



Exploiting Modern Hardware for High-Dimensional Nearest Neighbor Search

Fabien André

► To cite this version:

Fabien André. Exploiting Modern Hardware for High-Dimensional Nearest Neighbor Search. Computer Science [cs]. INSA Rennes, 2016. English. ⟨NNT : ⟩. ⟨tel-01658922⟩

HAL Id: tel-01658922

<https://hal.science/tel-01658922v1>

Submitted on 8 Dec 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



HAL Authorization

Exploiting Modern Hardware for High-Dimensional Nearest Neighbor Search

Fabien André

PhD thesis defended on
November 25, 2016
at INSA Rennes

Thesis Advisors

Anne-Marie Kermarrec
Research Director, Inria

Nicolas Le Scouarnec
Senior Scientist, Technicolor

Thesis Committee

Achour Mostefaoui
President
Professor, University of Nantes

Gaël Thomas
Referee
Professor, Telecom SudParis

Peter Triantafillou
Referee
Professor, University of Glasgow

This work is distributed under a Creative Commons Attribution 4.0 International License (CC BY 4.0). For more information, see <https://creativecommons.org/licenses/by/4.0/>.



Sources are available at <https://github.com/Xion345/phd-thesis>.

Some of the techniques presented in this thesis may be covered by patents owned by Technicolor R&D France, Thomson Licensing and/or other parties. No rights to any party's patents are implied by the distribution of this thesis.

This document was compiled on October 30, 2017 by Xe_ΛTeX version 0.99996 using L^AT_EX 2_ε version 2017/01/01 and PGF version 3.0.1a.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisors, Anne-Marie Kermarrec and Nicolas Le Scouarnec, for their outstanding guidance. I would like to particularly thank them for their assistance through the process of defining my thesis topic, their original ideas, their technical help, and their precious advice on scientific writing. I have a feeling that I have discovered a new world during this thesis – the one of research – and I owe this discovery to Nicolas and Anne-Marie. On a more personal level, I would also like to thank them for their human qualities. The completion of a PhD thesis inevitably involves moments of doubt and hesitation and I was fortunate to be able to count on their support during these moments.

I also thank all other members of the jury, Achour Mostefaoui, Peter Triantafillou and Gaël Thomas. I am especially grateful to Peter Triantafillou and Gaël Thomas for reviewing the manuscript and providing feedback.

This thesis would not have been possible without the support of Technicolor which gave me the opportunity to fully devote myself to my research activities during these three years. I also thank Inria for providing access to computing resources without which many experiments of this thesis could not have been done. More specifically, I was fortunate to have access to Grid’5000, a highly flexible large-scale platform which enables complex experiments. I cannot stress enough how important these platforms are for the research in computer science, and especially for systems research.

I also have a thought for my colleagues and fellow PhD students at Inria and Technicolor without whom these three years would not have been the same. Thank you for creating an intellectually stimulating environment, thank you for your friendship, and thank you for the crazy discussions at the coffee break – I will not forget them. I owe a special thank you to Clémentine, my fellow PhD student at Technicolor, for sharing with me the difficulties of doing a PhD.

Lastly, these acknowledgements would not be complete without a word of thanks for my family who gave me much-needed emotional support over these three years. They offered a sympathetic ear when I needed, and I am convinced I could not have come this far without their continuous support.

La présente thèse comporte un résumé de onze pages rédigé en français (« Résumé en Français ») dont les pages sont numérotées en chiffres romains. Le reste de la thèse est rédigé en anglais, et les pages sont numérotées en chiffres arabes. Le résumé en Français ne contient aucune information qui n'est pas également présente dans le reste de la thèse.

The present thesis comprises a summary of eleven pages written in French (« Résumé en Français ») the pages of which are numbered with Roman numerals. The remainder of the thesis is written in English, and the pages are numbered with Arabic numerals. The summary in French contains no information that is not also present in the remainder of the thesis.

Contents

Contents	7
List of Figures	10
List of Tables	12
Résumé en Français	i
Introduction	i
Quantification Produit	i
Analyse de Performance	iii
PQ Fast Scan	v
Quick ADC	vii
Derived Quantizers	viii
Conclusion et Perspectives	x
1 Introduction	1
1.1 The Big Data Trend	1
1.2 Applications of Nearest Neighbor Search	3
1.3 Classification of Nearest Neighbor Search Problems	5
1.4 The Curse of Dimensionality	5
1.5 Product Quantization	6
1.6 Thesis Outline	7
2 State of the Art	11
2.1 Space and Data Partitioning Trees	12
2.1.1 KD-trees	12
2.1.2 Bag of Features and Vocabulary Trees	13
2.1.3 FLANN Library	15
2.2 Locality Sensitive Hashing (LSH)	16
2.2.1 Euclidean Distance LSH (E2LSH)	16
2.2.2 Virtual Rehashing Schemes	18
2.3 Product Quantization	19
2.3.1 Vector Encoding	20
2.3.2 Inverted Indexes	22
2.3.3 ANN Search	24
2.3.4 Product Quantization Derivatives	27
3 Performance Analysis	31
3.1 Impact of Memory Accesses	31

3.2	Issues with SIMD Implementation	35
3.3	Lessons Learned	37
4	PQ Fast Scan	39
4.1	Motivation	39
4.2	Presentation	40
4.2.1	Overview	40
4.2.2	Code Grouping	42
4.2.3	Minimum Tables	44
4.2.4	Quantization of Distances	45
4.2.5	Lookups in Small tables	46
4.3	Evaluation	47
4.3.1	Experimental Setup	47
4.3.2	Distribution of Response Times	48
4.3.3	Performance Counters	49
4.3.4	Impact of init and r Parameters	49
4.3.5	Impact of Distance Quantization	50
4.3.6	Impact of the Size of Inverted Lists	51
4.3.7	Large Scale Experiment	52
4.3.8	Impact of CPU Architecture	52
4.4	Discussion	53
4.4.1	Compatibility with Inverted Indexes	53
4.4.2	Applicability to Product Quantization Derivatives	54
5	Quick ADC	57
5.1	Motivation	57
5.2	Presentation	58
5.2.1	Overview	58
5.2.2	SIMD Distance Computations	59
5.3	Evaluation	61
5.3.1	Experimental Setup	61
5.3.2	Exhaustive Search in SIFT1M	62
5.3.3	Non-exhaustive Search in SIFT1M	64
5.3.4	Non-exhaustive Search in GIST1M	65
5.3.5	Non-exhaustive Search in Deep1M	66
5.3.6	Non-exhaustive Search in SIFT1B	67
5.4	Discussion	68
5.4.1	Compatibility with Inverted Indexes	68
5.4.2	Applicability to Product Quantization Derivatives	68
6	Derived Quantizers	69
6.1	Motivation	69
6.2	Presentation	70
6.2.1	Overview	70
6.2.2	Training Derived Quantizers	71
6.2.3	ANN Search with Derived Quantizers	73
6.3	Evaluation	76
6.3.1	Experimental Setup	76
6.3.2	Small Datasets, 64-bit Codes	76
6.3.3	Small Dataset, 32-bit Codes	79

6.3.4	Large Dataset, 64-bit Codes	80
6.4	Discussion	81
6.4.1	Limitations	81
6.4.2	Comparison with Other Quantization Approaches	82
6.4.3	Comparison with ADC+R	82
7	Conclusion and Perspectives	85
7.1	Summary of the Thesis	85
7.1.1	Context	85
7.1.2	Contributions	86
7.2	Perspectives	87
7.2.1	Application to Product Quantization Derivatives	87
7.2.2	Exploitation of Future Hardware	89
7.2.3	Generalization to Other Algorithms	90
	Bibliography	93

List of Figures

1	Comparaison de ADC Scan et PQ Fast Scan (25M vectors)	vi
2	Impact de la taille des listes inversées sur PQ Fast Scan	vii
3	Recall et temps de réponse (SIFT1M, codes 64 bits)	x
1.1	Overview of Content Based Image Retrieval (CBIR) systems	2
2.1	ANN search system based on an inverted index	14
2.2	ANN search in KD-trees and Vocabulary trees	15
2.3	Geometrical interpretation of LSH functions	16
2.4	Asymmetric Distance Computation (ADC)	26
3.1	Scan times (25M vectors) and Instructions Per Cycle (IPC)	34
3.2	Scan performance counters (per scanned code)	35
3.3	SIMD vertical add	35
3.4	SIMD gather	36
3.5	Standard layout and transposed layout	36
4.1	Overview of PQ Fast Scan	41
4.2	SIMD in-register shuffle (<code>pshufb</code>)	42
4.3	Small tables building process	42
4.4	Portions of the first distance table	43
4.5	Code grouping	43
4.6	Centroid indexes assignement	44
4.7	Minimum tables	45
4.8	Selection of quantization bounds	46
4.9	Use of small tables to compute lower bounds	46
4.10	Distribution of response times, performance counters and IPC	49
4.11	Impact of init Parameter (all inverted lists)	50
4.12	Impact of r Parameter (all inverted lists, init=0.5%)	50
4.13	Pruning power using quantization only (all inverted lists)	51
4.14	Impact of inverted list size (init=0.5%, r=100)	51
4.15	Experiments on other CPU architectures (see Table 4.4)	53
5.1	Transposed block	59
5.2	SIMD Lookup-add ($j = 0$)	60
5.3	SIMD 4-bit Right Shift ($j = 0$)	61
5.4	ADC and QADC response time and recall (PQ, SIFT1M, Exhaustive search)	62
5.5	ADC and QADC response time and recall (OPQ, SIFT1M, Exhaustive search)	63

6.1	Derived quantizer training process	71
6.2	Capped buckets data structure	75
6.3	Impact of r_2 on recall and response time (SIFT1M, 64-bit codes)	76
6.4	Recall and response time (SIFT1M, 64-bit codes)	77
6.5	Impact of r_2 on recall and response time (SIFT1M, 32-bit codes)	78
6.6	Recall and response time (SIFT1M, 32-bit codes)	79
6.7	Impact of r_2 on recall and response time (SIFT100M, 64-bit codes) . . .	80
6.8	Recall and response time (SIFT100M, 64-bit codes)	81

List of Tables

1	Temps de réponse et précision (codes 64 bits, SIFT1M)	iii
2	Comparaison ADC - Quick ADC (SIFT1M, 64 bit)	viii
3	Comparaison ADC - Quick ADC (SIFT1B, 64 bit)	ix
2.1	Memory use of different LSH approaches (SIFT1M)	19
3.1	Properties of different types of memory (Nehalem-Haswell)	32
3.2	ANN search speed and accuracy (SIFT1M)	33
4.1	Properties of table lookup techniques	41
4.2	Size of inverted lists used for experiments	47
4.3	Response time distribution	48
4.4	Systems	52
5.1	Systems	62
5.2	Datasets	62
5.3	Non-exhaustive search, SIFT1M, 64 bit	64
5.4	Non-exhaustive search, GIST1M, 128 bit	65
5.5	Non-exhaustive search, Deep1M, 64 bit	66
5.6	Non-exhaustive search, Deep1M, 128 bit	66
5.7	Non-exhaustive search, SIFT1B, 64 bit	67
5.8	Non-exhaustive search, SIFT1B, 128 bits	67

Résumé en Français

Introduction

Cette thèse s'intéresse à la recherche de plus proche voisin en haute dimensionnalité et à large échelle. Ce problème revêt une importance particulière dans le contexte actuel d'utilisation massive de réseaux sociaux en ligne. L'utilisation de ces réseaux sociaux par des millions d'utilisateurs a permis l'accumulation de jeux de données très volumineux, une tendance connue sous le nom de *big data*. Ces jeux de données comprennent non seulement des données textuelles (messages, discussions) mais aussi multimédia (images, sons, vidéos).

La recherche de plus proche voisin dans les espaces de haute dimensionnalité est un problème clé dans le domaine des bases de données multimédia. Les objets multimédia (images, sons, vidéos) peuvent être représentés par des vecteurs caractéristiques en haute dimensionnalité. Rechercher deux objets multimédias similaires revient alors à rechercher deux objets multimédia ayant des vecteurs caractéristiques similaires. Cependant, la recherche de plus proche voisin est un problème difficile, en particulier à large échelle. En haute dimensionnalité, la tristement célèbre « malédiction de la dimensionnalité » rend impossible la recherche de solutions exactes. Les travaux de recherche actuels s'attachent donc à trouver des solutions approchées à la recherche de plus proche voisin.

La product quantization est une des solutions approchées de recherche de plus proche voisin les plus efficaces. Son principal avantage est qu'elle permet de stocker des bases de données volumineuses entièrement en RAM, grâce à une technique de compression des vecteurs de haute dimensionnalité en codes compacts. Ceci permet alors de répondre à des requêtes de recherche de plus proche voisin sans accéder à un disque dur ou à un SSD. Ainsi, la product quantization offre des temps de réponse faibles. Dans cette thèse, nous proposons différentes solutions pour réduire davantage les temps de réponse offerts par la product quantization en exploitant les fonctionnalités des processeurs modernes.

Quantification Produit

Quantifieur vectoriel Pour compresser les vecteurs de haute dimensionnalité en codes compacts, la quantification produit s'appuie sur des quantifieurs vectoriels. Un quantifieur vectoriel est une fonction q qui associe un vecteur $x \in \mathbb{R}^d$ à un vecteur $c_i \in \mathbb{R}^d$ appartenant à un ensemble prédéfini de vecteurs $\mathcal{C} = \{c_0, \dots, c_{k-1}\}$. Les vecteurs c_i sont appelés centroides, l'ensemble de centroides $\mathcal{C}$ est le dictionnaire.

Un quantifieur optimal associe un vecteur x à son centroïde le plus proche : $q(x) = \arg \min_{c_i \in \mathcal{C}} \|x - c_i\|$.

Quantification produit La quantification produit, ou *Product Quantization* (PQ), divise un vecteur $x \in \mathbb{R}^d$ en m sous-vecteurs $x = (x^0, \dots, x^{m-1})$. Chaque sous-vecteur x^j est ensuite quantifié en utilisant un quantifieur q^j distinct. Chaque quantifieur q^j dispose d'un dictionnaire $\mathcal{C}^j$ distinct. Un quantifieur produit pq quantifie un vecteur x comme suit :

$$\begin{aligned} \text{pq}(x) &= (q^0(x^0), \dots, q^{m-1}(x^{m-1})) \\ &= (\mathcal{C}^0[i_0], \dots, \mathcal{C}^{m-1}[i_{m-1}]). \end{aligned}$$

Un quantifieur produit peut être utilisé pour encoder un vecteur x en un code compact défini par la concaténation des indices des m centroïdes retournés par les m quantifieurs q^j : $\text{code}(x) = (i_0, \dots, i_{m-1})$. Un code compact $\text{code}(x)$ utilise $m \cdot b$ bits de mémoire, avec $b = \log_2(k)$ où k est le nombre de centroïdes de chaque quantifieur q^j . Un code compact $\text{code}(x)$ utilise beaucoup moins de mémoire qu'un vecteur de haute dimensionnalité x . Ainsi, un vecteur en 128 dimensions peut être représenté par un code de 64 bits ($m = 8, b = 8, m \times b = 64$), alors qu'il occupe $d \cdot 32 = 4096$ bits stocké sous forme d'un tableau de flottants.

Recherche de plus proche voisin Lorsque la quantification produit est utilisée, les vecteurs de haute-dimensionnalité de la base de données sont compressés en codes compacts. Les codes compacts sont ensuite stockés dans un tableau contigu en RAM. Pour trouver le plus proche voisin d'un vecteur requête y , il est nécessaire de calculer la distance entre le vecteur y et chacun des codes stockés en RAM. Pour ce faire, la quantification produit s'appuie sur une procédure nommée Asymmetric Distance Computation (ADC), ou calcul de distance asymétrique, qui permet de calculer la distance entre un vecteur de haute dimensionnalité y et n'importe quel code c de la base de données. Plus précisément, la recherche de plus proche voisin se déroule en deux étapes, nommées Tables et Scan :

- *Tables*. Le vecteur y est divisé en m sous-vecteurs $y = (y^0, \dots, y^{m-1})$. On construit ensuite m tables de correspondance $\{D_0, \dots, D_{m-1}\}$. Chaque table de correspondance D_j contient la distance entre le sous-vecteur y^j et chaque centroïde du dictionnaire $\mathcal{C}^j$.
- *Scan*. Ensuite, on calcule la distance entre le vecteur y et chaque code c de la base de données grâce à la procédure ADC, comme suit :

$$\text{adc}(y, c) = \sum_{j=0}^{m-1} D^j[c[j]] \quad (1)$$

Cette thèse s'attache à améliorer les performances de la procédure ADC, et en particulier de l'étape Scan, car celle-ci consomme la majorité des cycles CPU.

Index inversés La quantification produit est généralement associée à des systèmes d'index inversés. Ces systèmes divisent la base de données en un ensemble de listes inversées. Pour répondre à une requête, les listes inversées les plus appropriées sont sélectionnées. Ces listes sont ensuite scannées en utilisant la procédure ADC. Les

TAB. 1 : Temps de réponse et précision (codes 64 bits, SIFT1M)

$m \times b$	Tables size	Cache	Recall@100	Tables time	Scan time
16×4	1 KiB	L1	83.1%	0.001 ms	6 ms
8×8	8 KiB	L1	92.2%	0.011 ms	2.6 ms
4×16	1 MiB	L3	96.5%	0.82 ms	7.9 ms

index inversés permettent de réduire les temps de réponse en évitant de scanner la totalité de la base de données. Ils permettent également d’augmenter la précision de la recherche de plus proche voisin, grâce à un système connu sous le nom d’encodage de résidus.

Quantification produit optimisée La quantification produit divise un vecteur x en sous-vecteurs x^j et encode chaque sous-vecteur en un indice de b bits, indépendamment de la quantité d’information contenue dans chaque sous-vecteur. Ce procédé donne des résultats sous-optimaux lorsque les sous-vecteurs ne contiennent pas la même quantité d’information ou que certaines dimensions sont corrélées avec d’autres. La quantification produit optimisée, ou *Optimized Product Quantization* (OPQ), remédie à ce problème en multipliant les vecteurs x par une matrice ortho-normale $R \in \mathbb{R}^{d \times d}$. Cette matrice permet une rotation et une permutation arbitraire des composantes du vecteur x , et est apprise de façon à minimiser l’erreur de quantification. Un quantifieur produit optimisé opq quantifie un vecteur x comme suit :

$$\text{opq}(x) = \text{pq}(Rx), \text{ tel que } R^T R = I,$$

où pq est quantifieur produit. Un quantifieur produit optimisé peut encoder des vecteurs en codes compacts de la même manière qu’un quantifieur produit.

Analyse de Performance

Analyse des opérations Afin d’améliorer la performance de la procédure ADC, et en particulier de l’étape Scan, on commence par identifier les goulots d’étranglement qui la limitent. Lors de l’étape Scan, chaque calcul de distance asymétrique nécessite :

- m accès mémoire pour charger les indices des centroides $c[j]$ (mem1)
- m accès mémoire pour charger les valeurs $D_j[.]$ depuis les tables de correspondance (mem2)
- m additions ($\sum_{j=0}^{m-1}$)

Parmi ces opérations, les accès mémoires sont les plus coûteuses. En effet, un accès mémoire prend entre 4 et 40 cycles CPU en fonction du niveau de cache atteint, alors qu’une addition ne prend qu’un seul cycle. Ainsi, la performance de PQ Fast Scan est principalement limitée par le grand nombre d’accès cache qu’elle effectue.

Accès mémoire Les accès mémoire mem1 atteignent toujours le cache le plus rapide (cache L1), grâce aux *hardware prefetchers* inclus dans les processeurs modernes. On accède aux codes c *séquentiellement*: premier code, puis second code etc. ce qui

permet aux *hardware prefetchers* de précharger les codes c dans le cache L1. En revanche, le niveau de cache atteint par les accès mem2 dépend du paramètre b du quantifieur produit utilisé. Un quantifieur produit est complètement défini par deux paramètres m , le nombre de quantifieurs, et b le nombre de bits par quantifieur (qui détermine le nombre $k = 2^b$ de centroides par quantifier).

Compromis précision-vitesse La précision d'un quantifieur produit dépend principalement du produit $m \cdot b$. Ce produit définit l'occupation mémoire de chaque code, et donc l'occupation mémoire de la base de données. Ainsi un quantifieur produisant des codes de 128 bits ($m \cdot b = 128$) sera plus précis qu'un quantifieur produisant des codes de 64 bits ($m \cdot b = 64$). Le quantifieur produisant des codes de 128 bits engendrera cependant un doublement de la taille de la base de données, puisque chaque code occupera 128 bits au lieu de 64 bits. Pour limiter l'occupation mémoire, le produit $m \cdot b$ est souvent fixé à 64. Pour un produit $m \cdot b$ fixé (par exemple $m \cdot b = 64$), les paramètres m et b peuvent varier. Plus b est grand (et donc plus m est petit), meilleure est la précision du quantifieur, et donc meilleure est la précision de la recherche de plus proche voisin. Ainsi, un quantifieur produit 4×16 ($m = 4, b = 16$) et un quantifieur produit 8×8 ($m = 8, b = 8$) produisent tous deux des codes de 64 bits, mais le quantifieur 4×16 offre une meilleure précision. Pour mesurer la précision des techniques de recherche de plus proche voisin approchées, on utilise le *Recall@R*. Le Recall@R est défini comme la proportion de requêtes pour lesquelles le plus proche voisin *exact* du vecteur requête se trouve parmi les R plus proches voisins renvoyés par la technique de recherche approchée. En général, on paramètre les techniques de recherche approchées pour qu'elles retournent R=100 voisins, donc on utilise le Recall@100 comme mesure de précision (Table 1). On constate alors qu'un quantifieur produit 4×16 ($b = 16$) offre un meilleur Recall@100 qu'un quantifieur 8×8 ($b = 8$). Cependant, plus b est grand, plus les tables de correspondance D_j sont volumineuses, et doivent alors être stockées dans des niveaux de cache plus lents. Pour $b = 16$, les tables doivent être stockées dans le cache L3 (latence de 40 cycles), au lieu du cache L1 (latence de 4-5 cycles). Ceci se traduit par un triplement du temps de réponse (Table 1). Pour $b = 8$ et $b = 4$, les tables sont stockées dans le cache L1. Le paramètre $b = 8$ offre alors un meilleur temps de réponse car il nécessite moins d'opérations ($m = 8$ contre $m = 16$). Le paramétrage $m \times b = 8 \times 8$ offre un bon compromis entre précision et temps de réponse, raison pour laquelle il est utilisé dans la quasi-totalité de la littérature sur la quantification produit. Cependant, même lorsqu'ils atteignent le cache L1 (niveau de cache le plus rapide), les accès mémoire restent un facteur limitant la performance de la procédure ADC.

