jeu d'instructions SSE 4.1 veut également dire que nous avons à disposition des registres de 128 bits. Nous traiterons donc 128 bits par 128 bits, vectorisant ainsi le traitement de nos blocs de données élémentaires plus petits. Pour des raisons de simplicité, représentons 16 de ces 128 bits dans un premier temps.

— Deux registres de 128 bits sont initialisés avec les données appropriées.
— Un "ou logique" est appliqué à ces deux opérandes à l'aide de l'instruction appropriée appliquée à des registres de cette taille. Il s'agit donc désormais de compter la population dans le registre résultat.
— On divise nos bits en deux groupes (L pour "low" et H pour "high" ci-dessous) : le premier est construit en appliquant un masque, le second en appliquant ce même masque après décalage (Figure 6).

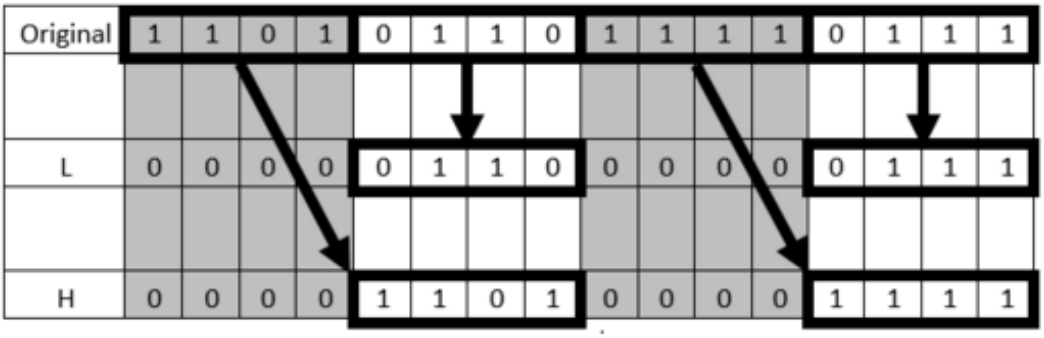


Figure 6 – *Division des bits à compter en deux groupes*

— On va maintenant compter la population. Pour cela, on dispose de l'instruction "PSHUFB" qui agit telle une lookup table : elle considère chaque octet comme l'index d'un tableau de 16 cellules de 8 bits (Figure 7). Ce tableau est stocké dans un registre de 128 bits. Or, à l'issue de la manipulation précédente, nous avons justement des octets qui peuvent prendre des valeurs de 0 à 15 (0b00000000 à 0b00001111). On peut donc utiliser un registre en guise de lookup table qui associe à un nombre son poids de Hamming. On applique donc "shuffle" ("PSHUFB") à L et H.
— Dans un deuxième temps, on peut ajouter Nb(L) et Nb(H) (pour "nombre de bits à 1" dans L et H) sans problème : les zéros font en sorte que l'on n'a pas à se soucier d'une retenue (elle peut intervenir mais ne débordera pas sur l'octet suivant avant un moment). Le résultat est stocké dans un accumulateur que l'on qualifiera de local. En l'état, nous avons donc à l'issue de cette première itération la population de mots de 128 bits sous la forme d'octets à ajouter ultérieurement.
— Comme nous l'avons dit, les zéros nous protègent des retenues qui pourraient avoir un effet de bord sur nos compteurs. Il y en a assez pour répéter l'opération décrite précédemment (sur 128 autres bits) plusieurs fois avant de vraiment se soucier d'une retenue débordant des octets faisant office d'accumulateurs. Dans le pire des cas, on

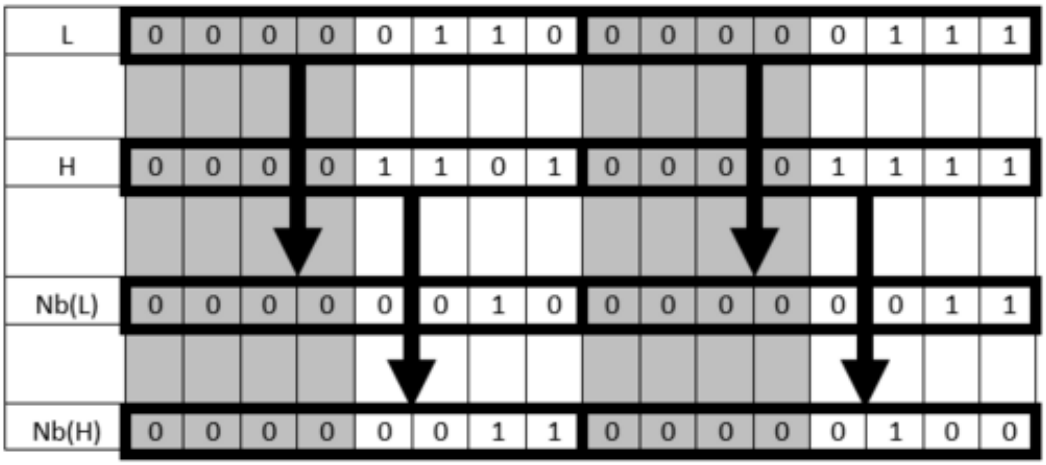


Figure 7 – *Illustration de l'instruction "PSHUFB"*

ajoute en effet 8 à chaque itération, et on dispose d'accumulateurs de 8 bits pouvant stocker une valeur allant jusqu'à 255. Ainsi, dans le pire des cas, on peut avoir 31 répétitions de l'approche ci-dessus avant une retenue potentiellement problématique.

— Dès lors qu'on atteint les 31 répétitions, il nous faut donc une procédure pour vider l'accumulateur local avant de continuer. A ce moment, on a un accumulateur local qui ressemble à la Figure 8.

A15	A14	A13	A12	A11	A10	A09	A08	A07	A06	A05	A04	A03	A02	A01	A00

Figure 8 – *Forme du registre SSE*

Chaque case représente un octet, c'est à dire un compteur partiel de la population observée jusqu'à présent. On voudrait les ajouter entre eux et les stocker ailleurs, dans un accumulateur que l'on qualifiera de global.

— Pour cela, nous allons utiliser l'instruction "PSADW" qui opère justement sur les octets constituant deux registres de 128 bits. Cette unique instruction calcule alors deux sommes : la première est la somme des valeurs absolues des différences entre les 8 premiers octets des deux registres, la seconde concerne les 8 octets suivant. On obtient un registre de 128 bits divisé en deux et dont chaque partie contient le résultat de l'opération décrite (Figure 9).

A15	A14	A13	A12	A11	A10	A09	A08	A07	A06	A05	A04	A03	A02	A01	A00

B15	B14	B13	B12	B11	B10	B09	B08	B07	B06	B05	B04	B03	B02	B01	B00

ABS(A08-B08) + ABS(A09-B09) + ABS(A10-B10) + ABS(A11-B11) + ABS(A12-B12) + ABS(A13-B13) + ABS(A14-B14) + ABS(A15-B15)	ABS(A00-B00) + ABS(A01-B01) + ABS(A02-B02) + ABS(A03-B03) + ABS(A04-B04) + ABS(A05-B05) + ABS(A06-B06) + ABS(A07-B07)

Figure 9 – *Illustration de l'instruction "PSADW"*

Dans notre cas, on veut simplement sommer les octets de notre accumulateur local, on réalisera donc l'instruction ci-dessus avec une des deux opérandes nulle (B dans l'illustration). On peut répéter cette procédure toutes les 31 itérations (ou dès lors qu'on a parcouru la totalité du mot) : on somme les octets de l'accumulateur local et on ajoute le résultat à l'accumulateur global. On peut alors poursuivre avec un accumulateur local nul.

— En fin de comptage, on a presque directement le résultat dans notre accumulateur global. Il suffit de sommer les deux parties de 64 bits qui composent ce registre pour l'obtenir.

L'utilisation du jeu d'instructions SSE4 est essentielle à l'implémentation de l'approche ci dessus puisqu'elle offre les instructions machines « PSADW » et « PSHUFB ». Elle s'est traduite par un gain sensible (de l'ordre d'un facteur 3) en termes de temps d'exécution : on passe alors d'un record à 1.24 µs à un record à 0.45 µs. Cela s'explique par une utilisation plus intelligente des registres, une réduction du nombre d'accès à la mémoire vive puisque la pseudo LUT est en registre, et une disparition des tests générés par le compilateur en vectorisation automatique.

```
virtual void perform() {
        size_t bytes_cnt = 0;
        size_t index = 0;
        size_t inner_ite = 0;

        // Counting is done in two loops
        // The main one is made to handle carries when the inner loop reaches the counter's limit
        // The inner loop counts SSE_MAXITE_BEFORE_CARRYING * 128 bits before carrying (unless there isn't enough data)
        // One counter for each of them

        __m128i global_acc = _mm_setzero_si128();
        while(bytes_cnt < DATA_LENGTH) {
            __m128i local_acc = _mm_setzero_si128();
            // Handles 128 bits per iteration, ie 16 bytes
            // Maximum SSE_MAXITE_BEFORE_CARRYING before carrying
            for(inner_ite = 0 ; inner_ite < SSE_MAXITE_BEFORE_CARRYING && bytes_cnt < DATA_LENGTH; inner_ite++, index++, bytes_cnt+= 16) {
                const __m128i comp = _mm_or_si128(a[index], b[index]);
                // We split the values in comp into two groups : high and low
                // low :  0xabcdef... => 0x 0 b 0 d 0 f
                // high : 0xabcdef... => 0x 0 a 0 c 0 e
                const __m128i comp_low = _mm_and_si128(comp, mask_low);
                const __m128i comp_high = _mm_and_si128(_mm_srli_epi16(comp, 4), mask_low);
                // The 0 are room for carries

                // Shuffling is equivalent to a 4bits => 4bits LUT
                const __m128i nb_low = _mm_shuffle_epi8(lut, comp_low);
                const __m128i nb_high = _mm_shuffle_epi8(lut, comp_high);

                // Adding the low and high counters
                local_acc = _mm_add_epi8(local_acc, nb_low);
                local_acc = _mm_add_epi8(local_acc, nb_high);
            }
            // We've reached the limit, we have to copy the local_acc to the global_acc
            // local_acc is like 0x a15 a14 a13 a12 a11 a10 a9 a8 a7 a6 a5 a4 a3 a2 a1 a0
            // mm_sad_epu8 computes 0x (a15+a14+a13+a12+a11+a10+a9+a8) (a7+a6+a5+a4+a3+a2+a1+a0)
            // So global_acc is like 0x AAAA BBBB (two accumulators inside)
            global_acc = _mm_add_epi64(global_acc, _mm_sad_epu8(local_acc, _mm_setzero_si128()));
        }

        uint64_t global_acc_low = _mm_cvtsi128_si64(global_acc);
        uint64_t global_acc_high = _mm_cvtsi128_si64(_mm_srli_si128(global_acc, 8));

        result = global_acc_low + global_acc_high;
    }
```