SIMD Les instructions Single Instruction Multiple Data (SIMD) sont couramment utilisées pour améliorer la performance des algorithmes, en particulier lorsqu'ils réalisent un grand nombre d'opérations arithmétiques. Les instructions SIMD appliquent la même opération (par exemple une addition) à plusieurs données simultanément, pour produire plusieurs résultats simultanément. Pour se faire, les instructions SIMD opèrent sur des registres larges, en général 128 bits. Ces registres sont divisés en plusieurs *voies*, par exemple 4 voies flottantes (4 voies de 32 bits, soit 128 bits). Les instructions SIMD sont donc susceptibles d'être utilisées pour réduire le nombre de cycles CPUs dédiés aux m additions nécessaires pour chaque calcul de distance. Cependant, la structure de la procédure ADC empêche une utilisation efficace des instructions SIMD. En effet, les valeurs $D_j[c[j]]$ chargées depuis les tables de corres-

pondance D_j ne sont pas contiguës en mémoire, ce qui rend impossible le chargement d'un registre SIMD de 128 bits en un seul accès mémoire. Au contraire, les valeurs doivent être insérées une à une dans chacune des voies des registres SIMD, ce qui nécessite un grand nombre d'instructions. Au final, ceci annule le gain offert par l'utilisation d'additions SIMD.

En définitive, nous pouvons tirer trois conclusions de cette analyse de performance :

- Les accès mémoire sont le principal facteur limitant la performance de la procédure ADC, bien qu'ils atteignent le cache L1 (niveau de cache le plus rapide) dans la majorité des cas.
- Les quantifieurs produit utilisant des quantifieurs 16 bits ($b = 16$) offrent une meilleure précision que les quantifieurs produit utilisant des quantifieurs 8 bits ($b = 8$) mais ils engendrent un triplement du temps de réponse lors de la recherche de plus proche voisin. Ceci est dû au fait que les quantifieurs 16 bits nécessitent de stocker les tables de correspondance utilisées par la procédure ADC dans le cache L3, alors que ces tables peuvent être stockées dans le cache L1 lorsque des quantifieurs 8 bits sont utilisés.
- La structure de la procédure ADC empêche l'utilisation efficace des instructions SIMD. En effet, la procédure ADC récupère des valeurs dans le cache avant de les insérer une à une dans les registres SIMD. Ces opérations sont coûteuses et annulent le gain offert par les instructions SIMD.

PQ Fast Scan

Grâce au stockage des bases de données en RAM, la quantification produit permet de scanner un grand nombre de codes en peu de temps. Cependant, la procédure ADC reste consommatrice de temps CPU. En particulier, nous avons montré que la performance de la procédure ADC est principalement limitée par le grand nombre d'accès cache qu'elle occasionne. De plus, la structure de cette procédure empêche une utilisation efficace des instructions SIMD pour augmenter sa performance. Ces limitations appellent donc une modification de la procédure ADC.

Pour ces raisons, nous avons conçu PQ Fast Scan, une procédure de scan de listes de codes compacts novatrice. PQ Fast Scan offre une performance 4 à 6 fois supérieure à la procédure ADC conventionnelle, tout en retournant exactement les mêmes résultats. PQ Fast Scan est exclusivement compatible avec les quantifieurs produits 8×8 . Il ne s'agit pas en pratique d'une limitation forte, étant donné que ce type de quantifieur est utilisé dans presque la totalité des cas. L'idée maîtresse de PQ Fast Scan consiste à remplacer les accès au cache L1 par des permutation SIMD (instruction `pshufb`). Cette modification permet également une utilisation efficace des additions SIMD. Utiliser des permutations SIMD nécessite de stocker les tables de correspondance dans les registres SIMD. Le principal défi que nous avons eu à résoudre pour la conception de PQ Fast Scan est que les tables de correspondances D_j sont beaucoup plus volumineuses (1 KiB chacune) que les registres SIMD (128 bits chacun).

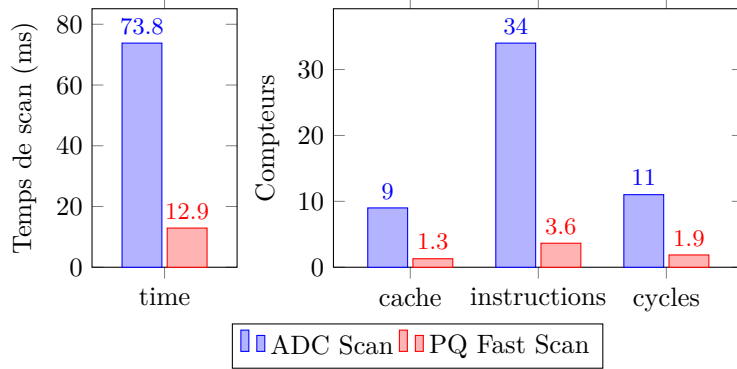


FIG. 1 : Comparaison de ADC Scan et PQ Fast Scan (25M vectors)

PQ Fast Scan dépasse cette difficulté en utilisant des *petites tables* dimensionnées pour tenir dans les registres SIMD à la place des tables de correspondance stockées en cache. Ces petites tables sont utilisées pour calculer des bornes basses sur les distances, sans accéder au cache L1. Par conséquent les calculs de bornes basses sont rapides. De plus, ils sont implémentés en utilisant des additions SIMD, ce qui améliore encore la performance. Nous utilisons les calculs de bornes basses pour élaguer les calculs de distances, qui nécessitent des accès au cache L1 et ne peuvent pas tirer profit des instructions SIMD. Ainsi, pour chaque code c dans la liste à scanner, on calcule la borne basse sur sa distance au vecteur requête y . Si la borne basse est supérieure à la distance entre le vecteur y et le plus proche voisin actuel, alors le code c est écarté et on passe au code suivant. Si la borne basse est inférieure ou égale à la distance entre le vecteur y et le plus proche voisin, on calcule la distance entre le code c et y pour déterminer si c peut devenir le plus proche voisin courant. Nos résultats expérimentaux sur des vecteurs SIFT montrent que les calculs de borne basses permettent d'élaguer plus de 95% des calculs de distance.

Chaque petite table S^j est calculée à partir de la table D^j correspondante, $j \in \{0, \dots, 7\}$. Pour construire les petites tables, nous combinons trois techniques : (1) le groupage des codes, (2) le calcul de tables de minimums et (3) la quantification des distances flottantes en entiers 8 bits. Les deux premières techniques, le groupage des codes et le calcul de tables de minimums, sont utilisées pour transformer les tables D_j de 256 flottants en tables de 16 flottants (256×32 bits $\rightarrow 16 \times 32$ bits). La troisième technique, la quantification des distance flottantes en entier 8 bits, est utilisée pour réduire chaque élément de 32 à 8 bits (16×32 bits $\rightarrow 16 \times 8$ bits).

Pour valider notre approche, on mesure la performance de PQ Fast Scan et de la procédure ADC Scan conventionnelle sur une liste de 25 millions de codes (Figure 1). PQ Fast Scan atteint un temps de scan de 5.72 fois plus faible qu'ADC Scan à 12.9 ms contre 73.8 ms. Grace au remplacement des accès cache par des permutations SIMD, PQ Fast Scan ne nécessite que 1.3 accès cache par code scanné contre 9 pour ADC Scan. Le nombre d'instructions nécessaires pour scanner un code passe de 34 pour ADC Scan à 3.6 pour PQ Fast Scan, à la fois grace à l'utilisation de permutations SIMD et d'additions SIMD. Au global, ceci se traduit par une baisse du nombre de cycles CPU par code scanné de 11 pour ADC Scan à 1.9 pour PQ Fast Scan.

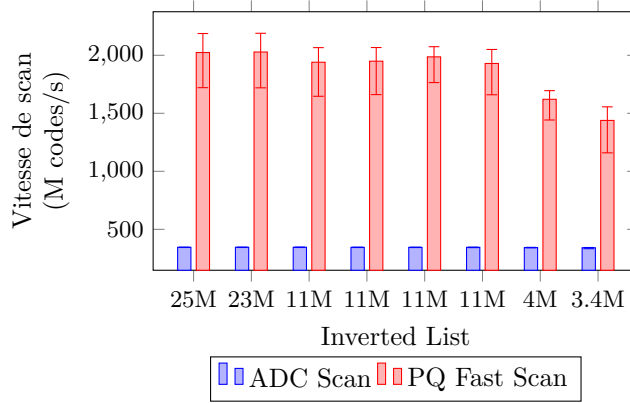


FIG. 2 : Impact de la taille des listes inversées sur PQ Fast Scan

PQ Fast Scan utilise trois techniques pour obtenir des petites tables qui tiennent dans les registres SIMD : (1) le groupage des codes, (2) le calcul de tables de minimums et (3) la quantification des distances flottantes en entiers 8 bits. Parmi ces trois techniques, le groupage des codes implique une réorganisation des listes de codes, et impose une taille minimale sur ces listes de codes. En effet, les codes sont répartis dans un nombre fixe de groupes, et la taille moyenne d'un groupe ne doit pas descendre en dessous d'un certain seuil, au risque d'impacter la performance. On peut observer cet effet en mesurant la vitesse de PQ Fast Scan sur 8 listes dont les tailles varient de 3.4 million de codes à 25 millions de codes (Figure 2). La taille de chaque liste est indiquée en abscisse. PQ Fast Scan a une vitesse constante pour les listes de 11 à 25 million de vecteurs. Cependant, la vitesse diminue fortement avec la taille de la liste à partir de 4 millions de vecteurs (Figure 2). Ainsi, PQ Fast Scan n'est intéressant que pour les listes dont la taille dépasse 2 à 3 millions de vecteurs. Ceci rend PQ Fast Scan incompatible avec les configurations d'index inversés les plus performants qui divisent la base de données en un grand nombre de listes inversées de petite taille (10000 vecteurs ou moins). En pratique, il est donc difficile de combiner l'accélération offerte par PQ Fast Scan et celle offerte par les index inversés.

Quick ADC

Nous avons montré que le principal inconvénient de PQ Fast Scan est qu'il n'est pas compatible avec les index inversés les plus performants. Pour cette raison, nous proposons Quick ADC, un autre procédure de scan rapide, mais pouvant être combinée avec n'importe quel type d'index inversé, y compris les plus performants. Comme PQ Fast Scan, Quick ADC, offre des performances 4 à 6 fois supérieures à la procédure ADC conventionnelle. Quick ADC s'appuie également sur un remplacement des accès cache par des permutations SIMD, et utilise des additions SIMD. Cependant, Quick ADC utilise des techniques différentes de PQ Fast Scan pour obtenir les petites tables stockées registres SIMD. Quick ADC s'appuie sur : (1) l'utilisation de quantifieurs 4 bits et (2) la quantification des distances flottantes en entiers 8 bits. L'utilisation de quantifieurs 4 bits permet d'obtenir des tables des correspondance D_j de $2^4 = 16$ flottants (16×32 bits). En quantifiant les flottants en entiers 8 bits, on obtient des tables de 16 entiers 8 bits (16×8 bits) pouvant être stockées dans les registres SIMD.

TAB. 2 : Comparaison ADC - Quick ADC (SIFT1M, 64 bit)

PQ ¹	ADC ²	Recall@100	Index	Tables	Scan	Total
Sans index inversé						
PQ	ADC	0.916	-	0.005	2.8	2.8
	QADC	0.826	-	0.001	0.38	0.38
		-9.8%		-80%	-86%	-86%
Avec index inversé (K=256, ma=24)						
PQ	ADC	0.949	0.008	0.18	0.3	0.48
	QADC	0.907	0.008	0.055	0.072	0.14
		-4.4%		-69%	-76%	-72%
OPQ	ADC	0.963	0.008	0.21	0.29	0.52
	QADC	0.949	0.008	0.089	0.073	0.17
		-1.5%		-59%	-75%	-67%

¹ PQ : Quantification Produit, OPQ : Quantification Produit Optimisée

² ADC : ADC (8×8), QADC : Quick ADC (16×4)

Le principal inconvénient des quantifieurs 4 bits est qu'ils entraînent une perte de précision en comparaison aux quantifieurs 8 bits. Nous montrons cependant que cette perte de précision est faible, voire négligeable, surtout lorsque Quick ADC est combiné avec des index inversés et la quantification produit optimisée. La quantification produit optimisée est un dérivé de la quantification produit offrant une meilleure précision et de plus en plus utilisé dans les publications récentes sur la recherche de plus proche voisin. Sur le jeu de données SIFT1M (1 million de vecteurs SIFT), Quick ADC est plus de 7 fois plus rapide (-86% sur le temps de réponse) que la procédure ADC conventionnelle (Table 2). Quick ADC entraîne cependant une baisse faible, mais non-négligeable de 9.8% du rappel. En combinant Quick ADC avec un index inversé, cette baisse est ramenée à 4.4%. On constate par ailleurs qu'utiliser un index inversé offre à la fois un meilleur temps de réponse (0.48 ms contre 2.8 ms) et un meilleur rappel (0.949 contre 0.916), avec ou sans Quick ADC. Pour cette raison, les index inversés sont utilisés dans la majorité des cas. Enfin, en combinant Quick ADC avec un index inversé et la quantification produit optimisée, la baisse de rappel devient négligeable (-1.5%). Quick ADC offre alors un gain de temps de réponse de 67%. Nous obtenons des résultats similaires sur d'autres types de vecteurs (vecteurs GIST etc.) lorsque l'on utilise OPQ. Sur les jeux de données volumineux, par exemple le jeu de données SIFT1B (1 milliard de vecteurs SIFT), Quick ADC offre également un gain en temps de réponse proche de 70% (Table 3). La perte de recall est cependant un peu plus importante, -7.3% sur le jeu de données SIFT1B. Quick ADC propose donc de troquer une perte de recall faible ou négligeable (-1.5% à -7.3%), pour un fort gain en temps de réponse (proche de 70%).

Derived Quantizers

Pour la plupart des cas d'utilisation actuels de la recherche de plus proche voisin, la quantification produit avec des quantifieurs 8 bits offre une précision suffisante.

TAB. 3 : Comparaison ADC - Quick ADC (SIFT1B, 64 bit)

PQ ¹	ADC ²	Recall@100	Index	Tables	Scan	Total
Avec index inversé (K=65536, ma=64)						
OPQ	ADC	0.806	0.52	0.51	4.2	5.2
	QADC	0.747	0.53	0.22	0.92	1.7
		-7.3%		-57%	-78%	-68%

¹ OPQ : Quantification Produit Optimisée

² ADC : ADC (8×8), QADC : Quick ADC (16×4)

À travers notre solution Quick ADC, nous avons montré qu'il était même possible d'utiliser des quantifieurs 4 bits. Pour améliorer la performance dans ces cas d'utilisation, nous avons proposé deux procédures de scan hautement optimisées, PQ Fast Scan et Quick ADC. Cependant, de nouveaux cas d'utilisations émergents nécessitent une meilleure précision. Par exemple, les descripteurs générés par des réseaux de neurones profonds sont de plus en plus populaires dans les applications multimédia. Ces vecteurs comprennent plusieurs milliers de dimensions, et sont donc plus difficiles à quantifier. Pour ces cas d'utilisation, il est intéressant d'utiliser des quantifieurs produits utilisant de quantifieurs 16 bits, étant donné qu'ils offrent une meilleure précision que les quantifieurs produit utilisant des quantifieurs 8 bits. Par ailleurs, l'utilisation de codes de 32 bits, à la place des codes de 64 bits usuels, soulève un intérêt grandissant. Les codes de 32 bits rendent l'utilisation de quantifieurs de 16 bits nécessaire parce que les quantifieurs 8 bits offrent une précision trop basse dans ce cas. Les quantifieurs 16 bits souffrent cependant d'un inconvénient majeur : ils entraînent un triplement du temps de réponse (Table 1), si bien qu'ils ne sont pas utilisés en pratique.

Pour résoudre ce problème, nous proposons une approche novatrice, les quantifieurs dérivés. Les quantifieurs dérivés rendent les quantifieurs 16 bits aussi rapides que les quantifieurs 8 bits, tout en conservant leur précision. L'idée principale de notre approche consiste à associer un quantifieur dérivé de 8 bits avec chaque quantifieur 16 bits utilisé dans le quantifieur produit. Les quantifieurs dérivés (8 bits) sont utilisés pour calculer des *tables de correspondance compactes*, stockées dans le cache L1. Ces tables de distances compactes sont utilisées pour calculer des distances *approchées* entre le vecteur requête et les codes de la liste à scanner. Ces calculs de distance approchés servent à sélectionner les $r2$ codes les plus proches du vecteur requête et construire ainsi un ensemble de candidats. Un calcul de distance précis est ensuite effectué pour chacun des $r2$ codes de l'ensemble de candidats. Cette évaluation précise de distance s'appuie sur des tables de correspondance calculées à partir des quantifieurs 16 bits. Étant donné qu'elles ont été calculées à partir de quantifieurs 16 bits, ces tables de distances sont volumineuses, et donc stockées dans le cache L3. En conséquence, les évaluations de distance précises sont lentes, car elles nécessitent d'accéder au cache L3. Cependant, ces évaluations précises de distances sont réalisées uniquement pour les $r2$ codes de l'ensemble de candidats précédemment généré. Au global, ceci permet d'obtenir des temps de réponse bas, très proches du temps de réponse obtenu avec des quantifieurs 8 bits.

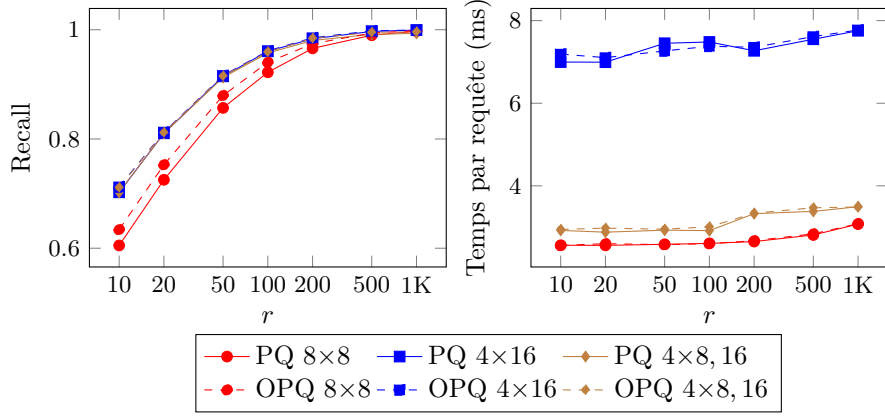


FIG. 3 : Recall et temps de réponse (SIFT1M, codes 64 bits)

Une solution naïve pour obtenir des quantifieurs 8 bits et des quantifieurs 16 bits consisterait à apprendre indépendamment ces deux types de quantifieurs. Chaque vecteur serait alors compressé en code compact deux fois : une première fois en utilisant les quantifieurs 8 bits, et une seconde fois en utilisant les quantifieurs 16 bits. Nous obtiendrions alors deux codes par vecteur, ce qui doublerait l'utilisation mémoire de la base de données. Un tel inconvénient serait rédhibitoire : au lieu d'utiliser des quantifieurs 16 bits pour améliorer la précision, on pourrait simplement utiliser des codes deux fois plus grands avec des quantifieurs 8 bits. Au lieu d'apprendre indépendamment les quantifieurs 8 bits et les quantifieurs 16 bits, on *dérive* le dictionnaire des quantifieurs 8 bits à partir du dictionnaire des quantifieurs 16 bits de tel sorte qu'ils puissent partager un même code. Les quantifieurs 16 bits sont utilisés pour encoder les vecteurs en code compacts et les quantifieurs 8 bits dérivés sont utilisés uniquement pour la recherche de plus proche voisin.

On évalue les quantifieurs dérivés sur le jeu de données SIFT1M (Figure 3). On observe que les quantifieurs 16 bits (4×16) offrent un meilleur recall que les quantifieurs 8 bits, que ce soit avec la quantification produit (PQ) ou la quantification produit optimisée (OPQ). Cependant les quantifieurs 16 bits engendrent un triplement du temps de réponse en comparaison aux quantifieurs 8 bits. Notre solution, les quantifieurs dérivés, permet de bénéficier du meilleur des deux mondes. En effet, les quantifieurs dérivés (notés $4 \times 8, 16$) offrent le même recall que les quantifieurs 16 bits, tout en offrant un temps de réponse très proche des quantifieurs 8 bits (Figure 3).

Conclusion et Perspectives

Dans ce résumé, nous avons présenté quatre contributions sur le problème de la recherche de plus proche voisin dans les espaces de haute dimensionnalité, et à large échelle. L'exploitation des bases de données multimédia repose largement sur la recherche de plus proche voisin. Ainsi, ce problème revêt une importance particulière dans le contexte actuel de collecte massive de données multimédia. Les contributions de cette thèse s'appuient sur la quantification produit, une des solutions actuelles de recherche de plus proche voisin les plus efficaces. La quantification produit compresse

les vecteurs de haute dimensionnalité en codes compacts, ce qui permet de stocker des bases de données volumineuses en RAM. Pour trouver le plus proche voisin d'un vecteur requête, la quantification produit calcule la distance entre ce vecteur requête et les codes compacts stockés en RAM. Pour ce faire, la quantification produit s'appuie sur une procédure de scan nommée Asymmetric Distance Computation (ADC), ou calcul de distance asymétrique. Cette procédure utilise des tables de correspondances stockées en cache pour les calculs de distances. Elle consomme un grand nombre de cycles CPU, et est le goulot d'étranglement des systèmes de recherche de plus proche voisin utilisant la quantification produit.

Cette thèse s'intéresse à l'amélioration de la performance de la procédure ADC. Notre première contribution consiste en une analyse des facteurs limitant la performance de la procédure ADC. Nous montrons que la performance de la procédure ADC est principalement limitée par le grand nombre d'accès cache qu'elle effectue. De plus, la structure de la procédure ADC empêche une implémentation efficace utilisant les instructions SIMD, couramment utilisées pour améliorer les performances. Suite à cette analyse, nous proposons deux procédures de scan optimisées : PQ Fast Scan et Quick ADC. Ces deux procédures s'appuient sur un remplacement des accès caches par des permutations SIMD. PQ Fast Scan n'entraîne aucune perte de précision, mais n'est pas compatible avec les index inversés, une méthode d'accélération de la quantification produit largement répandue. Quick ADC entraîne une faible perte de précision, mais est compatible avec les index inversés. Enfin, notre dernière contribution, nommée quantifieurs dérivés améliore la performance de la procédure ADC dans le cas où une très haute précision est nécessaire.

Ces contributions ouvrent des perspectives de recherche, à la fois à court terme, et à plus long terme. À court terme, on trouve l'adaptation de PQ Fast Scan et Quick ADC au jeu d'instructions SIMD AVX-512 qui sera introduit dans la prochaine génération de processeurs Intel Xeon (Skylake Purley, sortie prévue en 2017). Le jeu d'instructions AVX-512 offrira des instructions pour effectuer des permutations SIMD sur des tables de 512 bits (contre 128 bits actuellement). Ceci permettra à PQ Fast Scan d'offrir une meilleure compatibilité avec les index inversés, et à Quick ADC d'offrir une meilleure précision. En outre, AVX-512 permettra de traiter plus données par instruction, offrant ainsi des gains de performance. Parmi les perspectives à plus long terme, les techniques utilisées dans PQ Fast Scan pour réduire les tables de correspondance afin qu'elles tiennent dans des registres SIMD peuvent être adaptés pour accélérer d'autres algorithmes utilisant des tables de correspondances, ou certains algorithmes de base de données (requêtes topk etc.).