« En revanche », ce jeu d'instructions offre également une nouvelle instruction machine, et non des moindres : « POPCNT ». Comme son nom le laisse penser, cette instruction propose le calcul machine du poids de Hamming sur des mots de 64 bits. Et autant dire qu'il est difficile de rivaliser contre une instruction machine qui résout le problème en quelques cycles d'horloge. Ainsi, dans les mêmes conditions de test, le code utilisé par le compilateur lors de l'appel à la fonction native de comptage de bits change et se résume désormais à l'utilisation de cette instruction. C'est pourquoi le temps obtenu avec l'algorithme dit "naïf" descend à 0.30 µs.

On pourrait poursuivre cette course à la performance en vectorisant sur des registres plus grands, par exemple en ayant recours au jeu d'instructions AVX capable de manipuler des registres de 512 bits. On pourrait alors espérer un gain de l'ordre d'un facteur 4 par rapport à l'approche utilisant les instructions SSE, et ainsi battre le record qui reste pour l'instant détenu par l'algorithme utilisant l'instruction machine ; à condition que le comptage natif s'effectue toujours sur 64 bits avec le jeu d'instructions AVX à disposition, et que les instructions « PSADW » et « PSHUFB » aient des équivalents opérant sur 512 bits.

7 Bilan des contraintes

Chaque algorithme présenté ci-avant a ses propres contraintes quant à la taille des blocs élémentaires qu'il traitera parallèlement. Notons que ces contraintes sont ici données pour les algorithmes "en l'état". On l'a vu, tous peuvent être modifiés pour prendre en charge un nombre non contraint de bits. En effet, il est parfaitement envisageable d'utiliser les approches décrites autant de fois que possible (le quotient de la division entière de la taille totale par la taille contrainte des blocs) puis l'approche basique plus souple pour les bits restants (le reste dans cette même division).

Algorithme	Taille des données
Naïf (une boucle, builtin popcount)	Peu importe
Avec LUT (séparé en 2)	Multiple de 16 bits en l'état
Avec LUT (séparé en 3)	Multiple de 16 bits en l'état
Intrinsèques SSE	Multiple de 128 bits en l'état

3 Instrumentalisation et résultats

1 Instrumentalisation en C++

De manière à faciliter la comparaison des différents algorithmes et à les structurer, nous avons utilisé l'approche ci-dessous.

- Tous les algorithmes sont représentés par une classe propre, héritant de la classe Test. La classe mère Test définit une interface que doivent respecter les tests : implémenter les fonctions "setOperands", "perform" et "getResults".
  - L'initialisation de l'environnement de l'algorithme (calcul d'une look-up table par exemple) est réalisée dans le constructeur,
  - La transformation éventuelle des opérandes et leur stockage est fait dans "setOperands",
  - Le résultat est obtenu à la demande avec "getResult"

  Ainsi, on peut chronométrer uniquement l'exécution de "perform", qui constitue le cœur de chaque algorithme.
- En outre, une classe Tester permet d'instancier des objets à même de gérer les différents tests connus. Ils ont trois fonctionnalités principales : chronométrer l'exécution d'un algorithme donné (instance d'une classe fille de Test), afficher les statistiques des algorithmes chronométrés, et en vérifier les résultats.