1 Introduction

Contents

1.1	The Big Data Trend	1
1.2	Applications of Nearest Neighbor Search	3
1.3	Classification of Nearest Neighbor Search Problems	5
1.4	The Curse of Dimensionality	5
1.5	Product Quantization	6
1.6	Thesis Outline	7

1.1 The Big Data Trend

It is indisputable that humanity is producing an increasingly important amount of digital data, which is collected, stored and processed by computer systems in large data centers. A combination of factors arguably explain this surge in the amount of data we produce. First, the wide availability and popularity of capture devices such as digital cameras and smartphones have enabled mass acquisition of digital data. Second, the development of fast internet access and online social networks has allowed corporations to collect the data produced by massive amounts of users. Thus, over 1.6 billion users communicate via the Facebook social network every month [18], over 80 million photos are uploaded every day to the Instagram social network [31] and over 400 hours of videos are uploaded every minute to the Youtube [59] streaming service. This trend of pervasive data production and collection, often referred to as *big data*, impacts many aspects of society: economy, marketing, politics etc. The term "big data" is no longer a jargon word: between June 2015 and June 2016, the newspaper *The New York Times* published 83 articles including this term. Science is no exception to this trend with the advent of data-intensive science [28]. For instance, the LHC particle accelerator generates about 30 petabytes of data each year [14].

These large datasets collected by corporations are highly valuable assets, that generate a lot of revenue. Amazon uses the purchase history of its millions of users to automatically generate product recommendations, which allowed it to increase its sales by 29% [19]. In the same vein, Youtube massive video database attracts millions of users, which in turn generates advertising revenue. Generating value requires such data to be processed, analyzed or searched. Thus, Amazon uses *machine learning* algorithms on the purchase history of its users to learn a product recommendation model. Similarly, Youtube indexes its videos by title and keywords so that

1. Introduction

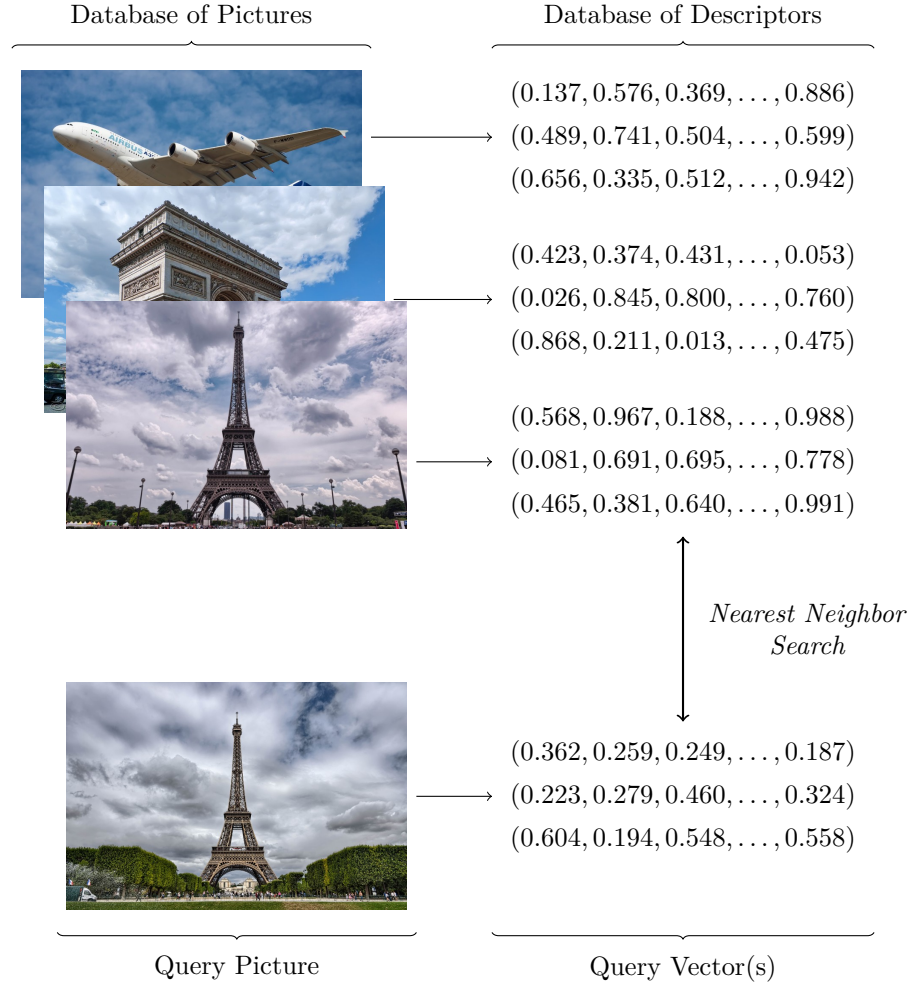


Photo Credits: (Top to bottom)

- "Farnborough Air Show - A380" by davidgsteadman, public domain
- "Arc de Triomphe de l'Étoile" by Joe deSousa, public domain
- "Eiffel Tower" by Joe deSousa, public domain
- "The Eiffel Tower, Paris" by Joe deSousa, public domain

Photos downloaded from Flickr

Figure 1.1: Overview of Content Based Image Retrieval (CBIR) systems

its database can be searched. Processing and extracting information from these large datasets is a challenging task, not only because of the high *volume* of data and the high *velocity* of acquisition, but also because of the *variety* of data types. Modern distributed processing frameworks such Hadoop MapReduce [3, 17] or Spark [4, 60] allow handling large volumes of data, potentially generated at a high velocity by taking advantage of computer clusters. This thesis focuses on the variety aspect of big data. Current datasets contain not only text data (web pages, messages, chats etc.), but also other types of data (images, videos, music, purchase histories etc.), for which conventional text processing approaches do not apply.

1.2 Applications of Nearest Neighbor Search

This thesis deals with high-dimensional data, a specific type of data that is found in many applications. In a scalar dataset, each record is a single integer or floating-point number. For instance, datasets of temperature records, power records, or any scalar physical quantity are scalar datasets. Datasets can also be composed of 2-dimensional or 3-dimensional points. For instance, in a dataset of post office positions or train station positions, each record would be a 2-dimensional point. More generally, a dataset can be seen as a collection of d -dimensional points, or vectors. Thus, a scalar dataset can be seen as a dataset of 1-dimensional vectors. By contrast, a high-dimensional dataset comprise high-dimensional vectors, i.e., d -dimensional vectors with a high dimensionality d . We consider that vectors with more than 100 dimensions, i.e., $d > 100$, are high-dimensional.

High-dimensional datasets span many different domains: genetics, recommender systems, multimedia data processing etc. In genetics, DNA microarrays are used to measure the expression of genes in different individuals. A DNA microarray is able to measure the expression of thousands of genes, resulting in a vector with thousands of dimensions. Gene expression measurements are used to study the impact of drugs and diseases on multiple individuals, producing a dataset of multiple high-dimensional vectors. A recommender system recommends items e.g., books, movies or products, based on how a given user rated other items. In a recommender system, the profile of a user consists of the ratings she gave to the items she bought. This profile is represented by a high-dimensional vector, where each dimension corresponds to an item in the recommender system. Typical recommender systems obviously include the profile of multiple users, thus resulting in a large dataset of high-dimensional vectors. Multimedia objects (such as pictures or movies), can be represented by high-dimensional feature vectors, that capture their contents. Therefore, a multimedia dataset, including multiple pictures or movies, can be represented as a datasets of high-dimensional vectors.

In this thesis, we focus on nearest neighbor search in high-dimensional datasets. Simply put, this problem consists in finding the closest vector to a query vector among a database of high-dimensional vectors. Despite its apparent simplicity, nearest neighbor search has many applications in machine learning and multimedia data processing. In machine learning, nearest neighbor search can be used as a standalone algorithm, but it is also at the basis of many algorithms. Examples of such algorithms include collaborative filtering algorithms, used in recommender systems. A classifier is a machine learning system which determines the category a new observation belongs to based on a training set of prior observations, the categories of which are known. A nearest neighbor classifier determines the category of a new observation by finding its nearest neighbor in the training set, and returning the category of the found nearest neighbor.

Above all, nearest neighbor search is of particular importance for multimedia information retrieval problems. Content Based Image Retrieval (CBIR), consists in finding the images in a database that are the most similar to a query image provided by the user. As the user essentially provides an example image of what he wants to retrieve, content-based image retrieval is also known as query by example. CBIR

systems rely heavily on nearest neighbor search in high-dimensional spaces, and are the guiding theme of this thesis. In particular, the nearest neighbor search systems presented in this thesis were evaluated in the context of CBIR systems, but may be used for other purposes. In the remainder of this section, we give an overview of how CBIR systems work. CBIR systems rely on the extraction of feature vectors, or descriptors, that capture the visual contents of pictures. Finding two similar pictures then comes down finding two pictures that have a similar set of descriptors. CBIR systems typically operate in two distinct phases: an offline phase and an online phase. During the offline phase, descriptors are extracted from every image in the dataset, therefore creating a database of high-dimensional descriptors. During the online phase, an user submits a query image to the system. High-dimensional query descriptors are extracted from the query image. The nearest neighbors of the query descriptors are searched in the database of high-dimensional descriptors created during the offline phase, which in turn allows to identify the most similar images to the query image (Figure 1.1). Because nearest neighbor search is used during the online phase, it is essential to use fast nearest neighbor search solutions in order to answer queries in a timely manner.

As the nearest neighbor search systems presented in this thesis have been tested on databases of high-dimensional descriptors extracted from pictures, we present these descriptors into more detail. The dimensionality of currently used descriptors ranges from a hundred dimensions to a few thousand dimensions. We distinguish between global descriptors and local descriptors, depending on the number of descriptors extracted per picture. Global descriptors describe a complete image, therefore an image can be described by a single global descriptor. On the contrary, local descriptors describe a small region of the picture, named patch. Therefore, an image can be described by a set of local descriptors. The number of local descriptors required to describe an image usually ranges from a hundred to a thousand. One of the most popular global descriptor is the GIST descriptor [47] the dimensionality of which ranges 384 and 960 dimensions. The most popular local descriptor is arguably the Scale Invariant Feature Transform (SIFT) descriptor [43], with 128 dimensions. An alternative to the SIFT descriptor with similar properties is the SURF descriptor [11], implemented in the open-source library OpenCV. Descriptors like SIFT and SURF descriptors were designed manually to have a strong descriptive power while being resistant to image variations. It has recently been proposed to use deep neural networks to learn robust descriptors with strong descriptive power without human intervention. These deep feature vectors tend to very high-dimensional (usually 4096 dimensions), but have been found to have a higher descriptive power than other descriptors, which is why they are gaining in popularity.

For CBIR, local descriptors generally offer a better accuracy than their global counterparts. In addition, local descriptors enable object detection, i.e., identifying objects inside pictures, which is not possible with global descriptors. For these reasons, local descriptors are often preferred to global descriptors. The drawback of local descriptors is that they lead to the creation of very large high-dimensional databases, as a single image is described by hundreds of local descriptors instead of a single global descriptor. This in turn increases the pressure on the nearest neighbor search system, which has to deal with a database hundreds of times larger.

1.3 Classification of Nearest Neighbor Search Problems

Most practical applications of nearest neighbor search require a set of near neighbors of the query vectors instead of the nearest neighbor. For instance a CBIR system returns multiple images similar to the query image, and not a single one. Similarly, recommender systems recommend items to an user based on the preferences of a set its nearest neighbors. We generally distinguish between two different nearest neighbor search problems: k -nearest neighbor search (k -NN) and ϵ -nearest neighbor search (ϵ -NN). Given a dataset $\mathcal{X}$ of n d -dimensional vectors, $\mathcal{X} = \{x_0, \dots, x_{n-1}\}$ and d -dimensional query vector q , these two problem are defined as follows:

k -NN The k -NN set of q is the set of k vectors of $\mathcal{X}$ that are the closest to q .

ϵ -NN The ϵ -NN set of q is the set of vectors of $\mathcal{X}$ whose distance to q is less than ϵ . As ϵ -NN search returns all neighbors in the hypersphere centered on q of radius ϵ , it is also known as radius search.

For many applications, it is generally more intuitive to pick a number k of nearest neighbors than a threshold ϵ , which is harder to determine. For instance, it is easier to determine that a CBIR system should return $k = 100$ similar images, rather than to determine a similarity threshold ϵ . Therefore, k -NN is often preferred to ϵ -NN. In this thesis, we focus on the k -NN problem. An important observation is that these two problems are related, the ϵ -NN set of q is the same as the k -NN set of q if the radius ϵ is the distance between q and its k th nearest neighbor. Therefore, methods designed to solve the ϵ -NN problem can be adapted to solve the k -NN problem.

The notion of nearest neighbor implies a similarity measure or distance metric. Depending on applications, different distance metric are used: l_1 norm, l_2 norm or even cosine similarity. In this thesis, we focus on the l_2 norm, as it is one of the most widely used distance metric. More specifically, it is used in CBIR systems as well as many machine learning algorithms.

1.4 The Curse of Dimensionality

The naive approach to solve the k -NN problem is to compute the distance between the query vector q and each vector of the dataset $\mathcal{X}$. This solution, known as linear scan or brute force scan, is obviously not tractable for large datasets. A dataset of $n = 10^9$ of SIFT descriptors ($d = 128$ dimensions) stored as a contiguous array of floats (4 bytes/float) uses $S = n \cdot d \cdot 4$ bytes = 512 GB of memory. If this dataset is stored on an SSD with a read throughput of 512 MB/s, scanning the whole dataset would require 1000 seconds, or 17 minutes. Thus, answering a single k -NN query would require 17 minutes, assuming that there are no other bottlenecks. This is of course not practical for an interactive CBIR system. Offline machine learning algorithms, such as recommender systems, often require to compute thousands to millions k -NN sets so linear scan is also not tractable for many non-interactive applications.

In computer science, divide-and-conquer is a common approach that often allows solving problems with a better efficiency than brute force algorithms. In a 1-dimensional space i.e., when the dataset $\mathcal{X}$ contains scalar values, nearest neighbor search admits a simple divide-and-conquer solution, binary search. The dataset $\mathcal{X}$

1. Introduction

should first be stored in an array and sorted in ascending order. The binary search algorithm then compares the scalar q to the middle element of $\mathcal{X}$. If the q is less than the middle element, then its nearest neighbor is in the first half of $\mathcal{X}$. Conversely, if q is greater than the middle, its nearest neighbor is in the second half. This procedure is recursively repeated until the nearest neighbor of q is found. Binary search has a logarithmic time complexity, while a brute-force scan has a linear complexity.

For spaces with 2 or more dimensions, the dataset $\mathcal{X}$ obviously cannot be sorted, but divider-and-conquer approaches based on space partitioning can still be designed. A popular structure for nearest neighbor search in d -dimensional spaces is the KD-tree [13, 20]. Each non-terminal node of KD-tree consists of one high-dimensional vector and an hyperplane orthogonal to one dimension of the hyper-space. Each terminal node only consists of one high-dimensional vector. The hyperplane of the root of the KD-tree is positioned at the median of the dimension where the data exhibits the greatest variance, thus halving the dataset. To build the other nodes, the dataset is recursively split in the same way until partitions contain only a single point. These single points form the terminal nodes of the tree. Thus, each node of a KD-tree defines a cell in the high-dimensional space. To answer a nearest neighbor query, a tree descent is performed by determining on which side of the hyperplanes the query vector q lies. This yields a first nearest neighbor candidate. This candidate is not necessarily the true nearest neighbor of q . Cells intersecting with the ball whose radius is the distance between q and the first candidate can contain a closer neighbor than this first candidate. These cells are therefore searched for better candidates. Nearest neighbor search stops when there are no cells intersecting with the hypersphere whose radius is the distance of q to the best candidate found so far.

Unfortunately, it has been shown that the number of cells that must be explored to be sure to find the true nearest neighbor of q grows exponentially with the dimensionality d [20]. Thus, nearest neighbor search with a KD-tree degrades to linear scan as the dimensionality increases. Worse, in [57], the authors show that it is not possible to design a space partitioning method that does not degrade to linear scan as the dimensionality increases. This phenomenon is known as the curse of dimensionality. As exact nearest neighbor search is inherently hard, the research community has focused on approximate nearest neighbor search. Approximate Nearest Neighbor (ANN) search aims at returning close enough neighbors instead of the exact closest ones. The key idea is to trade exactness for efficiency. Moreover, approximate nearest neighbors are sufficient in many applications, including CBIR systems. An obvious tradeoff in the case of KD-trees is to stop exploring cells after a fixed number have been explored. This reduces search time, but does not guarantee that the exact nearest neighbor will be found. More sophisticated approaches have been designed; we review the most prominent ones in the state of the art section.

1.5 Product Quantization

Product Quantization is a widespread ANN search solution that compresses high-dimensional vectors into short codes. In most cases, high-dimensional vectors can be represented by codes of 8 bytes to 16 bytes (64 bits to 128 bits). This allows very large datasets to be stored in RAM, therefore enabling nearest neighbor queries without accessing the SSD or HDD. RAM typically has a throughput in excess of 40GB/s

and a latency of around 100ns, while the highest-performing SSDs have a throughput of 2-3GB/s and a latency of 100µs. Because it entirely relies on RAM, product quantization allows answering nearest neighbor queries faster than approaches that rely on SSDs. To compress a high-dimensional vector into a short code, product quantization splits it into m sub-vectors and encodes each sub-vector using a distinct sub-quantizer. Each sub-quantizer has a codebook of 2^b entries, and produces a sub-code of b bits. For fixed memory budget of $m \times b$ bits per short code, a large b (and therefore a small m) offers a better accuracy, but makes nearest neighbor search slower.

In practice, product quantization is used in combination with a form of inverted index. Different types of inverted indexes have been proposed, but they all split the database into multiple inverted lists so that only a fraction of the database has to be scanned at query time. However, because of the curse of dimensionality, it is difficult to build efficient indexes. Thus, it is still necessary to scan a large part of the database to answer queries (typically 1-20%). Generally, database preparation takes two steps. First, the database is split into several parts using the index. Second, the high-dimensional vectors composing each part are compressed into short codes using product quantization, as described in the previous paragraph. Both the index and the short codes are stored in RAM, to enable fast answers to ANN queries.

One of the unique features of product quantization is that it is able to compute the distance between an uncompressed high-dimensional query vector y and compressed database vectors. Therefore, it is not necessary to decompress the database to answer ANN queries. To answer an ANN query, the relevant partition is first selected using the index, and then scanned for nearest neighbors. Scanning the partition consists in computing the distance between the query vector y and all short codes of the partition. To do so, a set of m lookup tables, derived from the quantizer codebooks, are computed. Then, the distance between the query vector y and any short code can be computed using a technique named Asymmetric Distance Computation (ADC). ADC consists in (1) performing m lookups in the previously computed tables (2) adding the looked up values, thus performing $m - 1$ additions. To date much work has been dedicated to improving the indexes used with product quantization. *On the contrary, this thesis focuses on improving the performance of ADC, by making it more hardware friendly.*

1.6 Thesis Outline

In the next chapter of this thesis, we review the state of the art of ANN search in high-dimensional spaces. Each contribution (Performance Analysis, PQ Fast Scan, Quick ADC, Derived Quantizers) is then detailed in a dedicated chapter. The last chapter draws conclusions from the contributions presented in this thesis, and elaborates on some perspectives that they open.

State of the Art We present three families of nearest neighbor search approaches: (1) Space and data partitioning trees, (2) Locality Sensitive Hashing (LSH) and (3) Product Quantization. We show that partitioning trees and LSH use large amounts of memory when dealing with large databases. This makes it necessary to use the SSDs or HDDs, causing high response times. We then present product quantization and

show that it can store large databases entirely in RAM, which allows low response times. All our contributions build on product quantization, therefore the last section of the State of the Art chapter is also the background of our work.

Performance Analysis Using hardware performance counters, a mechanism included in CPUs that allows monitoring the operations they perform, we analyze the performance of ADC. We show that the table lookups performed by ADC account for the majority of CPU time, even if these lookup tables are cache resident. We show that number b of bits per sub-quantizer strongly impacts response time, because it impacts the cache level in which lookup tables are stored. In almost all publications on product quantization, 8-bit sub-quantizers ($b = 8$) are used because it has been experimentally observed that they offer a good tradeoff between speed and accuracy. Our analysis explains why, but also suggests means of improving this tradeoff. Lastly we introduce SIMD instructions (Single Instruction Multiple Data) and show that they are a candidate for accelerating the additions performed by ADC. We however demonstrate that ADC cannot be easily implemented in SIMD, which prevents obtaining a large speedup. All systems presented in this thesis (PQ Fast Scan, Quick ADC, Derived Quantizers) stem from the conclusions of our performance analysis.

PQ Fast Scan PQ Fast Scan achieves a 4-6 times speedup over the conventional ADC algorithm. This is achieved by replacing cache accesses to lookup tables by SIMD in-register shuffles. SIMD in-register shuffles are much faster than cache accesses, but they require lookup tables to be stored in SIMD registers, which are much smaller than cache. PQ Fast Scan shrinks lookup tables by (1) reorganizing short codes, (2) computing tables of minimums and (3) quantizing floating-point distances to 8-bit integers. PQ Fast Scan uses 8-bit sub-quantizers, as it is common practice in the literature, and provides the exact same results as the conventional ADC algorithm. The drawback of PQ Fast Scan is that because it re-organizes short codes, it requires the database to be split in relatively coarse parts. Therefore, it is not compatible with the most efficient types of inverted indexes usually used with product quantization.

Quick ADC. Like PQ Fast Scan achieves a 4-6 times speedup over the conventional ADC algorithm using SIMD in-register shuffles. However, Quick ADC uses a different approach to shrink lookup tables. Quick ADC shrinks lookup tables by (1) using 4-bit sub-quantizers instead of the common 8-bit sub-quantizers and (2) quantizing floating point distances to 8-bit integers. Using 4-bit sub-quantizers instead of 8-bit sub-quantizers decreases accuracy but contrary to PQ Fast Scan, Quick ADC imposes no constraints on inverted lists, and may be combined with any type of inverted index. Moreover, we show that by using optimized product quantization, a derivative of product quantization, the loss of accuracy caused by the use of 4-bit sub-quantizers can be made negligible in most cases.

Derived Quantizers. While PQ Fast Scan and Quick ADC aim at slashing response time without impacting accuracy, derived quantizers aim at increasing accuracy without increasing response time. To increase accuracy, it is possible to use 16-bit sub-quantizer instead of the 8-bit sub-quantizer used in most publications. However, this strategy causes a threefold increase in response time, because lookup

tables have to be stored in slower cache levels. Derived quantizers allow combining the accuracy of 16-bit sub-quantizers with the speed of 8-bit sub-quantizers. This is achieved by deriving 8-bit sub-quantizers from 16-bit sub-quantizers, so that they generate the same short codes. At query time, 8-bit (fast) sub-quantizers are used to build a small candidate set, which is then reranked using the 16-bit (precise) sub-quantizers.

Conclusion and Perspectives We conclude by reviewing the key features of our contributions, and we discuss the research opportunities that stem from them. More specifically, we discuss how our contributions can be adapted to recent derivatives of product quantization that offer a higher accuracy. We also show that our contributions can be implemented on other CPUs than Intel CPUs (and in particular ARM CPUs). Lastly, we discuss the applicability of the techniques we developed in the context of product quantization to other algorithms that rely on lookup tables.