La mesure du temps d'exécution d'un algorithme est paramétrée de deux façons : le nombre d'échantillons N, et le nombre d'itérations par échantillon M. On répète donc chaque algorithme M * N fois au total. Les statistiques (minimum, maximum, moyenne, et la plus parlante - la médiane), sont observées à l'échelle de l'échantillon.

- Pour un même algorithme, le temps d'exécution de plusieurs échantillons sera mesuré et stocké. Pour chaque échantillon un jeu de données différent est utilisé. Quelque soit l'algorithme, le premier échantillon aura toujours le même jeu de données A, le deuxième aura toujours le même jeu de données B, et ainsi de suite. On utilise donc, pour chaque échantillon, la méthode "setOperands" du Test et on stocke le résultat à la fin avec le "getResult" pour un usage ultérieur.
- Dans chaque échantillon, l'algorithme est répété M fois sans changer les données, simplement pour obtenir un temps d'exécution plus significatif et donc moins sensible aux variations de l'environnement d'exécution. On chronomètre donc M exécutions de "perform", ce qui nous donne le temps pour cet échantillon et pour l'algorithme étudié.

La deuxième fonction de Tester consiste à déterminer et afficher des statistiques pour les différents algorithmes déjà traités. Sont alors calculées, pour chaque algorithme :

— Le temps moyen d'exécution parmi les différents échantillons d'un même algorithme
— Le temps médian d'exécution parmi les différents échantillons d'un même algorithme
— Le temps minimum d'exécution parmi les différents échantillons d'un même algorithme
— Le temps maximum d'exécution parmi les différents échantillons d'un même algorithme
— Le temps total d'exécution de l'ensemble des échantillons d'un même algorithme

Si les temps sont stockés pour M exécutions, les temps affichés sont donnés pour une unique exécution (hormis le temps total). La dernière fonction de Tester est la vérification des résultats retournés par les algorithmes testés. Pour alléger ce processus, seule une partie réglable des échantillons est concernée. Par exemple, on peut s'assurer que, pour chaque algorithme, les 5 premiers échantillons retournent bien le même résultat. Dans le cas contraire, une alerte s'affiche pour mettre en évidence cette incohérence. On utilise pour cela les valeurs stockées à l'étape précédente.

La sortie est affichée en Figure 1.

```
Results used as reference (first 5 iterations):
        24568
        24648
        24501
        24571
        24561
Statistics are computed over 10 samples for each test.
For every sample, the algorithm was executed 100000 times.
Results are given per iteration (ie to OR and then COUNT 4096 bytes).

For test #1:
        Min    (in µs): 0.306760
        Max    (in µs): 0.354960
        Mean   (in µs): 0.330052
        Median (in µs): 0.336670
        Total  (in µs): 3.300520
For test #2:
        Min    (in µs): 1.392730
        Max    (in µs): 1.679630
        Mean   (in µs): 1.516045
        Median (in µs): 1.521280
        Total  (in µs): 15.160450
For test #3:
        Min    (in µs): 1.333640
        Max    (in µs): 1.371360
        Mean   (in µs): 1.351019
        Median (in µs): 1.346970
        Total  (in µs): 13.510190
For test #4:
        Min    (in µs): 0.488590
        Max    (in µs): 0.512340
        Mean   (in µs): 0.496162
        Median (in µs): 0.494760
        Total  (in µs): 4.961620
```

Figure 1 – *Résultats donnés par l'application*

2 Résultats

Nous avons retenu le temps médian obtenu sur 10 échantillons d'un million de répétitions de l'algorithme, ramené au temps pour une exécution. Chaque exécution de l'algorithme effectue un "ou logique" et un comptage sur 4 KB.

Algorithme	Naïf (une boucle, builtin popcount)	Avec LUT (séparé en 2)	Avec LUT (séparé en 3)	Intrinsèques SSE
Sans information particulière	1.669 µs	2.082 µs	2.925 µs	
Vectorisation activée (gc -O2 -ftree-vectorize)	1.645 µs	1.340 µs	1.248 µs	
Vectorisation activée et SSE4.1 disponible (gcc -O2 -ftree-vectorize -msse41)	0.30 µs	1.33 µs	1.24 µs	0.45 µs

4 Application : appariement de deux images

1 Principe de l'appariement

L'une des situations possibles en analyse d'image consiste à apparier deux images. On s'intéresse ici à un cas particulier : on cherche à rapprocher une petite image X avec une image plus grande Y, et ainsi localiser des probables apparitions de motifs semblables à X dans Y.

Par la suite, on appellera "imagette" l'image X et "image de référence" l'image Y. De plus, on se limitera à la comparaison d'images en niveaux de gris (un seul canal).