2 State of the Art

Contents

2.1	Space and Data Partitioning Trees	12
2.1.1	KD-trees	12
2.1.2	Bag of Features and Vocabulary Trees	13
2.1.3	FLANN Library	15
2.2	Locality Sensitive Hashing (LSH)	16
2.2.1	Euclidean Distance LSH (E2LSH)	16
2.2.2	Virtual Rehashing Schemes	18
2.3	Product Quantization	19
2.3.1	Vector Encoding	20
2.3.2	Inverted Indexes	22
2.3.3	ANN Search	24
2.3.4	Product Quantization Derivatives	27

In this chapter, we present three families of ANN search approaches: (1) Space and data partitioning trees, (2) Locality Sensitive Hashing (LSH) and (3) Product Quantization (PQ). We first present Space and data partitioning trees and LSH. We show that these two families of approaches use large amounts of memory, mandating the use of SSDs or HDDs to store large databases. We then present product quantization, an approach that compresses high-dimensional into short codes, making it possible to store large databases in RAM. Because RAM is much faster than SSDs or HDDs, product quantization offers low response times, especially for large databases. We explain how product quantization can be combined with inverted indexes to further decrease response times. Some of the inverted indexes used in combination with product quantization were inspired by the space and data partitioning trees presented in the beginning of this chapter.

All the contributions of this thesis are based on product quantization, thus the last section of this chapter (Section 2.3) also describes the background of our work.

2.1 Space and Data Partitioning Trees

The first data structures proposed for efficient nearest neighbor search are space and data partitioning trees. Space-partitioning trees (KD-trees) divide the high-dimensional space along predetermined lines. On the other hand, data partitioning trees (Vocabulary trees, k-means trees) divide the high-dimensional space according to data clusters.

2.1.1 KD-trees

Tree structure KD-trees [13, 20] can be seen as a generalization of binary search trees. At each non-terminal node of a KD-tree, the dataset is divided into two halves, by a hyperplane orthogonal to a chosen dimension at a threshold value. In the literature, the chosen dimension is sometimes known as the discriminator and the threshold is known as the partition value. Generally, the dimension where the data exhibits the greatest variance, or the greatest spread, is chosen as the discriminator. The median of the values in the chosen dimension is chosen as the partition value. Each non-terminal node of a KD-tree contains two pointers to two sons, or successors nodes. These two successor nodes start two sub-tree. The left sub-tree contains the high-dimensional vectors for which the value in the dimension chosen as discriminator is less than the partition value. Correspondingly, the right sub-tree contains the high-dimensional vectors for which the value in the dimension chosen as discriminator is greater or equal to the partition value. Each terminal node generally contains a single high-dimensional vector of the dataset, although in some implementations, terminal nodes may contain more than one vector. Each node in a KD-tree defines a cell in $\mathbb{R}^d$. Moreover, The cells defined by terminal nodes are mutually exclusive, and thus form a partition of $\mathbb{R}^d$.

Build process KD-trees are built in a top-down fashion, following a simple recursive process. At the root, the whole dataset is split by a hyperplane at the median of the dimension where the data has the greatest variance. The resulting two halves of the data are then recursively split until the tree is fully built. The resulting height of the tree is $O(\log_2(n))$, where n is the number of points in the dataset. Figure 2.2 shows a KD-tree with 3 levels. The hyperplane of the root is shown by a solid line, the hyperplanes of the second level nodes are shown by dashed lines, while the hyperplanes of the third level nodes are shown by dotted lines.

Nearest Neighbor Search To answer a nearest neighbor query, a tree descent is first performed. This tree descent consists in determining to which side of the hyperplane the query vector falls at each node of the tree. The descent involves $\log_2(n)$ comparisons and yields a terminal node, and thus a cell in $\mathbb{R}^d$. The vectors located in this cell are examined to find a first nearest neighbor candidate. However this first candidate may not be the nearest neighbor of the query vector. For instance, if the query vector is close to a boundary of the cell the tree descent landed in, the first candidate will be a vector in this cell. There may however be a closest neighbor in an adjacent cell, close to the boundary. In the remainder of this section, we denote B_{query} , the ball centered at the query vector with a radius equal to the distance between the query vector and the current nearest neighbor. To find the exact nearest neighbor, it is therefore necessary to explore all cells the boundaries of which overlap the ball B_{query} (Figure 2.2). In [57], the authors show that the number of cells

that need to be explored grows rapidly with the dimensionality, to the point where almost the whole dataset needs to be explored when the dimensionality exceeds 15. This phenomenon is a consequence of the curse of dimensionality. This issue can be overcome by stopping search when a fixed number n_{max} of cells have been explored. However, the guarantee of finding the exact nearest neighbors is lost, and this is therefore an approximate solution.

Priority Search In the original search algorithm, the cells to explore are determined by backtracking in the tree, and checking if hyperplanes overlap with the ball centered at the query vector. The order in which cells are explored thus depends on the structure of the tree, and not on the position of the cells relative to the query vector. However, cells that are close to the query vectors are more likely to contain good neighbors than cells that are far away. Therefore, an intuitive idea to find better approximate neighbors is to explore cells based on their proximity with the query vector. This can be achieved by maintaining a priority queue during the initial tree descent. At each node, the position in the tree, and the distance of the query vector to the unexplored cell (i.e., the cell at the other side of the hyperplane) are stored in the priority queue. When backtracking, the top entry of the priority queue is removed, and the search continues by exploring the corresponding cell. This approach is known as Best Bin First (BBF) [12] or priority search.

KD-forests Exploring more cells to improve the quality of approximate neighbors leads to diminishing returns [51]: the more cells have been explored, the less the benefit of exploring an additional cell is. This is because searches in different cells are not independent: the more explored cells, the further away the cells are from the query point. In [51], the authors propose building l KD-trees with a different structure, such that they allow independent searches. In total, n_{max} cells are explored, so n_{max}/l cells are searched in each tree. Three techniques are proposed to build forests of independent trees: NKD-trees, RKD-trees, and PKD-trees. To build an NKD-tree, vectors are randomly rotated before building the tree. To build an RKD-tree, instead of using the one dimension with the greatest variance to position the hyperplane, a random dimension among the few with the greatest variance is chosen. Lastly, PKD-trees rely on Principal Component Analysis and random rotations. The authors show that NKD-trees and RKD-trees perform equally. PKD-trees perform slightly better but are much more costly to build.

2.1.2 Bag of Features and Vocabulary Trees

Bag of features The Bag of Features (BoF) approach [52] was not original presented as generic nearest neighbor search system, but as a Content Based Image Retrieval (CBIR) system. However, it was one of the first image retrieval system that scaled well, and can be adapted into a generic nearest neighbor search system, which is why we present it. We first present the original approach, and then show how to generalize it into a generic nearest neighbor search system. In a CBIR systems, an image is described by a set of high-dimensional feature vectors, or descriptors, where each descriptor captures the contents of a patch of the image (Section 1.2). The key idea of the BoF approach is to partition the vector space into K mutually exclusive Voronoi cells using the k-means algorithm. A subset of the descriptors extracted from the database of images is used to learn the partition. This partition is used to build

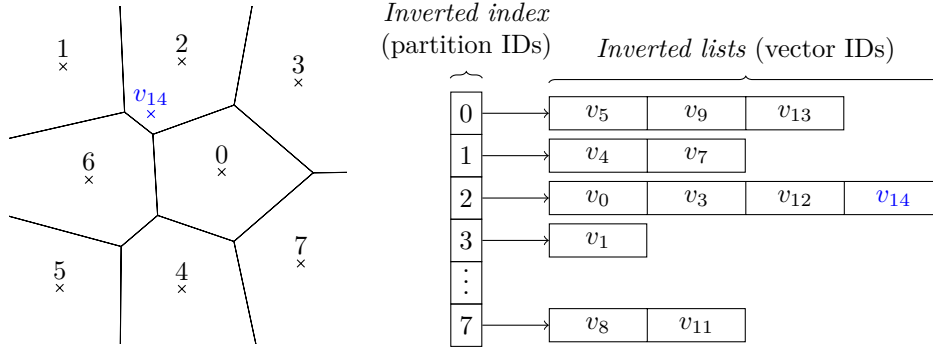


Figure 2.1: ANN search system based on an inverted index

and *inverted index*. The inverted index takes the form of an array of K elements, where each element corresponds to one Voronoi cell of the partition. Each element contains a pointer to a list of image IDs. When an image is added to the database, its descriptors are extracted. Then the cell in which each descriptor falls is determined, and the image ID is added to the corresponding lists of the inverted index. For instance, if an image has descriptors that fall in cells 5, 768 and 1023, the ID of this image is added to the lists 5, 768 and 1023 of the inverted index. To retrieve the most similar image to a query image, the descriptors of the query image are first extracted. The cells in which they fall are determined, and the corresponding lists of image IDs are retrieved from the inverted index. A voting scheme is then applied to determine the most similar image. The simplest voting scheme consists in finding the image with the most frequent image ID among the retrieved lists, but more elaborate schemes have been proposed [52]. As this system was inspired by text search engines, where documents are indexed based on the words they contain, the Voronoi cells are sometimes named *visual words*.

The idea of using an inverted index, based on a Voronoi partition of the vector space, can be adapted to build a generic nearest neighbor search system. Instead of storing images IDs in the lists of the inverted index, high-dimensional vectors can be stored. Adding a vector to the database then consists in determining in each cell it falls, and adding it to the corresponding inverted list (Figure 2.1). To answer a nearest neighbor query, the cell in which the query vector falls is first determined. The corresponding list is retrieved, and the distance of the query vector to all vectors in the retrieved list is computed to determine the nearest neighbor. To improve the accuracy, the ma inverted lists corresponding to the ma closest cells to the query vector can be scanned.

Vocabulary trees The main issue with the bag of features approach, especially when used as an image retrieval system, is that the size of the visual vocabulary, i.e., the number K of Voronoi cells, is limited. Dividing the space into more than $K = 2^{16}$ using k-means is not tractable. However, using a large vocabulary is desirable, as it better discriminates between similar images patches, and thus provides a better accuracy. The vocabulary tree [45] allows using a larger visual vocabulary (i.e., millions of visual words), while maintaining an acceptable computational cost. In [45],

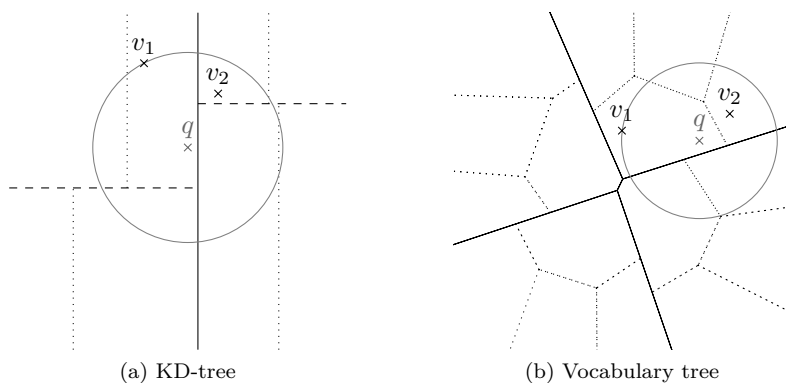


Figure 2.2: ANN search in KD-trees and Vocabulary trees

vocabulary trees store image IDs, like in the bag of features approach described in [52]. The vocabulary tree is built by the means of a hierarchical k-means clustering. Like in the bag of features approach, the vector space is first partitioned into k cells using the k-means algorithm. This partitioning is used to build the root of the vocabulary tree: the root node has k child nodes, each one corresponding to a cell. Each cell is then divided into k cells using the k-means algorithm to build the second level of the tree. This process is recursively applied to build the other levels of the tree, until the tree reaches a depth l . The resulting vocabulary has a branching factor of k , and a depth of l , therefore the number of terminal nodes and the total number of cells is k^l . Figure 2.2 shows a vocabulary tree with a branching factor $k = 4$ and a depth $l = 2$. The root divides the space in $k = 4$ cells (solid lines). At the second level of the tree, each cell is again divided into $k = 4$ cells (dotted lines). A similar idea (hierarchical k-means tree) had been proposed in [21].

2.1.3 FLANN Library

FLANN [44] (Fast Library for Approximate Nearest Neighbors) is an open-source ANN search library, included in the widely used computer vision framework OpenCV. Both FLANN and OpenCV are distributed under the BSD license. Because it is open-source, and because it is one of the only production ready ANN search libraries that can be used by non-specialists, FLANN is highly popular and used in many computer vision projects. FLANN mainly relies on (1) forests of randomized KD-trees (RKD-trees, Section 2.1.1) and (2) the priority search k-means tree. The k-means trees used in FLANN are similar to the vocabulary trees used in [45], except that they store high-dimensional vectors, instead of image IDs. Therefore, high-dimensional vectors of the database are added to the tree by perform a tree descent. The vectors are then stored in the obtained terminal nodes. To search these trees, the authors of [44] propose using the same priority search method as the one used to search KD-trees. Thus, to answer a nearest neighbor search query, a first tree descent is performed to generate a first candidate. It is followed by a backtracking process where the cells that overlap B_{query} (ball centered at the query vector with a radius equal to the distance between the query vector and the current nearest neighbor) are explored (Figure 2.2). The key advantage of FLANN is that it chooses the best algorithm, and determines the optimum parameters automatically. This selection is based on

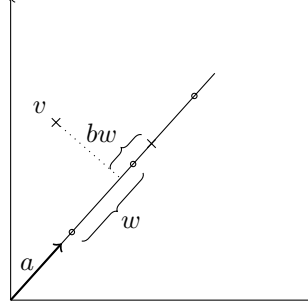


Figure 2.3: Geometrical interpretation of LSH functions

the type of high-dimensional vectors, and user-specified constraints: desired accuracy, search time, maximum memory overhead, maximum tree build time. This feature makes the FLANN library easy to use for non-specialists.

2.2 Locality Sensitive Hashing (LSH)

Locality Sensitive Hashing (LSH) is another prominent nearest neighbor search approach. The key difference between LSH and other approaches mentioned in this thesis is that LSH offers theoretical guarantees on the quality of the returned nearest neighbors. LSH tackles a variant of ANN search, the c -approximate nearest neighbor search problem. Formally, a vector v is a c -approximate nearest neighbor of a query vector q if the distance between v and q is at most c times the distance between q and its exact nearest neighbor v^* , i.e., $\|v - q\| \leq c \cdot \|v^* - q\|$. Accordingly, the c -approximate k nearest neighbors problem consists in finding k vectors that are respectively the c -approximation of the exact k nearest neighbors of q .

2.2.1 Euclidean Distance LSH (E2LSH)

LSH was originally proposed for the Hamming space (l_1 norm) [25], and later adapted to Euclidean spaces (l_2 norm), with the E2LSH method [16]. As this thesis focuses on nearest neighbor search methods for the l_2 norm, we only describe the E2LSH method, and not the original Hamming space LSH method.

E2LSH does not solve the c -approximate nearest neighbor problem directly but rather focuses on the (R, c) -approximate nearest neighbor problem, a decision version of the c -approximate nearest neighbor problem. In the remainder of this section, we denote $B(q, r)$, the ball centered at the query vector q with the radius r . With this notation, the (R, c) -approximate nearest neighbor problem is defined as follows:

1. If there is at least one vector in $B(q, R)$, a vector in $B(q, cR)$ is returned.
2. If there is no vector in $B(q, cR)$, nothing is returned

The (R, c) -approximate nearest neighbor problem is sometimes known as "ball-cover" search in the literature.

E2LSH uses (R, cR, p_1, p_2) -sensitive LSH functions to solve the (R, c) -approximate nearest neighbor problem. A (R, cR, p_1, p_2) -sensitive LSH function h has the following properties, for any $v \in \mathbb{R}^d$:

1. If $v \in B(q, R)$, then $\Pr[h(v) = h(q)] \geq p_1$
2. If $v \notin B(q, cR)$, then $\Pr[h(v) = h(q)] \leq p_2$

The LSH functions used in E2LSH have the following form:

$$h(v) = \left\lfloor \frac{a \cdot v + bw}{w} \right\rfloor \quad (2.1)$$

In these functions, a is a vector $a \in \mathbb{R}^d$, where each dimension has been drawn from a normal distribution (mean 0, variance 1). The parameter w is a scalar constant, and b is a real number uniformly drawn from $[0, 1)$. These LSH functions can be interpreted geometrically: the vector v is first projected onto a line, the direction of which is given by the vector a (Figure 2.3). The projection of v is then shifted by the constant bw . The line is divided into intervals of size w , and the hash of v is the number of the interval in which its shifted projection falls. It has been shown [16] that the probability $\Pr[h(v) = h(q)]$ that two vectors q and v collide under h decreases when $r = \|q - v\|$ increases, but increases when w increases. Intuitively, we see that two far away points may still fall in the same interval, for instance, if they are far away in a direction orthogonal to a . To increase the discriminating power, E2LSH therefore uses compound hash functions G , which consist in the concatenation of m LSH functions h : $G(v) = (h_1(v), \dots, h_m(v))$. E2LSH uses L distinct compound hash functions, $G_1, \dots, G_L$ to build L hash tables, $T_1, \dots, T_m$. Each database vector v is hashed using the L compound hash functions and stored in the corresponding bucket of each of the L hash tables. To answer nearest neighbor queries, E2LSH hashes the query vector q with the L compound hash functions $G_1(v), \dots, G_m(v)$ and retrieves the vectors stored in the corresponding buckets of $T_1, \dots, T_m$. The number of vectors retrieved is limited to $3L$ to avoid checking a too large number of candidates. For each one of the $3L$ retrieved vectors, E2LSH checks if it is in $B(q, cR)$. As soon as a vectors in $B(q, cR)$ is found, it is returned and search stops. If no vector in $B(q, cR)$ can be found, E2LSH returns nothing. The m and L parameters are chosen such that the two following properties hold with a constant probability: (1) if there is a database vector v in $v \in B(q, cR)$, then $G_j(v) = G_j(q)$ must be true for at least one compound hash function, and (2) the total number of vectors v that collide with q under one G_j and which are not in $B(q, cR)$ is less than $3L$.

The scheme described so far solves the (R, c) -approximate nearest neighbor problem. The c -approximate nearest neighbor problem can be solved by issuing several (R, c) nearest neighbor queries with an increasing radius $R = \{1, c, c^2, c^3, \dots\}$. As the m and w parameters used in compound hash functions $G_1(v), \dots, G_L(v)$ depend on the radius R , different compound hash functions must be used for different radius R . Therefore, a set of L hash tables has to be built for each radius $R = \{1, c, c^2, c^3, \dots\}$. This approach is known as *rigorous-LSH*, and offers theoretical guarantees on the quality of nearest neighbors. However, the duplication of hash tables leads to a very high memory use, and makes rigorous-LSH intractable for anything but very small datasets (up to a few tens of thousands of vectors). Another approach was proposed, *ad hoc-LSH*, which consists in building hash tables for a single "magic" radius r_m . This

greatly reduces the memory cost, but theoretical guarantees are lost. Furthermore, it has been shown that this approach offers poor empirical performance.

2.2.2 Virtual Rehashing Schemes

The main challenge in designing an efficient LSH scheme is to offer theoretical guarantees or good empirical performance while maintaining an acceptable memory use. Modern LSH schemes rely on a form a *virtual rehashing*, although the details vary widely. The key idea of virtual rehashing is to allow queries with different radii, without physically building a set of hash tables for each radius $R = \{1, c, c^2, c^3, \dots\}$. We present three LSH schemes that rely on virtual rehashing: LSB-forest [54], Collision-counting LSH (C2LSH) [22], and Sorting-Keys LSH (SK-LSH) [41]. Both LSB-forest and C2LSH offer theoretical guarantees while SK-LSH does not. However, SK-LSH has been shown to offer better empirical performance than LSB-forest or C2LSH.

LSB-forest [54] Like E2LSH, LSB-forest hashes database vectors v using compound hash functions $G_j(v)$. The m -dimensional hash values $G_j(v)$ are converted to Z-order values $z_j(v)$. Z-order values are bit strings, obtained by grid partitioning the m -dimensional hash value space. An LSB-tree is then created by building a conventional B-tree over the Z-order values. Lastly, an LSB-forest is created by building L LSB-trees using L distinct compound hash functions $G_1, \dots, G_L$ (converted to Z-order values $z_1(v), \dots, z_L(v)$). To answer a nearest neighbor search query, the L LSB-trees are explored. To explore an LSB-tree, the Z-order value of the query vector q is computed $z_j(q)$. The entries in LSB-trees are explored by decreasing Length of the Longest Common Prefix (LLCP) with $z_j(q)$. Exploring Z-order values $z_j(v)$ in decreasing order of their LLCP with $z_j(q)$ simulates the process of issuing several (R, c) nearest neighbor queries with an increasing radius.

Collision-counting LSH (C2LSH) [22] Unlike E2LSH or LSB-forest, C2LSH does not use predetermined compound hash functions G_j . Instead, C2LSH builds m hash tables $T_1, \dots, T_m$ using m $(1, c, p_1, p_2)$ -sensitive LSH functions ($R = 1$) $h_1, \dots, h_m$, and stores database vectors into these hash tables. To answer a nearest neighbor search query, C2LSH hashes the query vector q with all hash functions $h_1, \dots, h_m$, and retrieves the vectors in the corresponding buckets. The list of retrieved vectors is likely to contain duplicates, as it is likely that some databases vector collide with q under multiple hash functions. C2LSH builds candidate set C of database vectors that collide with q under l or more hash functions, i.e., that are duplicated at least l times in the list of retrieved vectors. This collision counting procedure can be seen as the dynamic creation of compound hash functions, tailored to the query vector. This allows C2LSH to answer (R, c) nearest neighbor queries for $R = 1$. In [22], the authors show that if a function $h(\cdot)$ is $(1, c, p_1, p_2)$ -sensitive, then the function $H^R(\cdot) = h(\cdot)/R$ is (R, cR, p_1, p_2) -sensitive. If R is an integer, the $T_1, \dots, T_m$ hash tables can therefore be used to answer (R, c) nearest neighbor queries for $R > 1$ but retrieving R contiguous buckets.

Sorting-Keys LSH (SK-LSH) [41] Like E2LSH and LSB-forest, SK-LSH uses compound hash functions G_j . In [41], the authors build a linear order on the m -dimensional hash keys generated by compound hash functions. This linear order allows building B-trees on compound hash keys. Database vectors are hashed using a

Table 2.1: Memory use of different LSH approaches (SIFT1M)

Approach	LSB	C2LSH	SK-LSH
Memory use	81 GB	1.1 GB	384 MB

small number L (usually $L = 3$) of compound hash functions $G_1, \dots, G_L$, and stored in corresponding B-trees $T_1, \dots, T_L$. To answer nearest neighbor search queries, the query vector is hashed using compound hash functions, and a tree descents are performed. Entries are explored by increasing distance with the compound hash key of the query vector, until a fixed number of entries has been explored.