L'approche consiste alors à parcourir Y avec X, de pixel en pixel, et comparer la portion de Y avec X. On comparera les deux opérandes sous la forme de vecteurs. On peut alors calculer leur similarité sous la forme d'une distance, par exemple en comptant les pixels identiques. Dans les faits, on observe plusieurs phénomènes et on a recourt à différentes mesures introduites dans [5], adaptées à diverses situations. Si l'on considère que l'on parcourt bit à bit les deux vecteurs, on peut compter le nombre d'occurrences de certains phénomènes. Illustrons d'abord avec une figure, nous l'expliquerons juste après.

Imagette \ Image de référence	0	1
0	n_{00}	n_{01}
1	n_{10}	n_{11}

Figure 1 – *Calcul des n_{uv}*

Ce tableau présente ce que nous appellerons par la suite les n. n_{00} représente par exemple le nombre de fois où l'imagette et le fragment de l'image de référence possédaient toutes deux un bit à 0 au même endroit. De la même manière, n_{01} indique que l'imagette possède un bit à 0 là où le bit du fragment de l'image de référence était à 1. Ces nombres permettent donc d'avoir des statistiques de similarités entre les deux images. Ces mesures sont très anciennes et sont utilisées dans de nombreuses formules de calcul de distance entre les images (voir la figure Figure 2, extraite de [4]).

Sur cette figure, on voit par exemple la méthode intuitive de comparaison d'image, à savoir le décompte du nombre d'occurrences de pixels identiques (Hamming). Dans notre cas, on va s'intéressera vraisemblablement à un calcul de distance plus robuste tel que la méthode de Yule and Kendall.

Measure	$S(X,Y)$	Range
Inner Product (IP)	n_{11}	$[0,+\infty[$
Jaccard and Needham ($Jaccard$)	$\frac{n_{11}}{n_{11}+n_{10}+n_{01}}$	$[0,1]$
Dice ($DICE$)	$\frac{n_{11}}{2n_{11}+n_{10}+n_{01}}$	$[0,0.5]$
Russel-Rao ($RUSS$)	$\frac{n_{11}}{n}$	$[0,1]$
Kulzinsky (KUL)	$\frac{n_{11}}{n_{10}+n_{01}}$	$[0,+\infty[$
Hamming ($Hamming$)	$n_{11}+n_{00}$	$[0,+\infty[$
Sokal and Michner (SM)	$\frac{n_{11}+n_{00}}{n}$	$[0,1]$
Rogers and Tanimoto (RT)	$\frac{n_{11}+n_{00}}{n_{11}+n_{00}+2(n_{10}+n_{01})}$	$[0,1]$
Correlation ($CORR$)	$\frac{n_{11}n_{00}-n_{10}n_{01}}{\sqrt{n_{1x}n_{0x}n_{x1}n_{x0}}}$	$[-1,1]$
Yule and Kendall ($Yule$)	$\frac{n_{11}n_{00}-n_{10}n_{01}}{n_{11}n_{00}+n_{10}n_{01}}$	$[-1,1]$

Figure 2 – *Mesures classiques de similarités en Analyse Dynamique d'Image*

2 Méthode d'appariement de deux images

2.1 Démarche classique

De manière à calculer l'une des distances ci-dessus, il nous faut pouvoir calculer les n_{uv}. Pour cela, la méthode classique consiste à comparer les deux vecteurs représentant l'imagette et le fragment de l'image de référence avec plusieurs opérateurs logiques et compter la population dans les résultats. Un OR permettra d'obtenir $n_{01}+n_{10}+n_{11}$ Un AND permettra d'obtenir n_{11} Un XOR permettra d'obtenir $n_{10}+n_{01}$ Connaissant la population totale, on a ainsi un simple système d'équations à résoudre.

On a déjà un gain de performance notable avec le recours au fast bit counting face à un comptage de population plus traditionnel. Toutefois, il existe une démarche plus pertinente.

2.2 Recours à une démarche moins coûteuse par l'image intégrale

Une autre approche est proposée et résumée ci-dessous, ainsi que sur la Figure 3.

2.2.1 Encodage

Pour procéder au traitement, il nous faut obtenir des fragments de Y, de la même taille que l'imagette. Pour cela, on effectue un "encodage" de l'image de référence.

Après avoir défini une taille de bloc $h \times w$, il est possible, pour chaque pixel de Y, de créer un vecteur regroupant les h ou w bits suivant ce pixel, suivant que l'on encode horizontalement ou verticalement. On stocke le vecteur ainsi créé à la position correspondante dans une matrice de taille légèrement inférieure à Y sur une dimension (comme on le montrera l'exemple ci-après).

Prenons l'exemple d'une imagette X de 2 pixels par 2 pixels et d'une image Y de 3 pixels par 3 pixels. Que l'on encode dans un sens ou dans un autre, on aura des vecteurs correspondant à la concaténation de 2 pixels. Choisissons d'encoder verticalement. Par exemple, sur la Figure 4, on créerait un vecteur de longueur 2 (représenté par la colonne rouge), qui contiendrait la valeur

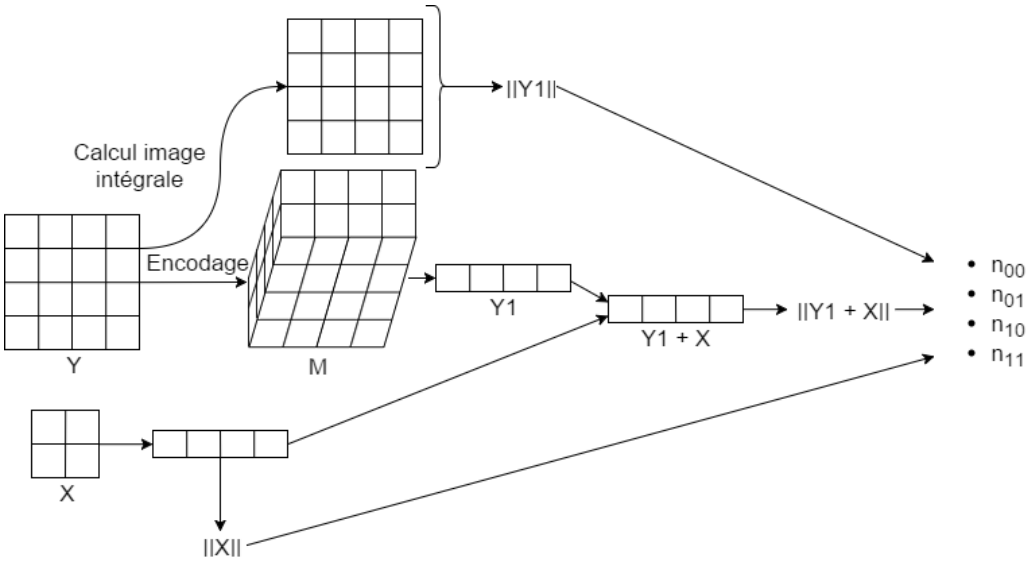


Figure 3 – *Démarche d'appariement proposée*

des deux pixels désignés par les flèches rouges, et stocké aux coordonnées $(0,0)$. On remarque que la matrice des vecteurs sera plus petite que Y, le cas limite étant donné avec la colonne bleue, dans le coin opposé, qui doit dans notre cas prendre en compte les pixels désignés par la flèche bleue. En effet, un vecteur en (3,3) devrait prendre en compte un pixel en (4,3) dans Y, qui n'est pas défini.

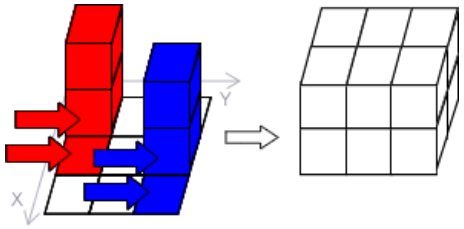


Figure 4 – *Encodage lors de l'appariement*

Ensuite, de manière à obtenir un vecteur représentant un fragment de Y, il suffira donc de lire respectivement les w ou h vecteurs l'un après l'autre, à partir de la position adaptée. C'est particulièrement simple puisque la représentation en mémoire de ces vecteurs est déjà linéaire.

Toujours dans notre exemple, pour lire un fragment de deux pixels par deux pixels, il suffirait de lire les deux vecteurs désignés par les flèches dans la Figure 5. Nous avions encodé verticalement. Si on stocke les vecteurs ligne par ligne les uns à la suite des autres, les 4 pixels que nous cherchions sont donc écrits de manière contiguë en mémoire.

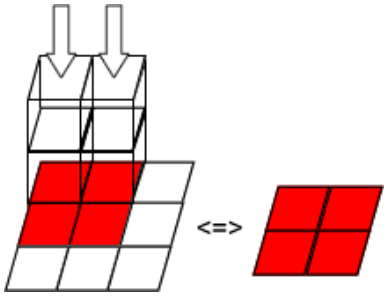


Figure 5 – *Lecture de la forme encodée lors de l'appariement*

En somme, on passe d'une image 2D $s \times t$ à une matrice de vecteurs de taille inférieure (selon

la taille des vecteurs). Cette transformation permet d'extraire facilement et à moindre coût un fragment de l'image référence.

2.2.2 Image intégrale

La force de cette méthode repose sur le recours à l'image intégrale (de Y dans notre cas). On désigne ainsi une matrice de la même taille que Y qui ne contient non plus des pixels (ici un niveau de gris) mais un entier. Pour l'élément aux coordonnées i et j de la matrice, cet entier représente la population rencontrée dans l'image d'origine pour le fragment $0 <= x <= i$ et $0 <= y <= j$. Un exemple est proposé en Figure 6.