While E2LSH is only suitable for very small datasets (up to a few tens of thousands of vectors), the recent improvements presented in this section make LSH suitable for small and medium datasets (up to a few million vectors). Table 2.1 summarizes the memory consumption of the LSH schemes presented in this section for a medium dataset of 1 million 128-dimensional SIFT vectors (SIFT1M). The size of the data structures used by recent LSH approaches makes it possible to store the database in RAM. However, the memory use of these approaches remains high. The size of the SIFT1M dataset is 128MB. Therefore, even SK-LSH causes a threefold increase in memory use. FLANN (Section 2.1.3) typically uses less than 250MB of RAM for this dataset. For large datasets (e.g., 1 billion vectors), LSH requires storing the dataset on an SSD or HDD, which causes high response times.

2.3 Product Quantization

So far, we presented two families of ANN search approaches : (1) Space and data partitioning trees, and (2) LSH approaches. As these approaches do not compress the database, large databases do not fit in RAM and must be stored in SSDs or HDDs. Unlike the previously presented approaches, product quantization allows large databases to be stored in RAM, by compressing high-dimensional vectors into short codes. To answer a query, short codes are scanned for nearest neighbors, without decompressing the database. Thus, for the medium SIFT1M dataset (1 million 128-dimensional SIFT vectors), product quantization only uses 8 to 20 MB of RAM. For a large database of 1 billion SIFT vectors, product quantization uses only 8 GB to 20 GB of RAM. Such a database would be impossible to store in RAM with partitioning trees or LSH.

Moreover, product quantization can be combined with inverted indexes to decrease response times. Inverted index makes it possible to answer nearest neighbor queries by only scanning a part of the database, while product quantization makes it possible to store the database in RAM. Some inverted indexes used with product quantization were inspired by the space partitioning trees presented in Section 2.1. Because RAM has much higher performance than SSDs, combining product quantization with inverted indexes allows for shorter response times than approaches that solely rely on inverted indexes.

In this section, we first describe how product quantization encodes high-dimensional vectors into short codes. We then present the different types of inverted indexes that

can be combined with product quantization. We show how a database of short codes can be searched for nearest neighbors, without decompressing it. In particular, we introduce the Asymmetric Distance Computation (ADC) procedure, which is the focus of this thesis. Lastly, we review some derivatives of product quantization, that also compress high-dimensional vectors into short codes. These recently introduced derivatives achieve a lower quantization error but sometimes involve a more costly search process. We fully discuss their advantages and drawbacks.

2.3.1 Vector Encoding

Vector Quantizer To compress high dimensional vectors as short codes, product quantization builds on vector quantizers [27]. A vector quantizer, or quantizer, is a function q which maps a vector $x \in \mathbb{R}^d$, to a vector $c_i \in \mathbb{R}^d$ belonging to a predefined set of vectors $\mathcal{C} = \{c_0, \dots, c_{k-1}\}$. Vectors c_i are called *centroids*, and the set of centroids $\mathcal{C}$, of cardinality k , is the *codebook*. For a given codebook $\mathcal{C}$, a quantizer which minimizes the quantization error must satisfy Lloyd’s condition [42]. The Lloyd condition requires that the vector x is mapped to its closest centroid c_i :

$$q(x) = \arg \min_{c_i \in \mathcal{C}} \|x - c_i\|.$$

At the implementation level, the set of centroids $\mathcal{C} = \{c_0, \dots, c_{k-1}\}$, or codebook, is stored as an array of vectors. Therefore, $\mathcal{C}[0]$ denotes the first centroid, $\mathcal{C}[1]$ the second centroid etc. Storing the codebook $\mathcal{C}$ as an array amounts to *ordering* it, but the ordering is arbitrary. Thus, any two centroids $\mathcal{C}[i], \mathcal{C}[j], i \neq j$ may be swapped without affecting the properties of the quantizer. The index of each centroid i.e., its position in the array, is used as a unique identifier for this centroid.

From a vector quantizer q , it is possible to define an encoder, a function which encodes a vector $x \in \mathbb{R}^d$ as a short code $i \in \{0, \dots, k-1\}$. For given quantizer q , the corresponding encoder enc maps a vector $x \in \mathbb{R}^d$ to the index of its closest centroid, as follows:

$$enc(x) = i, \text{ such that } q(x) = \mathcal{C}[i].$$

The short code i is an integer using $b = \lceil \log_2(k) \rceil$ bits, which is typically much less than the $d \cdot 32$ bits used to store a vector $x \in \mathbb{R}^d$ as an array of floats. For instance, using a quantizer with 2^8 centroids, a 16-dimensional vector can be encoded into a 8 bits (1 byte) short code. Stored as an array of floats, a 16-dimensional vector would use $16 \cdot 32 = 512$ bits (64 bytes) of memory. Vector quantizers are not specific to high-dimensional nearest neighbor search and are used in many fields of computer science, especially in telecommunications systems and data compression. However, encoding high-dimensional vectors (> 100 dimensions) while maintaining the quantization error low requires quantizers with a very large number of centroids e.g., 2^{64} or 2^{128} centroids. Training such quantizers is obviously intractable: storing the codebook of a quantizer with 2^{64} centroids would require more memory than the aggregated HDD or SSD capacity of all computers on earth.

Product Quantizer Product quantizers [39] overcome this issue by dividing a vector $x \in \mathbb{R}^d$ into m sub-vectors $(x^0, \dots, x^{m-1})$, assuming that d is a multiple of m :

$$x = (\underbrace{x_0, \dots, x_{d/m-1}}_{x^0}, \dots, \underbrace{x_{d-d/m}, \dots, x_{d-1}}_{x^{m-1}}).$$

Each sub-vector $x^j \in \mathbb{R}^{d/m}$ is quantized using a distinct sub-quantizer q^j . Each sub-quantizer q^j has a distinct codebook $\mathcal{C}^j = \{c_0^j, \dots, c_{k-1}^j\}$ of cardinality k . Overall, a product quantizer pq maps a vector $x \in \mathbb{R}^d$ as follows:

$$\begin{aligned} \text{pq}(x) &= (q^0(x^0), \dots, q^{m-1}(x^{m-1})) \\ &= (\mathcal{C}^0[i_0], \dots, \mathcal{C}^{m-1}[i_{m-1}]). \end{aligned}$$

By definition, a product quantizer has an implicit codebook $\mathcal{C}$ which is the Cartesian product of the sub-quantizers codebooks i.e., $\mathcal{C} = \mathcal{C}^0 \times \dots \times \mathcal{C}^{m-1}$. The codebook $\mathcal{C}$ of the product quantizer has a cardinality of k^m , but it is never stored explicitly. Here lies the key feature of product quantizers: they are able to mimic a quantizer with k^m centroids, while only requiring to train and store m codebooks of k centroids. For instance, a product quantizer with 8 sub-quantizers of 256 centroids each has an implicit codebook of $256^8 = 2^{64}$ centroids, while only requiring to store 8 codebooks of 256 centroids.

Like simple vector quantizers, product quantizers can be used to encode vectors as short codes. For a given product quantizer pq, the corresponding encoder enc maps a vector $x \in \mathbb{R}^d$ to the concatenation of indexes of the closest centroids of the sub-vectors x^j in each sub-quantizer codebook:

$$\text{enc}(x) = (i_0, \dots, i_{m-1}), \text{ such that } q(x) = (\mathcal{C}^0[i_0], \dots, \mathcal{C}^{m-1}[i_{m-1}])$$

The short code $(i_0, \dots, i_{m-1})$ is the concatenation of m integers of $b = \lceil \log(k) \rceil$ bits each, for a total size of $m \cdot b$ bits. A common practice is to use product quantizers with 8 sub-quantizers of 256 centroids each to encode 128-dimensional SIFT vectors into short codes. In this case, 128-dimensional vectors use $128 \cdot 32 = 4096$ bits (512 bytes) while their short code is composed of 8 integers of 8 bits each, or 64 bits (8 bytes) in total. Storing the database as short codes therefore achieves a 64 times decrease in memory use.

Product Quantizer Parameters When a product quantizer is used for approximate nearest neighbor search, its parameters m , number of sub-quantizers, k , number of centroids per sub-quantizer impact: (1) the memory use of the database, (2) the accuracy of nearest neighbor search and (3) the speed of ANN search. We only consider sub-quantizers whose number of centroids is a power of 2 i.e., $k = 2^b$. We name b -bit sub-quantizer a sub-quantizer with $k = 2^b$ centroids. We denote PQ $m \times b$ a product quantizers with m b -bit sub-quantizers ($k = 2^b$ centroids per sub-quantizer). An $m \times b$ product quantizer has total number of $2^{m \cdot b}$ centroids, produces short codes of $m \cdot b$ bits. The total number of centroids of a product quantizer is the parameter that has the most impact on quantization error and search accuracy [39]. This creates a trade-off between accuracy and memory use: a high $m \cdot b$ product increases both accuracy and memory use. In practice, product quantizers such that $m \cdot b = 64$ or $m \cdot b = 128$ are selected, depending on vector type. For a fixed $m \cdot b$ product (e.g., $m \cdot b = 64$), the relative values of m and b create a trade-off between accuracy and speed. For a fixed $m \cdot b$ product, it has been observed that the higher the b , the higher the accuracy but also the slower the search speed [39]. If search speed generally decreases when b increases, this decrease is not continuous; we fully discuss and explain the impact of b on search speed in Chapter 3.

2.3.2 Inverted Indexes

Exhaustive search Product quantization encodes high-dimensional vectors into short codes, and is able to compute the distance between a high-dimensional query vector and any short code. The simplest search strategy is to encode all database vectors into short codes, and store the short codes in a contiguous array in RAM. Answering a nearest neighbor query then comes down to computing the distance between the query vector and all short codes in the array, i.e., scanning the whole database. This strategy, known as *exhaustive search* [39] is suitable for databases up to a few million vectors. Even if product quantization offers an efficient way to compute distances between a high-dimensional query vector and short codes, scanning the whole database becomes too costly as its size increases.

Inverted index, non-exhaustive search To overcome this issue, *inverted indexes*, also known as *inverted files* are combined with product quantization. The database is first partitioned into several parts. Each part is then stored as an *inverted list*, a contiguous array of short codes. The inverted index structure holds pointers to all inverted lists, and provides a fast access to any one of them. At query time, the most relevant inverted lists to the query are scanned for nearest neighbors. Inverted indexes allow scanning only a fraction of the database to answer a query; this strategy is known as *non-exhaustive search*.

In general, making small inverted lists i.e., finely partitioning the database and scanning several inverted lists at query time gives better results than making large inverted lists and scanning only one. Therefore, in practice, multiple inverted lists are scanned to answer a query. This method is known as *multiple assignment*. There are two variants of this approach: either (1) a fixed number (ma) of inverted lists are scanned or (2) inverted lists are scanned until the total number of short codes scanned reaches a given threshold. Note that multiple assignment is for queries only, database vectors are always stored in a single inverted list.

Different indexing techniques have been proposed to partition the database and retrieve the relevant inverted lists at query time. We detail the two most popular indexing approaches (IVFADC [39], Multi-D-ADC [9]), and quickly review less popular approaches. Because they avoid scanning the whole database, inverted indexes combined with product quantization offer shorter response times than product quantization alone. More surprisingly, some indexing techniques (in particular IVFADC and Multi-D-ADC) offer better accuracy than product quantization alone, despite the fact that only a fraction of the database is scanned. For these reasons, these techniques are often preferred to exhaustive search in practical scenarios.

Simple inverted index (IVFADC) [39] Vector quantizers can be used to encode vectors as short codes, but they can also be used to partition the database. IVFADC (Inverted File with Asymmetric Distance Computation) uses a vector quantizer with K centroids to partition the vector space into K Voronoi cells. Each centroid and its corresponding Voronoi cell are assigned an integer identifier between 0 and $K - 1$. This approach is similar to the Bag of Features presented in Section 2.1, except that instead of storing high-dimensional vectors in inverted lists, short codes produced by a product quantizer are stored. The partitioning quantizer is named index quantizer,

and denoted q_i . Before they are encoded into short codes, the residual $r(x)$ of each database vector x is computed:

$$r(x) = x - q_i(x)$$

The residual $r(x)$ of an high-dimensional vector can therefore be interpreted as its relative position to its closest centroid in the index quantizer q_i . Each residual $r(x)$ is then encoded into a short code using a product quantizer pq , and added to the proper inverted list. In this manner, if vector x lies in the Voronoi cell the identifier of which is 3, its residual will be encoded and added to the inverted list the identifier of which is 3. IVFADC uses two quantizers for two different purposes: (1) a vector quantizer to partition the database and (2) a product quantizer to encode high-dimensional vectors as short codes. Residuals $r(x)$ typically have a lower entropy than the original vectors x . Therefore, encoding residuals instead of the original vectors leads to a lower quantization error, which in turn increases accuracy. To answer a query, the ma closest centroids to the query vector are searched in the index quantizer q_i codebook, and the corresponding inverted lists are scanned. For a dataset of 1 billion SIFT vectors, two typical configurations are: (1) using an index quantizer q_i with $K = 2^{13}$ centroids, and scanning $ma = 64$ inverted lists at query time and (2) using an index quantizer q_i with $K = 2^{13}$ centroids, and scanning $ma = 64$ inverted lists at query time. In configuration 1, about $1/128^{\text{th}}$ of the database is scanned to answer a query while in configuration 2, $1/1024^{\text{th}}$ of the database is scanned. Both configurations offer similar accuracy, but configuration 2 is faster as less short codes are scanned. In general, finer inverted indexes allow scanning a smaller fraction of the dataset while retaining the same accuracy, and thus allow faster response times. However, retrieving the ma most relevant inverted lists requires computing the distance of the query vector to each of the K centroids of the index quantizer in order to find the ma closest ones. This operations becomes costly as K increases. Above $K = 2^{16}$ centroids, the increased cost of matching the query vector against the index quantizer codebook starts to outweigh the benefit of scanning less short codes.

Multi-index (Multi-D-ADC) [9] Multi-D-ADC overcomes this issue. More specifically, Multi-D-ADC makes it possible to build fine indexes, e.g., with more than $K = 2^{16}$ inverted lists, without making the retrieval of inverted lists costly during query processing. Multi-D-ADC therefore offers faster response times than IVFADC by scanning an even smaller fraction of the dataset. Like IVFADC, Multi-D-ADC uses two quantizers: one for the inverted index, and one to encode vectors as short codes. However, instead of using a vector quantizer q_i for the index, Multi-D-ADC uses a product quantizer for the index. Multi-D-ADC thus uses two product quantizers: (1) an index product quantizer pq_i to partition the database, (2) a product quantizer pq to encode high-dimensional vector as short codes. These two product quantizers typically have different parameters. The product quantizer pq used for vector encoding typically has $m = 4$ to $m = 16$ sub-quantizers (with $k = 2^b$ centroids each, $b = 4$ to $b = 16$). By contrast, in [9], the authors show that the index product quantizer should have only $m = 2$ sub-quantizers ($K \approx 2^{14}$ centroids each). The $m = 2$ sub-quantizers, q_1 and q_2 , of the index product quantizer have two distinct codebooks $\mathcal{C}_1 = \{c_i^1\}_{i=0}^K$ and $\mathcal{C}_2 = \{c_i^2\}_{i=0}^K$. The index product quantizer pq_i partitions the vector space in K^2 cells (e.g., $K^2 = 2^{28}$ cells for $K = 2^{14}$). Each cell is defined by the concatenation of two centroids (c_i^1, c_j^2) $i \in \{0, \dots, k-1\}$,

$j \in \{0, \dots, k-1\}$, and the corresponding inverted list is identified by the (i, j) tuple. Like IVFADC, Multi-D-ADC allows encoding the residuals of high-dimensional vectors instead of the original vectors, as follows: $r(x) = x - (c_i^1, c_j^2)$. Computing the distance between the query vector and all (c_i^1, c_j^2) $i \in \{0, \dots, K-1\}$, $j \in \{0, \dots, K-1\}$ to retrieve the most relevant inverted lists would require K^2 distance computations. Multi-D-ADC would therefore bring no benefit over IVFADC and using more than $K^2 = 2^{16}$ centroids would still be impractical. Instead, the multi-sequence algorithm, introduced in [9] takes advantage of the properties of the index product quantizer pq_i to retrieve relevant inverted lists while requiring $2K$ distance computations only. The multi-sequence algorithm first computes the distance of the first half of the query vector y , denoted y^1 to all centroids of $\mathcal{C}_1$ and the distance of the query vector y^2 to all centroids of $\mathcal{C}_2$. The codebooks $\mathcal{C}_1$ copied and sorted by increasing distance from y^1 and y^2 respectively. The multi-sequence iterates over the two codebooks by increasing $d(y^1, c_i^1) + d(y^2, c_j^2)$ distance, where $d(y^1, c_i^1)$ is the distance from y^1 to the i -th centroid of $\mathcal{C}_1$ (respectively for $d(y^2, c_j^2)$). The inverted lists (i, j) are then marked for scanning, until enough vectors have been gathered.

Using a Multi-D-ADC with $K = 2^{14}$ (thus $K = 2^{28}$ cells in total) and scanning $1/50000^{th}$ of the database gives the same accuracy as using an IVFADC with $K = 2^{16}$ cells and scanning $1/1024^{th}$ of the database. Even if the multi-sequence algorithm is more efficient than computing the distance to all (c_i^1, c_j^2) $i \in \{0, \dots, k-1\}$, $j \in \{0, \dots, k-1\}$, its cost is not negligible. Consequently, if Multi-D-ADC offers a decrease in response time over IVFADC, this decrease is not proportional to the decrease in code scanned i.e., Multi-D-ADC is not 50 times faster in the aforementioned case even if 50 times less short codes are scanned.

Other approaches Other approaches have been proposed to partition the database into inverted lists. In [9], the authors explore the idea of using KD-trees to partition the database. They however show that Multi-D-ADC offers better performance in all cases. The use of a variant of LSH to build the inverted index has also been considered [58]. The major flaw of this approach is that it causes a four times increase in memory use, partially offsetting the benefit of product quantization. For this reason, this approach is not very popular.

2.3.3 ANN Search

ANN search in a database of vectors encoded as short codes takes three steps: *Index*, which involves retrieving the most relevant inverted lists from the index; *Tables*, which involves computing lookup tables used for fast distance computation and *Scan*, which involves computing the distance between the query vector and short using the previously computed lookup tables. Obviously, the step *Index* is only required for non-exhaustive search, and is skipped in the case of exhaustive search (Algorithm 1). We detail these three steps in the following paragraphs.

Index In this step, the most relevant lists to the query vector are retrieved from the inverted index. The exact procedure depends on the type of inverted index used; we presented the most common ones in Section 2.3.2. Therefore the implementation of the `INDEX_GET_LIST` function depends on the type of inverted index used (Algorithm 1). If needed, the residual $r(y)$ of the query vector y is also computed. In

Algorithm 1 ANN Search

```

1: function NNS( $\{\mathcal{C}^j\}_{j=0}^m, database, y$ )
2:    $list, y' \leftarrow \text{INDEX\_GET\_LIST}(database, y)$  ▷ Index Step
3:    $\{D^j\}_{j=0}^m \leftarrow \text{COMPUTE\_TABLES}(y', \{\mathcal{C}^j\}_{j=0}^m)$  ▷ Tables Step
4:   return SCAN( $list, \{D^j\}_{j=0}^m$ ) ▷ Scan Step

5: function SCAN( $list, \{D^j\}_{j=0}^m$ )
6:    $neighbors \leftarrow \text{binheap}(r)$  ▷ binary heap of size  $r$ 
7:   for  $i \leftarrow 0$  to  $|list| - 1$  do
8:      $c \leftarrow list[i]$  ▷  $i$ th code
9:      $d \leftarrow \text{ADC}(p, \{D^j\}_{j=0}^m)$ 
10:     $neighbors.add((i, d))$ 
11:  return  $neighbors$ 

12: function ADC( $c, \{D^j\}_{j=0}^{m-1}$ )
13:   $d \leftarrow 0$ 
14:  for  $j \leftarrow 0$  to  $m$  do
15:     $index \leftarrow c[j]$ 
16:     $d \leftarrow d + D^j[index]$ 
17:  return  $d$ 

```

practice, multiple inverted lists are retrieved from the inverted index. However, for the sake of simplicity, this section describes the search process for a single inverted list. For ma inverted lists, each operation should be repeated ma times: ma inverted lists are retrieved from the inverted index, ma residuals are computed, and ma inverted lists are scanned. In the remainder of this section, $y' = r(y)$ if an inverted index is used. Otherwise, $y' = y$.

Tables In this step, a set of m lookup tables are computed $\{D^j\}_{j=0}^m$, where m is the number of sub-quantizers of the product quantizer. The j th lookup table D_j is composed of the distances between the j th sub-vector of y' and all centroids of the j th sub-quantizer:

$$D^j = \left(\|y'^j - \mathcal{C}^j[0]\|^2, \dots, \|y'^j - \mathcal{C}^j[k-1]\|^2 \right) \quad (2.2)$$

We do not detail the COMPUTE_TABLES function in Algorithm 1, but it would correspond to an implementation of Equation 2.2.

Scan In this step, the inverted list retrieved from the inverted index is scanned for nearest neighbors (SCAN function, Algorithm 1). This requires computing the distance between the query vector y' and all short codes in the inverted list (Algorithm 1, line 7). Squared Euclidean distances are used for ANN search instead of Euclidean distances because Squared Euclidean distances avoid a costly square root computation and preserve the order. Asymmetric Distance Computation (ADC) uses the lookup tables generated in the previous step to compute the distance between the query vector y' and a short code c , as follows:

$$\text{adc}(y', c) = \sum_{j=0}^{m-1} D^j[c[j]] \quad (2.3)$$

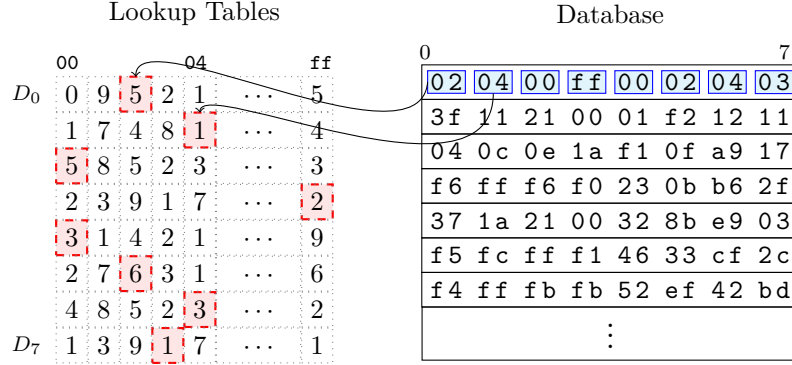


Figure 2.4: Asymmetric Distance Computation (ADC)

The ADC function is an implementation of Equation 2.3. Figure 2.4 also illustrates ADC procedure for short codes generated by a 8×8 product quantizer ($m = 8$, $b = 8$). Each centroid index $c[j]$ constituting the short code c (02, 04, etc.) is used to look up a value in the j th lookup table (D_0 , D_1 , etc.). All looked up values (in red) are then summed to obtain the asymmetric distance.

Substituting Equation 2.2 into Equation 2.3, we have:

$$\text{adc}(y', c) = \sum_{j=0}^{m-1} \left\| y'^j - \mathcal{C}^j[c[j]] \right\|^2 \quad (2.4)$$

In Equation 2.4, $\left\| y'^j - \mathcal{C}^j[c[j]] \right\|^2$ is the squared distance between the j th sub-vector of y' and the j th centroid referenced by code c . ADC therefore sums the distances of the sub-vectors of y' to the centroids referenced by the short code c in the m sub-spaces of the product quantizer. Because it splits high-dimensional vectors into m sub-vectors, a product quantizer generates m *orthogonal* sub-spaces. This makes it possible to compute partial distances in each sub-space and to sum them. When the number of codes in cells is large compared to k , the number of centroids of sub-quantizers, using lookup tables avoids computing $\left\| y'^j - \mathcal{C}^j[i] \right\|^2$ for the same i multiple times. Lookup tables therefore provide a significant speedup.

During the scan of an inverted list, nearest neighbors are stored in a binary heap of size r (Algorithm 1, line 6), where r is the number of requested nearest neighbors. Nearest neighbors are stored in the binary heap as (i, d) pairs, where i is the index of the short code and d is the distance to the query vector. The binary heap holds the r pairs with the lowest d . If a new pair is added (Algorithm 1, line 10) and the binary heap is not full, the pair is added. If a new pair is added and the binary heap is full, the pair is added only if its distance d is lower than the distance of the pair with the highest distance in the binary heap. The pair with the largest distance is removed from the heap, and the new pair is added. Adding a pair to the binary heap takes $O(\log r)$ operations. Checking if a pair has a lower distance than the pair with the highest distance takes $O(1)$ operations.

2.3.4 Product Quantization Derivatives

Optimized Product Quantization (OPQ) [24]¹ A product quantizer quantizes a d -dimensional vector by splitting it into m sub-vectors, and quantizing each sub-vector independently (Section 2.3.1). The i th sub-vector comprises the dimensions $i \cdot d/m$ to $(i + 1) \cdot d/m$. In this manner, a product quantizer partitions the d -dimensional vector spaces into m orthogonal sub-spaces, each sub-space comprising a subset of the dimensions the original vector space. Intuitively, we see that this sub-space decomposition may not be optimal, for instance if vectors have different variances across their dimensions. Some sub-spaces may then carry more information than others. Moreover, this sub-space decomposition prevents taking advantage of correlations between dimensions that are not in the same sub-space to decrease quantization error.

Optimized product quantization overcomes these issues by multiplying a vector $x \in \mathbb{R}^d$ by an orthogonal matrix $R \in \mathbb{R}^{d \times d}$ before quantizing them with a product quantizer. An optimized product quantizer `opq` thus maps a vector $x \in \mathbb{R}^d$ as follows:

$$\text{opq}(x) = \text{pq}(Rx),$$

such that $R^T R = I$ and `pq` is a product quantizer.

An orthogonal matrix R can represent any permutation of dimensions. Thus, the matrix R determines the decompositions of the d -dimensional space in m sub-spaces of equal dimensionality. The matrix R is learned jointly with the codebooks $\{C_j\}_{j=0}^{m-1}$ to minimize quantization error. In [24], the authors train R and $\{C_j\}_{j=0}^{m-1}$ by alternating between two steps (A) and (B) to achieve a locally optimal solution. Step (A) involves fixing R and optimizing $\{C_j\}_{j=0}^{m-1}$, by running a single k-means step in each sub-space. Step (B), involves fixing $\{C_j\}_{j=0}^{m-1}$ and optimizing R . For fixed codebooks $\{C_j\}_{j=0}^{m-1}$, R has a closed-form optimal solution obtained through Singular Value Decomposition (SVD). In most cases, the algorithm converges after less than 100 iterations, where one iteration involves running Step (A) followed by Step (B).

Optimized product quantizers can be used to encode vectors as short codes, exactly like product quantizers (Section 2.3.1). Like product quantizers, optimized product quantizers divide the vector space in sub-spaces that are orthogonal. Therefore, ADC (Section 2.3.3) can be used to compute the distance between an high-dimensional vector y and short codes generated by an optimized product quantizer. If short codes were generated by an optimized product quantizer, it is necessary to multiply y by the matrix R before computing tables. Distances to short codes can then be computed by performing m table lookups. Like product quantizers, optimized product quantizers can be combined with inverted indexes. Like IVFADC (Section 2.3.2), IVFOADC [9] uses a vector quantizer q_i to build an inverted index, but uses an optimized product quantizer to encode vectors instead of a product quantizer. Multi-D-ADC uses two product quantizers, one to build the inverted and the other to encode vectors as short codes. OMulti-D-O-ADC replaces both product quantizers by optimized product quantizers [9].

1. Two research teams concurrently discovered a similar generalization of product quantization. One team named this generalization "Optimized Product Quantization" [23], while the other used the term "Cartesian K-Means" [46], and both terms are commonly used in the literature. Both papers were originally published at CVPR'13. In this thesis, we refer to this generalization of product quantization as optimized product quantization.

Additive Quantization (AQ) [8] Both product quantization and optimized product quantization represent a d -dimensional vector x as a *concatenation* of d/m -dimensional centroids i.e., $\text{pq}(x) = (\mathcal{C}^0[i_0], \dots, \mathcal{C}^{m-1}[i_{m-1}])$. On the contrary, additive quantization represents a d -dimensional vector x as a sum of d -dimensional centroids:

$$\text{aq}(x) = \mathcal{C}^0[i_0] + \dots + \mathcal{C}^{m-1}[i_{m-1}]$$

Like product quantization, additive quantization can be used to encode vectors as short codes of $m \cdot \log_2(k)$ bits by concatenating centroids indexes, i.e., $\text{enc}(x) = (i_0, \dots, i_m)$. Additive quantization therefore uses more information than product quantization to represent a high-dimensional vector: m d -dimensional centroids for additive quantization versus m d/m -dimensional centroids for product quantization. This allows additive quantization to offer a better accuracy for nearest neighbor search than product quantization. However, additive quantization suffers from two major disadvantages. First, finding the best combination of centroids to sum from the codebooks $\{\mathcal{C}\}_{j=0}^{m-1}$ in order to encode a vector x is a hard problem. In [8], the authors propose an approximate algorithm based on Beam search but this solution remains multiple orders of magnitude more costly than the encoding process of product quantization. Additive quantization is therefore intractable for billion-size datasets. Second, computing the distance of the query vector to a short code c requires $m^2/2$ table lookups and $m^2/2$ additions, while a distance computation requires m tables lookups and m additions with product quantization. In most cases, nearest neighbor search is more than 3 times slower with additive quantization than with product quantization.

Tree Quantization (TQ) [10] Tree quantization is similar to additive quantization in the sense that it represents a high-dimensional vector x by a sum of m d -dimensional centroids, $\text{tq}(x) = \mathcal{C}^0[i_0] + \dots + \mathcal{C}^{m-1}[i_{m-1}]$. However, it imposes a special tree structure on the codebooks. This tree structure makes the encoding process faster, making tree quantization tractable for larger datasets. Tree quantization encoding however remains much slower than product quantization encoding. In addition, with tree quantization, computing the distance of the query vector to a short code c requires $2m$ table lookups and $2m$ additions. Nearest neighbor search is about 2-2.5 times slower with tree quantization than with product quantization. Tree quantization offers a similar accuracy to additive quantization for nearest neighbor search, and therefore a higher accuracy than product quantization.

Composite Quantization (CQ) [61] Like additive quantization and tree quantization, composite quantization represents a high-dimensional vector x by a sum of m d -dimensional centroids, $\text{cq}(x) = \mathcal{C}^0[i_0] + \dots + \mathcal{C}^{m-1}[i_{m-1}]$. Additive quantization and tree quantization do not enforce any orthogonality constraint on the centroids of the codebooks i.e., $\mathcal{C}^{j'}[i_{j'}] \cdot \mathcal{C}^j[i_j]$ can be any value $\forall j' \in \{0, \dots, m-1\}, \forall j \in \{0, \dots, m-1\}, j' \neq j, \forall i_j \in \{0, \dots, k-1\}$. Product quantization and optimized product quantization represent a vector x by a concatenation of m centroids, which implies that centroids in different codebooks are orthogonal, $\mathcal{C}^{j'}[i_{j'}] \cdot \mathcal{C}^j[i_j] = 0$ if $j' \neq j$. The distinctive feature of composite quantization is that it allows some degree of non-orthogonality between centroids, but it enforces that the sum of the pairwise dot products of centroids is a constant for any code c , i.e., $\sum_{j'=0}^{m-1} \sum_{j=0}^{m-1} \mathcal{C}^{j'}[i_{j'}] \cdot \mathcal{C}^j[i_j] = \epsilon$ for $j' \neq j, \forall c = (i_0, \dots, i_{m-1})$. Since the sum of pairwise centroid dot products is constant for all codes, it can be omitted at search time. Therefore computing

the distance of the query to a short code requires m table lookups and m additions. Nearest neighbor search with composite quantization is as fast as with product quantization, and therefore much faster than with additive quantization or tree quantization. Composite quantization has been shown to offer a better accuracy than product quantization, but its accuracy has not been compared to additive quantization or tree quantization.

3 Performance Analysis

Contents

3.1	Impact of Memory Accesses	31
3.2	Issues with SIMD Implementation	35
3.3	Lessons Learned	37

All the contributions of this thesis focus on increasing the speed of the *Scan* step of ANN search (Section 2.3.3). This step consists in computing the asymmetric distance between the query vector y and all short codes in the previously selected inverted list(s). Among the different steps of ANN search (Index, Tables, Scan), the Scan step generally takes the most time. ANN search is trivial to parallelize at the query level by executing different ANN queries on the different cores of a multi-core CPU. The Scan step itself is also trivial to parallelize by scanning a different chunk of the inverted list on each core. Therefore, we focus on the more challenging task of improving the *single-core* performance of the Scan step. In this section, we analyze the factors that limit the performance of the Scan step, and we show that memory accesses are the primary bottlenecks. We also show that the structure of the algorithm prevents an efficient SIMD implementation.

3.1 Impact of Memory Accesses

Breakdown of operations From Algorithm 1 (Section 2.3.3) we observe that each Asymmetric Distance Computation (ADC) requires the following operations:

- m memory accesses to load centroid indexes $c[j]$ (Algorithm 1, line 15) [mem1]
- m memory accesses to load $D^j[index]$ values from lookup tables (Algorithm 1, line 16) [mem2]
- m additions (Algorithm 1, line 16)

These operations are the *only* instructions executed by the CPU to perform an ADC. Since the parameter m is a small constant (typical $m = 4 - 32$) known at compile time, compilers are able to unroll the loop (Algorithm 1, line 14). The CPU therefore does not have to maintain the loop index and execute jump instructions at runtime. It is crucial for performance that this loop is unrolled: in our experiments, when the loop is not unrolled, the scan is 3 times slower (for typical parameters: $m = 8, b = 8$). In addition, as the function ADC is small and called repeatedly (Algorithm 1, line 9),

3. Performance Analysis

Table 3.1: Properties of different types of memory (Nehalem-Haswell)

Mem. Type	Size	Latency (CPU cycles)
L1 cache	32 KiB	4-5
L2 cache	256 KiB	11-13
L3 cache	2-3 MiB $\times$ core_count	25-40
RAM	4-512 GiB	100-300

so compilers are able to inline the function call. Therefore, there is no function call overhead at runtime. In this thesis, we take this implementation (Algorithm 1) with compiler optimizations enabled (i.e., loop unrolling, function call inlining) as a baseline and seek to improve its performance.

Cost of memory accesses When evaluating the complexity of an algorithm, memory accesses are often overlooked. In the original paper on product quantization [39], the authors report that "only m additions are required per distance calculation", forgetting about memory accesses. However, memory accesses are costly operations on a CPU. While an additions does not take more than 1 CPU cycle, memory accesses take 4 CPU cycles in the best case. Table 3.1 details the latency of memory accesses depending on the cache level, as well as the sizes of the different caches. These figures are valid for Intel CPU architectures from Nehalem (released in 2008) to Haswell (released in 2013). In the list of operations of the previous paragraph, we distinguished between two classes of memory accesses: *mem1* and *mem2*, as they may hit different cache levels. *Mem1* accesses always hit the L1 cache thanks to the *hardware prefetchers* included in CPUs. Hardware prefetchers are able to detect sequential memory accesses, and prefetch data to the L1 cache. We access $c[j]$ values sequentially. We first access $c[0]$, where c is the first code in the database, then $c[1]$ until $c[m-1]$. Next we perform the same accesses on the second code in the database, until we have iterated on all codes in the inverted list. On the contrary, *mem2* accesses may hit the L1 or L3 depending on the b parameter of the product quantizer.

Impact of m and b parameters In Section 2.3.1, we mentioned that the m and b parameters of a product quantizer impact: (1) the memory use of the database, (2) the accuracy of nearest neighbor search and (3) the speed of nearest neighbor search. An $m \times b$ product quantizer has a total of $2^{m \times b}$ centroids, and encodes each high-dimensional vector into a code of $m \times b$ bits. The higher the $m \times b$ product, the higher the number of centroids of the product quantizer, and therefore the higher the accuracy of nearest neighbor search. In Table 3.2, we use Recall@100 as accuracy measure. However, the higher the $m \times b$ product, the larger the codes, and therefore the higher the memory use of the database. In practice, codes of $m \times b = 64$ bits (2^{64} centroids) or $m \times b = 128$ bits (2^{128} centroids) are used. There has been a recent interest for codes of $m \times b = 32$ bits to achieve extreme compression ratios, although they offer a somewhat lower accuracy (Table 3.2). For a database of 1 million of 128-dimensional SIFT vectors (SIFT1M), 64-bit codes offer a satisfactory accuracy (Table 3.2), but 128-bit codes are necessary for larger databases or vectors of higher dimensionality.

Table 3.2: ANN search speed and accuracy (SIFT1M)

$m \times b$	Tables size	Cache	Recall@100	Tables time	Scan time
32-bit codes					
4×8	4 KiB	L1	59.0%	0.004 ms	1.3 ms
2×16	512 KiB	L3	79.0%	0.58 ms	2.7 ms
64-bit codes					
16×4	1 KiB	L1	83.1%	0.001 ms	6 ms
8×8	8 KiB	L1	92.2%	0.011 ms	2.6 ms
4×16	1 MiB	L3	96.5%	0.82 ms	7.9 ms
128-bit codes					
32×4	2 KiB	L1	96.5%	0.002 ms	12 ms
16×8	16 KiB	L1	99.8%	0.009 ms	5.4 ms
8×16	2 MiB	L3	99.8%	1.5 ms	17 ms

For a fixed code size, and thus a fixed $m \times b$ product (e.g., $m \times b = 64$), the parameters m and b impact the accuracy and speed of nearest neighbor search. We observe that a large b and therefore a small m , increases accuracy, but also increases scan time (Table 3.2). Thus, a 4×16 product quantizer offers a better accuracy than 8×8 product quantizer, but also incurs a higher scan time. This seems counter-intuitive: as small m means less operations per distance computation, and should therefore result in lower scan time. Each distance computation requires m *mem1* accesses, m *mem2* accesses and m additions. A 4×16 product quantizer ($m = 4$) requires less operations per distance computation than a 8×8 product quantizer ($m = 8$), but still results in a higher scan time. This is because a 4×16 product quantizer leads to larger $\{D^j\}_{j=0}^{m-1}$ lookup tables. The size of the lookup tables is $m \cdot 2^b \cdot \text{sizeof(float)} = m \cdot 2^b \cdot 4$ bytes (m lookup tables of 2^b floating-point values each). Larger lookup tables need to be stored in slower cache levels, which increases the cost of *mem2* accesses. For a 4×16 product quantizer, lookup tables have to be stored in the L3 cache (slowest cache) while they fit the L1 cache (fastest cache) for a 8×8 product quantizer. Overall, the increase in the cost of *mem2* access outweighs the decrease in the number of operations. This results in a threefold increase in scan time for 4×16 product quantizers compared to 8×8 product quantizers.

Table 3.2 reports the properties of different product quantizers only for a few values of b , namely $b = 4$, $b = 8$ and $b = 16$. First, training sub-quantizers (Section 2.3.1) with more than 2^{16} centroids is not tractable, thus $b > 16$ is not achievable. Theoretically, any $b \leq 16$ would be possible, e.g., a 5×13 product quantizer ($b = 13$, 65-bit codes), $c[j]$ centroid indexes need to be accessed individually (Algorithm 1, line 15). CPUs can natively address values of 8, 16, 32 or 64 bits in memory. Accessing integers of other sizes, e.g., 13-bit integers, requires additional bit shifting and bit masking operations, which increase scan time. Therefore, b values other than $b = 8$ or $b = 16$ are not suitable. We however added the properties of product quantizers for $b = 4$, to illustrate that scan time does not continuously decreases with b . Once lookup tables are small enough to fit the L1 cache, it is not useful to further shrink them. This does not make *mem2* accesses less costly, but results in a higher number

3. Performance Analysis

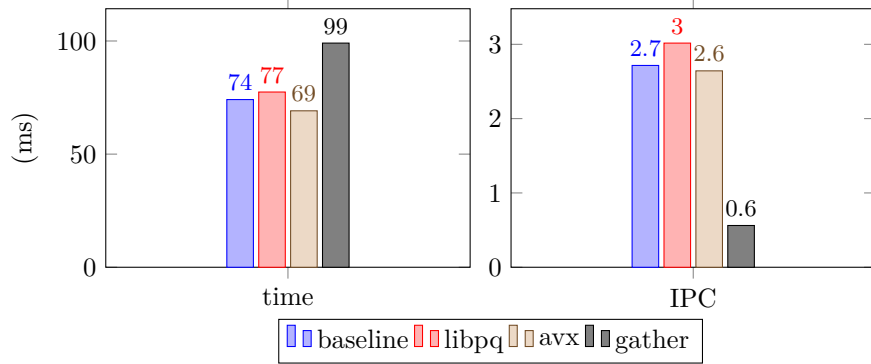


Figure 3.1: Scan times (25M vectors) and Instructions Per Cycle (IPC)

m of operations.

In practice, 8-bit sub-quantizers ($b = 8$) are used in almost all publications on product quantization as they offer a good tradeoff between speed and accuracy. As 64-bit codes, and 8-bit sub-quantizers are very common, we focus exclusively on 8×8 product quantizers ($m = 8, b = 8$, 64-bit codes) in the remainder of this section.

Libpq scan implementation We use hardware performance counters to study the performance of different scan procedure implementations experimentally (Figure 3.1, Figure 3.2). For all implementations, the number of cycles with pending load operations (cycles w/ load) is almost equal to the number of cycles, which confirms the scan procedure is a memory-intensive. We also measured the number of L1 cache misses (not shown on Figure 3.2). L1 cache misses represent less than 1% of memory accesses for all implementations, which confirms that both *mem1* and *mem2* accesses hit the L1 cache. The baseline implementation of the scan procedure (Algorithm 1) performs 16 L1 loads per scanned vector: $m = 8$ *mem1* accesses and $m = 8$ *mem2* accesses. The authors of [39] distribute the libpq library¹, which includes an optimized implementation of the ADC Scan procedure. We obtained a copy of libpq under a commercial licence. Rather than loading $m = 8$ centroid indexes of $b = 8$ bits each (*mem1* accesses), the libpq implementation of the scan procedure loads a 64-bit word into a register, and performs 8-bit shifts to access individual centroid indexes. This allows reducing the number of *mem1* accesses from 8 to 1. Therefore, the libpq implementation performs 9 L1 loads per scanned vector: 1 *mem1* access and 8 *mem2* accesses. However, overall, the libpq implementation is slightly slower than the baseline implementation on our Haswell processor. Indeed, the increase in the number of instructions offsets the increase in IPC (Instructions Per Cycle) and the decrease in L1 loads. It is however likely that this optimization was useful on older processors. Before, the Sandy Bridge architecture, Intel CPUs had a single memory read port, while newer CPUs have two read ports. This means that current CPUs can perform two concurrent cache reads, while older CPUs can only perform one. Therefore, reducing the number of memory reads may bring a higher benefit on older CPUs.

1. <http://people.rennes.inria.fr/Herve.Jegou/projects/ann.html>

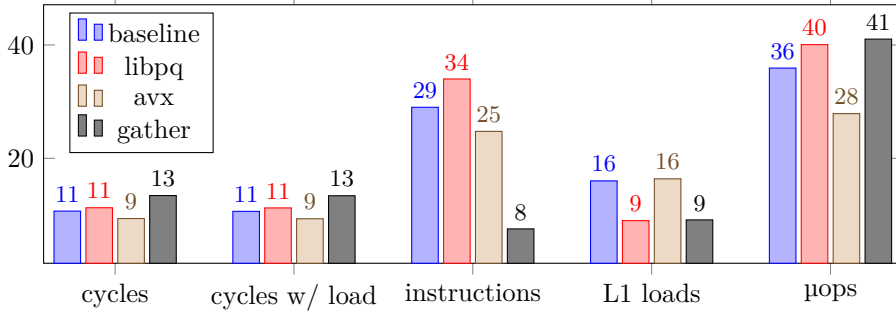


Figure 3.2: Scan performance counters (per scanned code)

3.2 Issues with SIMD Implementation

Introduction to SIMD In addition to memory accesses, each distance computation requires m additions. SIMD instructions are commonly used to increase the performance of algorithms performing arithmetic computations. Therefore, we evaluate the applicability of SIMD instructions to reduce the number of instructions and CPU cycles devoted to additions. SIMD instructions perform the same operation, e.g., additions, on multiple data elements in one instruction. To do so, SIMD instructions operate on wide registers. SSE SIMD instructions operate on 128-bit registers, while more recently introduced AVX SIMD instructions operate on 256-bit registers [36]. SSE instructions can operate on 4 floating-point *ways* (4×32 bits, 128 bits) while AVX instructions can operate on 8 floating-point *ways* (8×32 bits, 256 bits). In our case, AVX instructions offer only slightly higher performance than SSE instructions. As both instruction sets yield similar results, we only report results for the AVX implementation. We show that the structure of the scan procedure (Algorithm 1) prevents an efficient use of SIMD instructions.

Implementation details To enable the use of fast *vertical* SIMD additions, we compute the asymmetric distance between the query vector and 8 database vectors at a time, designated by the letters a to h . We still issue 8 addition instructions, but each instruction involves 8 different vectors, as shown on Figure 3.3. Overall,

(way 0)				(way 7)	
$D^0[a[0]]$	$D^0[b[0]]$	$D^0[c[0]]$	$D^0[d[0]]$	...	$D^0[h[0]]$
+	+	+	+		+
$D^1[a[1]]$	$D^1[b[1]]$	$D^1[c[1]]$	$D^1[d[1]]$	...	$D^1[h[1]]$
+	+	+	+		+
$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$
$D^7[a[7]]$	$D^7[b[7]]$	$D^7[c[7]]$	$D^7[d[7]]$	...	$D^7[h[7]]$

Figure 3.3: SIMD vertical add

3. Performance Analysis

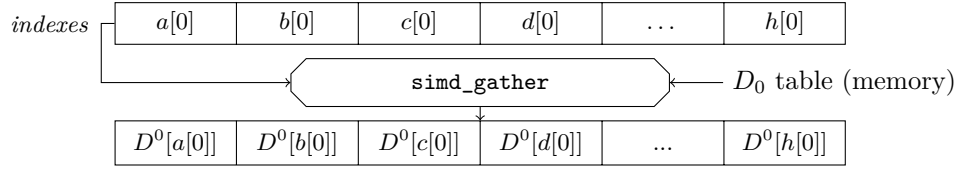


Figure 3.4: SIMD gather

the number of instructions devoted to additions is divided by 8. However, the gain in cycles brought by the use of SIMD additions is offset by the need to set the ways of SIMD registers one by one. SIMD processing works best when all values in all ways are *contiguous* in memory and can be loaded in one instruction. Because they were looked up in a table, $D_0[a[0]]$, $D_0[b[0]]$, $\dots$, $D_0[h[0]]$ values are not contiguous in memory. We therefore need to insert $D_0[a[0]]$ in the first way of the SIMD register, then $D_0[b[0]]$ in the second way etc. In addition to memory accesses, doing so requires many SIMD instructions, some of which have high latencies. Overall, this offsets the benefit provided by SIMD additions. This explains why algorithms relying on lookup tables, such as ADC Scan, hardly benefit from SIMD processing. Figure 3.1 and Figure 3.1 show that the AVX implementation of ADC Scan requires slightly less instructions than the naive implementation, and is only marginally faster.