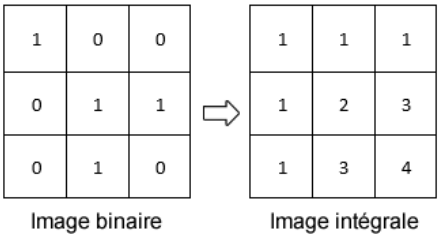
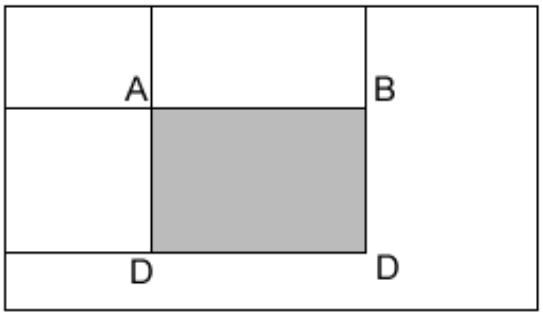


Figure 6 – *Exemple d'image intégrale avec une simple image binaire*

C'est un calcul que l'on peut réaliser une fois pour toute et conserver cette matrice. Elle permet de déterminer la population de n'importe quelle fragment rectangulaire de l'image de référence selon la formule mathématique et le schéma suivants :



$$\sum_{\substack{x_A<x'\leq x_C \\ y_A<y'\leq y_C}} i(x',y') = ii(A) + ii(C) - ii(B) - ii(D).$$

Figure 7 – *Calcul de population d'une fragment d'image par l'image intégrale*

De cette manière, le coup pour déterminer la population de tout fragment de Y se résume à quatre accès mémoire et trois opérations algébriques. On est bien loin du coup d'un comptage à chaque itération, le gros du calcul étant fait une unique fois.

2.2.3 Démarche globale

En utilisant les méthodes présentées ci-dessus, il est possible d'obtenir un fragment Y_i de Y sous la forme de vecteur ainsi que sa population de manière efficace. Par ailleurs, il est possible de calculer la population de X une fois pour toute au départ de l'algorithme.

Il est alors temps de réutiliser l'opération mise en oeuvre dans les parties précédentes de ce document : calculer $\|Y_i + X\|$, sachant que nous disposons de Y_i et X. A l'issue de cette opération nous disposons de :

— $\|Y_i\|$
— $\|X\|$
— $\|Y_i + X\|$ Et la taille des vecteurs en question (notée n)

L'article présentant cette méthode précise qu'il est alors possible d'obtenir les n_{uv} que nous cherchions :

— $n_{00} = n - \|Y_i + X\|$
— $n_{01} = \|Y_i + X\| - \|X\|$
— $n_{10} = \|Y_i + X\| - \|Yi\|$
— $n_{11} = \|Y_i\| + \|X\| - \|Y_i + X\|$

On peut répéter cette opération (seulement le calcul de $\|Y_i' + X\|$ et des n'_{uv}, $\|Y_i'\|$ selon la même image intégrale, le reste demeure) pour chaque position de X sur Y, calculer les distances et réaliser les déductions appropriées quant à la proximité des fragments de Y avec l'imagette.

2.2.4 Intérêt

Pourquoi s'embêter avec cette démarche compliquée ? Tout simplement parce qu'elle est bien plus efficace et exploite pleinement les possibilités du fast bit counting abordé dans les chapitres précédents.

— On abandonne une approche naïve qui consiste à parcourir les images bit par bit et les comparer afin d'incrémenter le n_{uv} correspondant à chacun des couples de bits rencontrés.
— De plus, on passe de trois opérations logiques entre les deux vecteurs à une unique opération
— Mais surtout, on remplace trois comptages de population, aussi rapides qu'ils soient, par un unique comptage de population et de quelques opérations algébriques.

Si certains mécanismes permettent d'alléger le processus de comparaison, le comptage de population reste une des briques de base de celui-ci. L'utilisation des travaux précédents permet de réduire encore davantage les temps de calcul, confirmant ainsi par l'exemple que le comptage de population n'est pas qu'un exercice théorique sans usage pratique.

Conclusion

Nous avons pu étudier différentes manières de résoudre notre problème de "OU suivi d'un POPCOUNT", la dernière partie du problème étant la plus problématique. Lorsqu'un problème revient aussi souvent que celui-ci, les fondeurs s'évertuent à créer des instructions au niveau machine pour une efficacité maximale. C'est par exemple le cas en cryptographie : certains processeurs disposent d'instructions de chiffrement natives alors qu'il était avant nécessaire de trouver des algorithmes performants. Notre exemple a suivi la même évolution : certaines instructions réalisent désormais le popcount de manière matérielle, en exploitant complètement les registres, rendant dérisoires nos tentatives d'améliorations. On ne peut de toute façon pas dépasser les performances d'une instruction de niveau matériel.

Cependant, dans le cas où les instructions machines n'existent pas, avoir recours au SIMD permet d'atteindre des performances intéressantes à condition d'avoir un algorithme vectorisable. Ainsi, le compilateur peut automatiquement procéder à la vectorisation du code et obtenir ce gain de performances sans demander un effort de programmation supplémentaire. La contrainte principale est de fournir une attention particulière à la possibilité de vectorisation lors de la conception de l'algorithme pour qu'il soit vectorisable.

Au début de cette étude, nous nous demandions si Java pouvait rivaliser avec un programme natif. Comme on l'a vu, cela n'est pas le cas, de part le poids de la machine virtuelle mais surtout à cause d'une gestion limitée des registres. Au maximum, seuls 64 bits sont exploités par la JVM quand un programme natif peut exploiter la totalité des registres SSE (128 bits) ou AVX (512 bits).

Cette promesse suscite évidemment l'intérêt de nombreux scientifiques, même dans le web. En effet, lors de nos recherches, nous avons par exemple découvert un papier sur la vectorisation en javascript [2]. L'article indique que la librairie est capable d'atteindre des performances très intéressantes. Néanmoins deux contraintes sont présentes : il faut coder d'une manière particulière, et javascript est un langage non typé. En effet, s'il est facile de vectoriser le calcul numérique, qu'en est-il des manipulations de chaînes de caractères par exemple ?

Webographie

[WWW1] Wojciech Muła. *SSSE3 : fast popcount*. 4 mar. 2015. URL : http://wm.ite.pl/articles/sse-popcount.html.

Annotation: Ce document présente différents algorithmes utilisant les fonctions intrinsèques du compilateur pour le comptage de bits.

[WWW2] Wikipédia. *Taxinomie de Flynn*. 21 nov. 2015. URL : https://fr.wikipedia.org/wiki/Taxinomie_de_Flynn.

Annotation: Cette page présente la taxinomie de Flynn.

[WWW3] Wikipédia. *Vectorisation (informatique)*. 18 sept. 2015. URL : https://fr.wikipedia.org/wiki/Vectorisation_(informatique).

Annotation: Cet article présente le principe de la vectorisation et les critères qui la permettent.

Bibliographie

[1] K. Marton A. Suciu P. Cobarzan. « The Never Ending Problem of Counting Bits Efficiently ». In : ().

Annotation: Ce papier revient sur les méthodes de comptage de bits et leur efficacité.

[2] Mario Dehesa-Azuara. « Automatic Vectorization of JavaScript using SIMD.js ». In : (29 avr. 2015). url : http://www.contrib.andrew.cmu.edu/~mdehesaa/main.pdf.

Annotation: Ce papier présente les travaux réalisés sur SIMD.js : un framework pour la vectorisation du langage javascript.

[3] Hwancheol Jeong. « Performance of SSE and AVX Instruction Sets ». In : (5 nov. 2012).

Annotation: Ce papier présente les performances offertes par la vectorisation et compare les résultats obtenus avec du code C++, de l'assembleur inline SSE et de l'assembleur inline AVX.

[4] M. Iwata M. Delalandre K. Kise. *Fast and Optimal Binary Template Matching - Application to Manga Copyright Protection*. 2014. url : http://mathieu.delalandre.free.fr/publications/DAS2014p2s.pdf.

Annotation: Poster présentant notamment les distances que l'on peut calculer dans le cadre de l'appariement de deux images.

[5] M. Iwata M. Delalandre T.A. Pham. « Near-duplicate Manga Image Detection with Binary Template Matching and Selection ». In : (2016). url : http://mathieu.delalandre.free.fr/publications/Draft2015.pdf.

Annotation: Cet article présente une utilisation pratique de l'algorithme de bit counting réalisé avec la vectorisation à l'esprit.

Projet ASR

Évaluation de performance en vectorisation de programmes natifs vs. systèmes virtuels

Résumé

Cette étude consiste à observer les performances offertes par la vectorisation de programmes natifs par rapport aux programmes exécutés sur des systèmes virtuels. Elle sera réalisée en implémentant différents algorithmes réalisant la même opération : une opération logique suivie d'un comptage de bits. Ce sera l'occasion d'évaluer les outils donnés aux développeurs pour utiliser la vectorisation au mieux.

Mots-clés

Architecture système, Vectorisation, Comptage de bit

Abstract

This study aims at analyzing the performances enabled by vectorizing native programs compared to the execution of programs on virtual machines. It will be performed by implementing various algorithms doing the same thing : a logical operation followed by a bit counting operation. It will be a way to assess the existing tools given to developers to help them vectorize their programs efficiently.

Keywords

System architecture, Vectorization, Fast Bit Counting

Tuteurs académiques
Mathieu Delalandre

Étudiants
Jonathan Le Bonzec (DI5)
Jean-François Béchu (DI5)
Lorry Moreau (DI5)