SIMD gather To tackle this issue, Intel introduced a **gather** SIMD instruction (AVX2 instruction set) in its latest architecture, Haswell [36, 37]. Given an SIMD register containing 8 indexes and a table stored in memory, **gather** looks up the 8 corresponding elements from the table and stores them in a register, in just one instruction. This avoids having to use many SIMD instructions to set the 8 ways of SIMD registers. Figure 3.4 shows how **gather** can be used to look up 8 values in the first distance table (D_0). To efficiently use **gather**, we need $a[0], \dots, h[0]$ to be stored contiguously in memory, so that they can be loaded in one instruction. To do so, we transpose the memory layout of the inverted list. We divide each inverted list in blocks of 8 codes ($a, \dots, h$), and transpose each block independently. In a transposed block (Figure 3.5), we store the first components of 8 vectors contiguously ($a[0], \dots, h[0]$), followed by the second components of the same 8 vectors ($a[1], \dots, h[1]$) etc., instead of storing all components of the first vector ($a[0], \dots, a[7]$), followed by the components of the second vector ($b[0], \dots, b[7]$). This also allows to reduce the number of *mem1* accesses from 8 to 1, similarly to the libpq

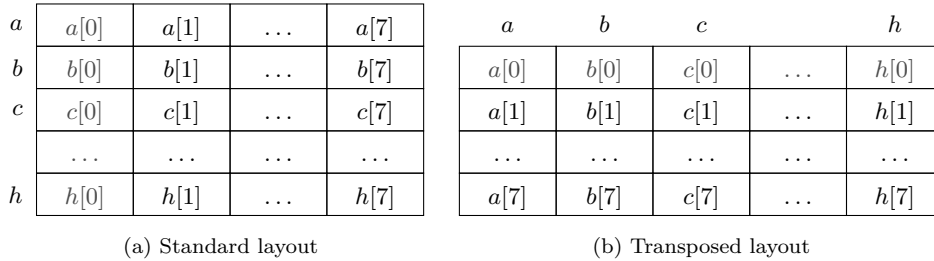


Figure 3.5: Standard layout and transposed layout

implementation. This transposition is performed online, and does not increase response time. Moreover, this transposition is cheap and does not notable increase the database creation time.

However, the gather implementation of ADC Scan is slower than the naive version (Figure 3.2), which can be explained by several factors. First, even if it consists of only one instruction, **gather** performs 1 memory access for each element it loads, which implies suffering memory latencies. Second, at the hardware level, **gather** executes 34 μops ² where most instructions execute only 1 μop . Figure 3.2 shows that the gather implementation has a low instructions count but a high μops count. For other implementations, the number of μops is only slightly higher than the number of instructions. It also has a high latency of 18 cycles and a throughput of 10 cycles, which means it is necessary to wait 10 cycles to pipeline a new **gather** instruction after one has been issued. This translates into poor pipeline utilization, as shown by the very low IPC of the gather implementation (Figure 3.2). In its documentation, Intel acknowledges that **gather** instructions may only bring performance benefits in specific cases [35] and other authors reported similar results [29].

3.3 Lessons Learned

We can draw three conclusions from the experiments conducted in this chapter:

- Accesses to cache resident lookup tables are the primary bottleneck limiting the performance of the ADC Scan procedure.
- Using 16-bit sub-quantizers offers a better accuracy than 8-bit sub-quantizers (for a fixed code size), but causes a threefold increase in response time. This is because using 16-bit sub-quantizers causes lookup tables to be stored in the L3 cache, instead of the L1 cache for 8-bit sub-quantizers.
- The structure of the ADC Scan algorithm prevents an efficient use of SIMD instructions, which are yet commonly used to increase performance. The main issue is that ADC Scan needs to look up values in the cache and insert them one by one in SIMD registers, which is costly.

In the remainder of this thesis, we build on these conclusions to design more efficient scan procedures.

2. Micro-operations (μops) are the basic operations executed by the processor. Instructions are sequences of μops .

4 PQ Fast Scan

Contents

4.1	Motivation	39
4.2	Presentation	40
4.2.1	Overview	40
4.2.2	Code Grouping	42
4.2.3	Minimum Tables	44
4.2.4	Quantization of Distances	45
4.2.5	Lookups in Small tables	46
4.3	Evaluation	47
4.3.1	Experimental Setup	47
4.3.2	Distribution of Response Times	48
4.3.3	Performance Counters	49
4.3.4	Impact of init and r Parameters	49
4.3.5	Impact of Distance Quantization	50
4.3.6	Impact of the Size of Inverted Lists	51
4.3.7	Large Scale Experiment	52
4.3.8	Impact of CPU Architecture	52
4.4	Discussion	53
4.4.1	Compatibility with Inverted Indexes	53
4.4.2	Applicability to Product Quantization Derivatives	54

4.1 Motivation

As it stores large databases in main memory, product quantization makes it possible to scan a large number of nearest neighbor candidates in a small amount of time. However, the ADC Scan procedure remains CPU intensive. In particular, we have shown that the performance of this scan procedure is limited by the large number of accesses to cache-resident lookup tables. Moreover, the structure of this procedure prevents an efficient use of SIMD instructions to boost performance. These limitations call for a modification of ADC Scan.

In this chapter, we introduce PQ Fast Scan, a novel scan procedure that achieves 4-6 times better performance than the conventional ADC Scan procedure, while returning the exact same results. PQ Fast Scan focuses exclusively on codes generated by 8×8 product quantizers but this is not a strong limitation, as this type of codes is used

in the vast majority of cases. The key idea behind PQ Fast Scan is to replace cache accesses by SIMD in-register shuffles. This change allows PQ Fast Scan to perform less than 2 cache accesses per distance computation, and enables an efficient SIMD implementation of additions. The main design challenge of PQ Fast Scan is that lookup tables used for distance computations are much larger (1 KiB each) than SIMD registers (128 bits). PQ Fast Scan overcomes this challenge by building small tables that fit SIMD registers. These small tables are used to compute lower bounds on distances, and prune unneeded cache accesses. More specifically, this chapter addresses the following points:

- We present the design of PQ Fast Scan. We describe the three techniques we use to build small tables: (1) code grouping, (2) minimum tables and (3) quantization of floating-point distances to 8-bit integers.
- We implement PQ Fast Scan on Intel CPUs and evaluate its performance on large datasets of high-dimensional vectors. We determine the parameters that impact its performance, and experimentally show that it achieves a 4-6 times speedup over the conventional scan procedure.
- We discuss the compatibility of PQ Fast Scan with different configurations of inverted indexes. We also review its applicability to other product quantizer configurations than 8×8 as well as its applicability to derivatives of product quantization.

4.2 Presentation

4.2.1 Overview

The key idea behind PQ Fast Scan is to use *small tables*, sized to fit SIMD registers instead of the cache-resident distance tables. These small tables are used to compute *lower bounds* on distances, without accessing the L1 cache. Therefore, lower bounds computations are fast. In addition, they are implemented using SIMD additions, further improving performance. We use lower bound computations to prune slow distance computations, which access the L1 cache and cannot benefit from SIMD additions. Figure 4.1 shows the processing steps applied to every database vector c . The $\otimes$ symbol means we discard the vector c and move to the next database vector. The *min* value is the distance of the query vector to the current nearest neighbor. Our experimental results on SIFT data show that PQ Fast Scan is able to prune more than 95% of distance computations.

Lower bound computations rely on SIMD in-register shuffles (`pshufb` instruction), which are key to PQ Fast Scan performance. Like cache accesses (`mov`), or SIMD gather operations (`gather`), SIMD in-register shuffles can be used to lookup values in tables. SIMD in-register shuffles however offer much better performance (Table 4.1). Using cache accesses allow performing 1 lookup at once, with a latency of 4-5 cycles (lookup tables in L1 cache). SIMD gather has a higher level of parallelism (8 lookups at once), but has a much higher latency (18 cycles). In addition, it has a throughput of 10 cycles, meaning the CPU has to wait 10 cycles before issuing another `gather` instruction. This makes it difficult to pipeline instructions efficiently, reducing performance. On the other hand, SIMD in-register shuffles offer both a high level of parallelism (16 lookups at once) and a low latency (1 cycle). This comes at

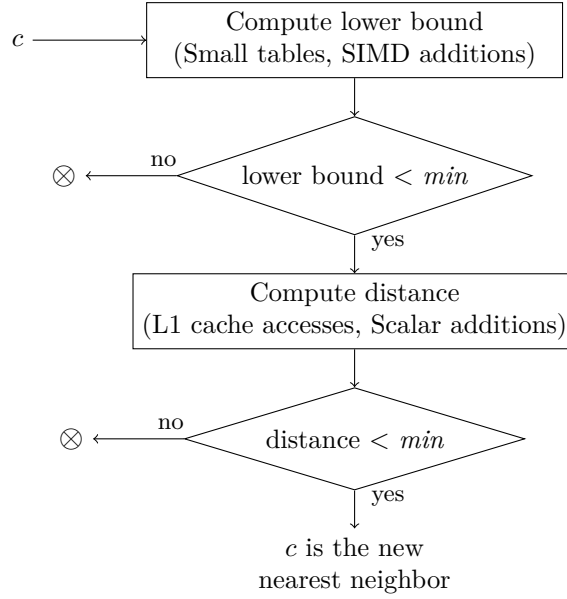


Figure 4.1: Overview of PQ Fast Scan

the price of strong constraints on lookup tables. When using caches accesses or SIMD gather, lookup tables are stored in memory. Therefore, this is virtually no limit on their size, although it should not exceed the size of the L1 cache for best performance. Moreover, they allow looking up 32-bit floating point values. By contrast, when using SIMD in-register shuffles, lookup tables must be stored in SIMD registers, which limit their size to 128 bits. There are different variants of SIMD in-register shuffles allowing either lookups in tables of 4 element of 32 bits (**pshufd**) or lookups in tables of 16 elements 8 bits (**pshufb**). In our case, we need tables with the most possible elements, thus we focus exclusively on **pshufb** (Figure 4.2).

To compute asymmetric distances, the conventional ADC Scan algorithm (with a PQ 8×8 quantizer) uses 8 distance tables D^j , $0 \leq j < 8$, and each distance table

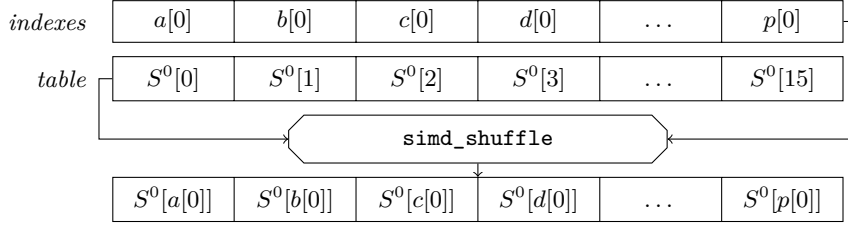
Table 4.1: Properties of table lookup techniques

Lookup technique	Par. ¹	Lat. ²	Throu. ³	μops	Table Elements	
					Count	Size
Cache accesses (mov)	1	4-5	0.5	1	No limit	32 bit
SIMD gather (gather)	8	18	10	34	No limit	32 bit
SIMD shuffle (pshufb)	16	1	0.5	1	16	8 bit

¹ Par.: Parallelism. Number of lookups performed by one instruction of this type.

² Lat.: Latency. Number of cycles to execute one instruction of this type.

³ Throu.: Throughput. Number of cycles to wait before issuing another instruction of this type.

Figure 4.2: SIMD in-register shuffle (**pshufb**)

comprises 256 elements of 32 bits. Hence, one distance table (256×32 bits) does not fit into an SIMD register, which is why we need to build small tables. Just like there are 8 distance tables D^j , $0 \leq j < 8$, we build 8 small tables S^j , $0 \leq j < 8$. Each small table S^j is stored in a distinct SIMD register and is built by applying transformations to the corresponding D^j table. To build 8 small tables suitable to compute lower bound on distances, we combine three techniques: (1) *code grouping*, (2) computation of *minimum tables* and (3) *quantization of distances*. The first two techniques, vector grouping and computation of minimum tables, are used to build tables of 16 elements (16×32 bits). The third technique, quantization of distances, is used to shrink each element to 8 bits (16×32 bits $\rightarrow 16 \times 8$ bits). We group vectors and quantize distances to build the *first four* small tables, $S^0, \dots, S^3$. We compute minimum tables and quantize distances to build the *last four* small tables, $S^4, \dots, S^7$. Figure 4.3 summarizes this process.

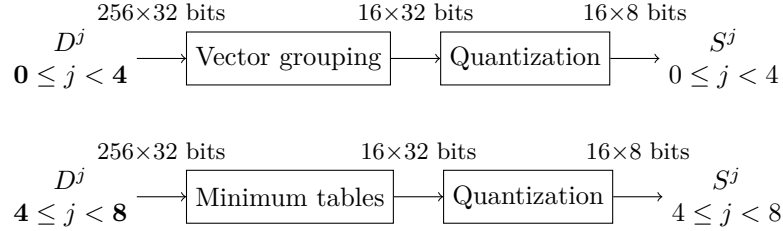


Figure 4.3: Small tables building process

4.2.2 Code Grouping

Database vectors are stored as short codes, which consist of 8 components of 8 bits (Figure 4.5a). When computing between a code and the query vector, each component is used as an index in the corresponding distance table, e.g., the 1st component is used as an index in the 1st distance table (Algorithm 1, Section 2.3.3). The key idea behind *code grouping* is to group vectors such that all codes belonging to a *group* hit the same *portion* of 16 elements of a distance table.

We focus on the first distance table, D^0 . We group vectors on their first component and divide the D^0 table into 16 portions, of 16 elements each (Figure 4.4). All database vectors c having a first component $c[0]$ between 00 and 0f (0 to 15) will trigger lookups in portion 0 of D^0 when computing the distance. These vectors form

	portion 0				portion 1					portion 15			
	00			0f	10			1f		f0			ff
D^0	7	2	...	1	8	3	...	5	...	6	9	...	5

Figure 4.4: Portions of the first distance table

group 0. All vectors having a first component between 10 and 1f (16 to 31) will trigger lookups in portion 1 of D^0 . These vectors form the group 1. We define 16 groups in this way. Each group is identified by an integer i , and contains database vectors c such that:

$$16(i-1) \leq c[0] < 16i$$

and only requires the portion i of the first distance table, D^0 . We apply the same grouping procedure on the 2nd, 3rd and 4th components. Eventually, each group is identified by four integers (i_0, i_1, i_2, i_3) , each belonging to $[0; 16[$ and contains vectors such that:

$$16(i_0-1) \leq c[0] < 16i_0 \wedge 16(i_1-1) \leq c[1] < 16i_1 \wedge \\ 16(i_2-1) \leq c[2] < 16i_2 \wedge 16(i_3-1) \leq c[3] < 16i_3$$

Figure 4.5b shows a database where all vectors have been grouped. We can see that all vectors in the group $(3, 1, 2, 0)$ have a first component between 30 and 3f, a second component between 10 and 1f, etc. To compute the distance of the query vector to any vector of a group (i_0, i_1, i_2, i_3) , we only need a portion of D^0 , D^1 , D^2 and D^3 . Before scanning a group, we load the relevant portions of $D^0, \dots, D^3$ into 4 SIMD registers to use them as the small tables $S^0, \dots, S^3$. This process is shown by solid arrows on Figure 4.9.

01 03 02 05 06 09 04 08	0 0 0 0
3f 11 21 00 01 f2 12 11	01 03 02 05 06 09 04 08
f6 ff f6 f0 23 0b b6 2f	08 0a 0b 01 3d bc 82 d6
f5 fc ff f1 46 33 cf 2c	⋮
08 0a 0b 01 3d bc 82 d6	⋮
0e 06 02 19 b0 8e c9 13	3 1 2 0
01 02 08 04 a1 97 6d af	3f 11 21 00 01 f2 12 11
34 16 25 06 23 92 bc d1	34 16 25 06 23 92 bc d1
⋮	⋮
	f f f f
	f5 fc ff f1 46 33 cf 2c
	f6 ff f6 f0 23 0b b6 2f
	⋮

(a) Unordered vectors

(b) Grouped vectors

Figure 4.5: Code grouping

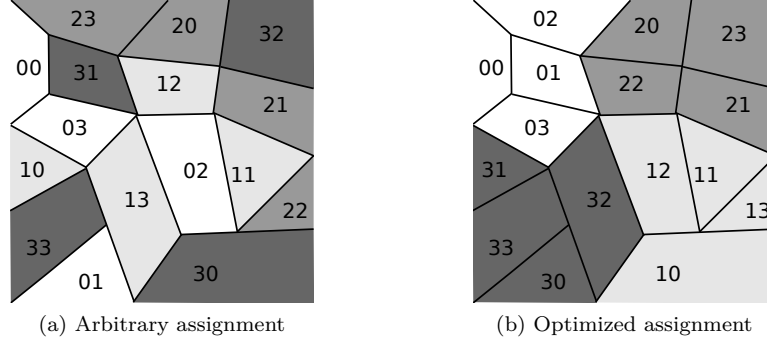


Figure 4.6: Centroid indexes assignment

We applying grouping on $g = 4$ components as it is a good tradeoff between pruning power and constraints on inverted list sizes. Grouping on $g = 4$ components allows pruning 95-99% distance computations, and thus offers a large speedup. Grouping on less than 4 components, e.g., 3 components, strongly impacts the tightness of lower bounds, and therefore strongly decreases pruning efficiency. This prevents PQ Fast Scan from offering a significant speedup, as it is not able to prune enough slow distance computations. On the other hand, grouping on more than 4 components, e.g., 5 components, does not substantially increases pruning power, while it imposes more constraints on the size of inverted lists. The number of groups created by code grouping is given by P^g , where P is the number portions in a lookup table ($P = 16$ in our case), and g is the number of components used for grouping. Therefore, the average size of a group is given by $s = n/P^g$. For best performance, s should exceed $s_{min} = 50$ codes. Before scanning a group, we load portions of lookup tables in SIMD registers, which is quite costly. If the group comprises less than 50 vectors, a large part of the CPU time is spent loading distance tables. This is detrimental to performance, as shown by our experimental results (Section 4.3.6). The minimum inverted list size $n_{min}(g)$ to be able to group vectors on g components is therefore given by $n_{min}(g) = s_{min} \cdot P^g$, and increases *exponentially* with the number of components used for grouping. For $g = 4$, the minimum inverted list size is 3 million vector ($s_{min} = 50, P = 16$).

Grouping vectors also allows decreasing the amount of memory consumed by the database by approximatively 25%. In a group, the 1st component of all vectors has the same 4 most significant bits. As we apply grouping on the 4 first components, their 2nd, 3rd and 4th components also have the same most significant bits. We can therefore avoid storing the 4 most significant bits of the 4 first components each database vector. This saves 4×4 bits = 16 bits on the 8×8 bits = 64 bits of each vector, which leads to a 25% reduction in memory consumption. Thus, on Figure 4.5b, the grayed out hexadecimal digits (which represent 4 bits) may not be stored.

4.2.3 Minimum Tables

We grouped vectors to build the first four small tables $S^0, \dots, S^3$. To build the last four small tables, $S^4, \dots, S^7$ we compute minimum tables. This involves dividing the original distance tables, $D^4, \dots, D^7$, into 16 portions of 16 elements each. We then

	Distance Tables					Minimum Tables			
	00	10		f0		0			f
D^4	2 5 ... 1	4 6 ... 8		6 1 ... 1	$\Rightarrow$	1 4 ... 1			
	4 6 ... 3	8 7 ... 3		7 8 ... 7		3 3 ... 7			
	1 1 ... 4	1 3 ... 2		0 2 ... 6		1 1 ... 0			
D^7	7 5 ... 2	5 2 ... 4		2 6 ... 0		2 2 ... 0			

Figure 4.7: Minimum tables

keep the minimum of each portion to obtain a table of 16 elements. This process is shown on Figure 4.7. Using the minimum tables techniques alone results in small tables containing low values, which is detrimental to PQ Fast Scan performance. If these values are too low, the computed lower bound is not tight, i.e., far from the actual distance. This limits the ability of PQ Fast Scan to prune costly distance computations.

To obtain small tables with higher values, we introduce an *optimized assignment* of sub-quantizer centroids indexes. Each value $D^j[i]$ in a distance table is the distance between the j th sub-vector of the query vector and the centroid with index i of the j th sub-quantizer (Section 2.3.3). When a sub-quantizer is learned, centroids indexes are assigned arbitrarily. Therefore, there is no specific relation between centroids having indexes corresponding to a portion of a distance table (e.g., centroids having indexes between 00 and 0f). On the contrary, our optimized assignment ensures that all indexes corresponding to a given portion (e.g., 00 to 0f) are assigned to centroids close to each other, as shown on Figure 4.6. Centroids corresponding to the same portion have the same background color. For the sake of clarity, Figure 4.6 shows 4 portions of 4 indexes, but in practice we have 16 portions of 16 indexes. This optimized assignment is beneficial because it is likely that a query sub-vector close to a given centroid will also be close to nearby centroids. Therefore, all values in a given portion of a distance table will be close. This allows computing minimum tables with higher values, and thus tighter lower bounds. To obtain this optimized assignment, we group centroids into 16 clusters of 16 elements each using a variant of k-means that forces groups of same sizes [50]. Centroids in the same cluster are given consecutive indexes, corresponding to one portion of a distance table. This optimized assignment of centroid indexes replaces the arbitrary assignment applied while learning sub-quantizers.

4.2.4 Quantization of Distances

The vector grouping and minimum tables techniques are used to build tables of 16 elements of 32 bits each, from the original D^j distance tables (256×32 bits). So that these tables can be used as small tables, we also need to shrink each element to 8 bits. To do so, we quantize floating-point distances to 8-bit integers. As there is no SIMD instruction to compare *unsigned* 8-bit integers, we quantize distances to *signed* 8-bit integers, only utilizing their positive range, i.e., 0-127. We quantize floating-point distances between a $qmin$ and a $qmax$ bound into $n = 127$ bins. The size of each bin

4. PQ Fast Scan

is $(qmax - qmin)/n$ and the bin number (0-126) is used as a representation value for the quantized float. All distances above $qmax$ are quantized to 127 (Figure 4.8).



Figure 4.8: Selection of quantization bounds

We set $qmin$ to the minimum value across all distance tables, which is the smallest distance we need to represent. Setting $qmax$ to the maximum possible distance, i.e., the sum of the maximums of all distance tables, results in a high quantization error. Therefore, to determine $qmax$, we find a *temporary* nearest neighbor of the query vector among the $init\%$ first vectors of the database (usually, $init \approx 1\%$) using the conventional ADC Scan procedure. We then use the distance between the query vector and this temporary nearest neighbor as $qmax$ bound. We do not need to represent distances higher than this distance because all future nearest neighbor candidates will be closer to the query vector than this temporary nearest neighbor. This choice of $qmin$ and $qmax$ bounds allows us to represent a small but relevant range of distances (Figure 4.8). Quantization error is therefore minimal, as confirmed by our experimental results (Section 4.3.5). Lastly, to avoid integer overflow issues, we use *saturated* SIMD additions.

4.2.5 Lookups in Small tables

For the sake of clarity, we do not fully describe the SIMD implementation of PQ Fast Scan. Instead, we focus on describing which small tables are used, and how they are indexed to compute lower bounds. The first four small tables, $S^0, \dots, S^3$ correspond to quantized portions of $D^0, \dots, D^3$. We load these quantized portions into SIMD registers before scanning each group, as shown by the solid arrows on Figure 4.9. Thus, two different groups use different small tables $S^0, \dots, S^3$. On the contrary, the last four small tables, $S^4, \dots, S^7$, built by computing minimum tables do not change and are used to scan the whole database. They are loaded into SIMD registers at the beginning of the scan process.

As small tables contain 16 values, they are indexed by 4 bits. Given a database vector p , we use the 4 *least* significant bits of $c[0], \dots, c[3]$ to index values in $S^0, \dots, S^3$ and the 4 *most* significant bits of $c[4], \dots, c[7]$ to index values in $S^4, \dots, S^7$. Indexes

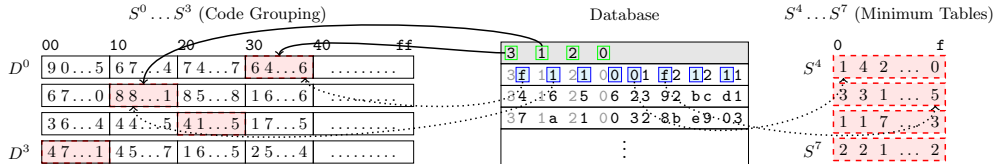


Figure 4.9: Use of small tables to compute lower bounds

are circled on Figure 4.9 (the 4 most significant bits correspond to the first hexadecimal digit, and the 4 least significant bits to the second hexadecimal digit) and lookups in small tables are depicted by dotted arrows. Lookups depicted by dotted arrows are performed using SIMD in-register shuffles. To efficiently use SIMD in-register shuffles, we need to transpose the vectors in each group, as in the gather implementation of ADC Scan (Section 3.2). However we do not show this transposition on Figure 4.9 for the sake of clarity. To compute the lower bound, we add the 8 looked up values. To decide on pruning distance computations, the lower bound is compared to the *quantized* value of *min*, the distance between the query vector and the current nearest neighbor.

4.3 Evaluation

The aim of this section is twofold: evaluating the performance of PQ Fast Scan and analyzing the parameters that influence it. We show that PQ Fast Scan outperforms ADC Scan by a factor 4-6 in common usage scenarios.

4.3.1 Experimental Setup

We implemented PQ Fast Scan in C++ using intrinsics, which allow accessing SIMD instructions from C or C++, without writing assembly code [33, 32]. Our implementation uses 128-bit SIMD instructions from the SSSE3, SSE3 and SSE2 instruction sets. We compared our implementation of PQ Fast Scan with the libpq implementation of the ADC Scan procedure, introduced in Section 3.1. On all our test platforms, we used the gcc and g++ compilers version 4.9.2, with the following compilation options: `-O3 -m64 -march=native -ffast-math`. We released our source code¹ under the Clear BSD license.

Table 4.2: Size of inverted lists used for experiments

Inverted list ID	0	1	2	3	4	5	6	7
# vectors	25M	3.4M	11M	11M	11M	11M	4M	23M
# queries	2595	307	1184	1032	1139	1036	390	2317

We evaluate PQ Fast Scan on the largest public dataset of high-dimensional vectors, ANN_SIFT1B². It consists of 3 parts: a learning set of 100 million vectors, a base set of 1 billion vectors and a query set of 10000 vectors. We restricted the learning set for the product quantizer to 10 million vectors. Vectors of this dataset are SIFT descriptors of dimensionality 128. We use two subsets of ANN_SIFT1B for experiments:

- ANN_SIFT100M1, a subset of 100 million vectors of the base set. We build an index with 8 inverted lists; each query is directed to the most relevant inverted list which is then scanned with PQ Fast Scan and ADC Scan. Table 4.2 summarizes the sizes of the different inverted lists.

1. <https://github.com/technicolor-research/pq-fast-scan>

2. <http://corpus-texmex.irisa.fr/>

4. PQ Fast Scan

Table 4.3: Response time distribution

	Mean	25%	Median	75%	95%
ADC Scan (libpq)	73.9	73.6	73.8	74.0	74.5
PQ Fast Scan	13.7	12.3	12.9	14.1	18.0
Speedup	5.4	6.0	5.7	5.2	4.1

- ANN_SIFT1B, the full base set of 1 billion vectors to test our algorithm on a larger scale.

We study the following parameters impacting the performance of PQ Fast Scan:

- *init*, the percentage of vectors kept at the beginning of the database (Section 4.2.4). Even when using PQ Fast Scan, these vectors are scanned using the conventional ADC Scan procedure to find a temporary nearest neighbor. The distance of the query vector to the temporary nearest neighbor is then used as the *qmax* value for quantization of distance tables.
- *r*, the number of nearest neighbors returned by the search process. For the sake of simplicity, we described ADC Scan and PQ Fast Scan as if they returned a single nearest neighbor. In practice, they return multiple nearest neighbors e.g., $r = 100$ for information retrieval in multimedia databases.
- *inverted list size*, the number of vectors in the scanned inverted list size.

We compare the performance of PQ Fast Scan, and the libpq implementation of ADC Scan. We compare their respective *scan speed*, expressed in millions of codes scanned per second (M codes/s). Scan speed is obtained by dividing response times by the size of inverted lists. We do not evaluate PQ Fast Scan accuracy, recall or precision because PQ Fast Scan returns the exact same results as ADC Scan and PQ accuracy has already been extensively studied [39]. In addition to theoretical guarantees, we checked that PQ Fast Scan returned the same results as the libpq implementation of ADC Scan for every experiment. Lastly, we run PQ Fast Scan across a variety of different platforms (Table 4.4) and demonstrate it consistently outperforms ADC Scan by a factor of 4-6. All experiments were run on a *single processor core*. Unless otherwise noted, experiments were run on laptop (A) (Table 4.4).

4.3.2 Distribution of Response Times

We study the distribution of PQ Fast Scan response times. Contrary to ADC Scan, PQ Fast Scan response time varies with the query vector. Indeed, PQ Fast Scan performance depends on the amount of asymmetric distance computations that can be pruned, which depends on the query vector. Figure 4.10 shows the distribution of response times of 2595 nearest neighbor queries executed on the inverted list 0. As expected, ADC Scan response time is almost constant across different query vectors. PQ Fast Scan response time is more dispersed, but it responds to the bulk of queries 4-6 times faster than ADC Scan, as shown in Table 4.3. In the remainder of this section, when studying the impact of different parameters on PQ Fast Scan performance, we

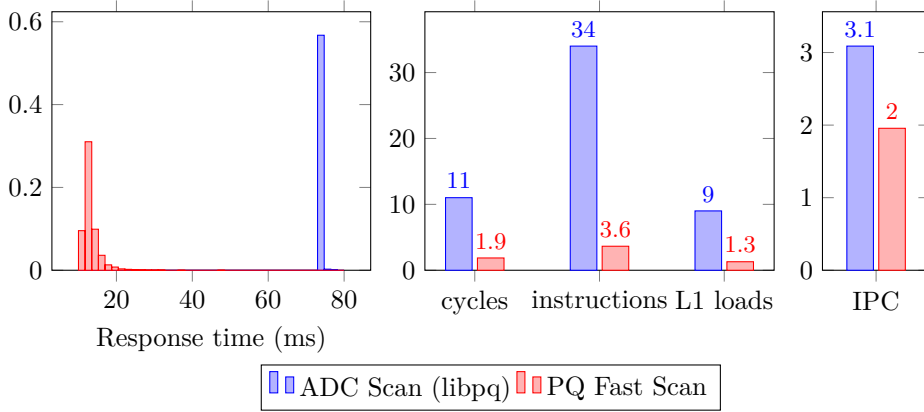


Figure 4.10: Distribution of response times, performance counters and IPC

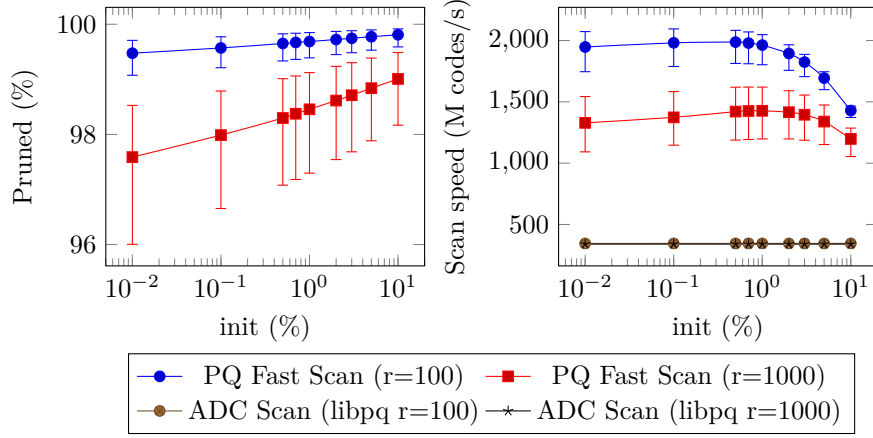
plot *median* response times or *median* scan speeds. We use the *1st quartile* (25th percentile) and *3rd quartile* (75th percentile) to draw error bars. Because it directly impacts performance, we also plot the percentage of pruned distance computations.

4.3.3 Performance Counters

We use performance counters to measure the usage of CPU resources of PQ Fast Scan and ADC Scan when scanning the inverted list 0 (Figure 4.10). Thanks to the use of register-resident small tables, PQ Fast Scan only performs 1.3 L1 loads per scanned vector, where the libpq implementation of ADC Scan requires 9 L1 loads. PQ Fast Scan requires 89% less instructions than ADC Scan thanks to the use of SIMD instructions instead of scalar ones (respectively 3.7 and 34 instructions per scanned vector). PQ Fast Scan uses 83% less cycles than ADC Scan (respectively 1.9 and 11 cycles per vector). The decrease in cycles is slightly less than the decrease in instructions because PQ Fast Scan has a lower IPC than ADC Scan. This is because SIMD instructions can be less easily pipelined than scalar instructions.

4.3.4 Impact of *init* and *r* Parameters

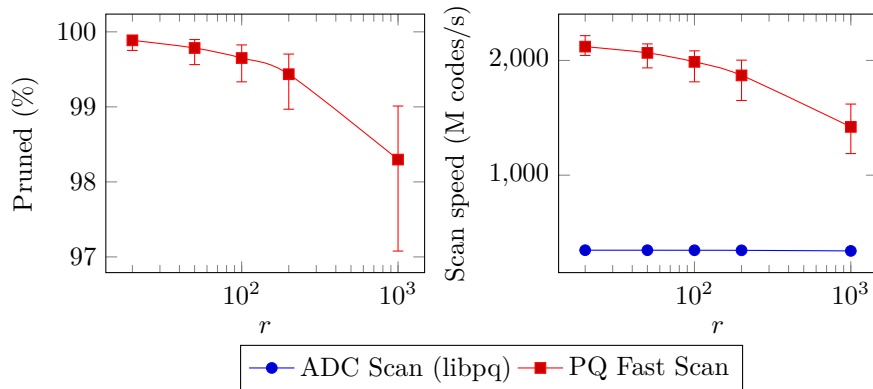
Both *init* and *r* impact the amount of pruned distance computations, and thus PQ Fast Scan performance. For information retrieval in multimedia databases, *r* is often set between 100 and 1000. Therefore, we start by studying the impact of *init* for *r* = 100 and *r* = 1000. The *init* parameter impacts the tightness of the *qmax* bound used for quantization. A higher *init* value means more vectors are scanned using the conventional ADC Scan procedure to find a temporary nearest neighbor (Section 4.2.4). This makes the *qmax* bound tighter and decreases the distance quantization error. Figure 4.11 shows that the pruning power increases with *init*; however this increase is moderate. For *r* = 1000, the pruning power is lower than for *r* = 100 and more sensitive to *init*. PQ Fast Scan can prune a distance computation if the lower bound of the currently scanned vector is higher than the distance between the query vector and the current *r*th nearest neighbor. A higher *r* value implies a higher distance between the query vector and the *r*th nearest neighbor. Therefore, less distance computations can be pruned.

Figure 4.11: Impact of *init* Parameter (all inverted lists)

The scan speed increases slightly with *init* as more distance computations get pruned, up to a threshold where it starts to collapse. After this threshold, the increase in pruned distance computations provided by the tighter *qmax* bound is outweighed by the increased time spent scanning the first *init*% vectors using the slow ADC Scan procedure. Overall, PQ Fast Scan is not very sensitive to *init*, and a decent *qmax* bound is found quickly. Any *init* value between 0.1% and 1% is suitable. We set *init* = 0.5% for the remainder of experiments. Lastly, we evaluate PQ Fast Scan performance for more *r* values. Figure 4.12 confirms that PQ Fast Scan performance decreases with *r*.

4.3.5 Impact of Distance Quantization

PQ Fast Scan uses three techniques to build small tables: (1) vector grouping, (2) minimum tables and (3) quantization of distances. Among these three techniques, minimum tables and quantization of distances impact the tightness of lower bounds,

Figure 4.12: Impact of *r* Parameter (all inverted lists, *init*=0.5%)

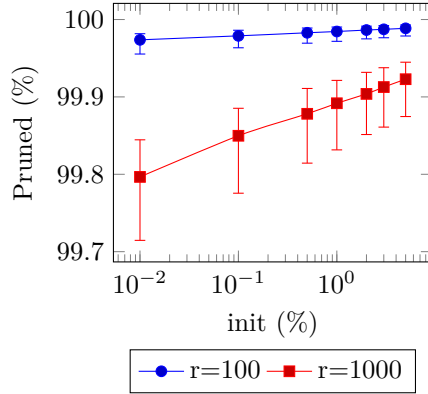


Figure 4.13: Pruning power using quantization only (all inverted lists)

and therefore pruning power. To assess the respective impact on pruning power of these two techniques, we implement a quantization-only version of PQ Fast Scan which relies *only* on quantization of distances (Figure 4.13). This version uses tables of 256 8-bit integers, while the full version of PQ Fast Scan uses tables of 16 8-bit integers. Therefore, the quantization-only version cannot use SIMD and offers no speedup. Hence, Figure 4.13 shows only the pruning power and does not show the scan speed. The quantization-only version of PQ Fast Scan achieves 99.9% to 99.97% pruning power. This is higher than the pruning power of the full version of PQ Fast Scan (i.e., using the three techniques) which is 98% to 99.7% (Figure 4.11). This demonstrates that our quantization scheme is highly efficient and that *most of the loss of pruning power comes from minimum tables*.

4.3.6 Impact of the Size of Inverted Lists

The size of inverted lists impacts scan speed without impacting pruning power (Figure 4.14). Inverted lists 0, 7, 2, 4, 5 and 3 have sizes comprised between 10 million vectors and 25 million vectors, and PQ Fast Scan speed is almost constant across all

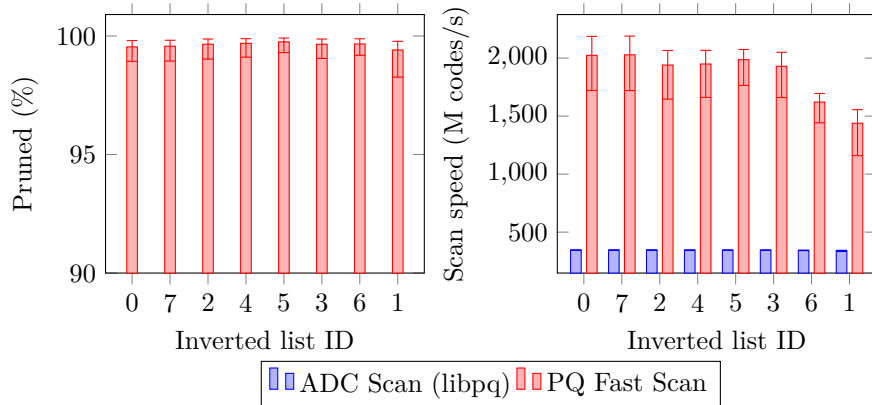


Figure 4.14: Impact of inverted list size (init=0.5%, r=100)

Table 4.4: Systems

	laptop (A)	workstation (B)	server (C)	server (D)
CPU Model	Core i7-4810MQ	Xeon E5-2609v2	Xeon E5-2640	Xeon X5570
CPU Arch.	Haswell	Ivy Bridge	Sandy Bridge	Nehalem
CPU Freq.	2.8-3.8 Ghz	2.5-2.5 Ghz	2.5-3.0 Ghz	2.9-3.33 Ghz
Mem.	8 GB (2 × 4 GB)	16 GB (4 × 4 GB)	64 GB (4 × 16 GB)	24 GB (6 × 4 GB)
Mem. Type	DDR3	DDR3	DDR3	DDR3
Mem. Freq.	1600 Mhz	1333 Mhz	1333 Mhz	1066 Mhz

theses inverted lists. Smaller inverted lists, e.g., inverted lists 6 and 1, exhibit lower scan speeds. PQ Fast Scan groups vectors on 4 components for inverted lists exceeding 3 million vectors, (Section 4.2.2). As inverted lists sizes approaches this threshold, the scan speed decreases. This is because the inverted list size s approaches the minimum inverted list size of $s_{min} = 50$ codes. Small tables are loaded too frequently in comparison with the number of codes scanned, which impacts performance.

4.3.7 Large Scale Experiment

We test PQ Fast Scan on the full database of 1 billion vectors (ANN_SIFT1B). For this database, we build an index which divides the database into 128 inverted lists. Inverted lists therefore have an average size of about 8 million vectors. We run 10000 NN queries. The most appropriate inverted list for each query is selected using the index, and scanned to find nearest neighbors. We scan inverted lists using both ADC Scan and PQ Fast Scan, and we compare mean response times to queries (Figure 4.15, SIFT1B). In addition to its lower response time, PQ Fast Scan also allows decreasing the amount of memory consumed by the database thanks to vector grouping (Section 4.2.2). Unlike previous experiments, this experiment was run on workstation (B) instead of laptop (A) (Table 4.4). The parameters $init = 1\%$, $r = 100$ were chosen.

4.3.8 Impact of CPU Architecture

To conclude our evaluation section, we compare PQ Fast Scan and ADC Scan over a wide range of using processors released between 2009 and 2014 (Table 4.4). On all these systems, PQ Fast Scan median speed exceeds ADC Scan median speed by a factor of 4-6, thus validating our performance analysis and design hypotheses (Figure 4.15, Scan speed). In addition, PQ Fast Scan performance is not sensitive to processor architecture. PQ Fast Scan loads 6 bytes from memory for each lower bound computation. Thus, a scan speed of 2000 M codes/s correspond to a bandwidth use of 16 GB/s. The memory bandwidth of Intel server processors ranges from 40 GB/s to 70 GB/s. When answering 8 queries concurrently on an 8-core server processor, PQ Fast Scan is bound by the memory bandwidth, thus demonstrating its highly efficient use of CPU resources.

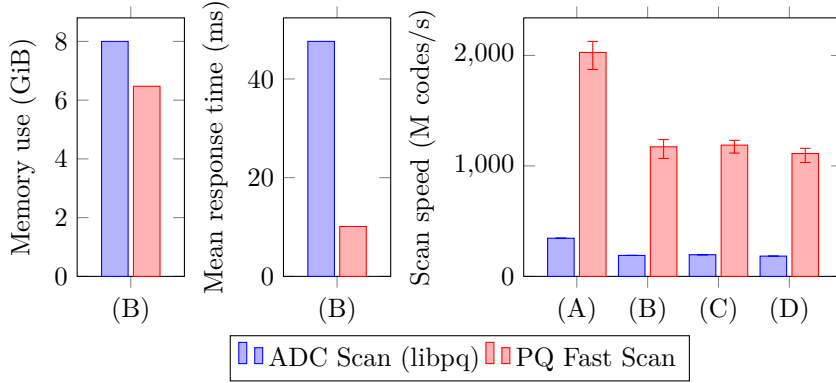


Figure 4.15: Experiments on other CPU architectures (see Table 4.4)

4.4 Discussion

4.4.1 Compatibility with Inverted Indexes

The main limitation of PQ Fast Scan is that it is not compatible with the most advantageous configurations of inverted indexes. Inverted indexes are commonly combined with product quantization to decrease response time and increase accuracy. It has been shown that using an inverted index that partitions the database into a large number of small inverted lists (fine partitioning) offers better results than an inverted index that partitions the database into a small number of inverted lists (Section 2.3.2). For large datasets (e.g., 1 billion vectors), it is common to use inverted indexes that partitions the database into $K = 8192 - 65536$ inverted lists. Inverted lists therefore have a size of $s = 15000 - 200000$ codes. On the other hand, PQ Fast Scan requires inverted lists of at least 3 million vectors (Section 4.2.2), and works better with larger inverted lists (Section 4.3.6). In practice, this makes it difficult to combine the speedup provided by fine inverted indexes and the speedup provided by PQ Fast Scan. In our experiments on a 1 billion vector dataset (Section 4.3.7), we used an inverted index with only $K = 128$ inverted lists. The mean size of inverted lists is therefore of 8 million codes, ideal for PQ Fast Scan. Combining a finer inverted index (e.g., with $K = 8192 - 65536$ inverted lists) with PQ Fast Scan would have offered an even better speedup, but this is currently not possible.

Two factors mitigate this issue. First, in very large databases i.e., exceeding hundreds of billions of vectors, inverted lists may exceed 3 million codes, even if a fine inverted index is used. Second, the upcoming AVX-512 SIMD instruction set will allow PQ Fast Scan to work with smaller inverted lists. The constraints on inverted lists sizes imposed by PQ Fast Scan come from code grouping (Section 4.2.2). We showed that the minimum inverted list size is given by $n_{min}(g) = s_{min} \cdot P^g$, where s_{min} is the minimum number of codes per group (in our case $s_{min} = 50$), and P is the number of portions in lookup tables. The current version of PQ Fast Scan relies on 128-bit in-register shuffles, which allows for small tables of 16 elements (16×8 bits). Lookup tables therefore have to be divided into $P = 256/16 = 16$ portions, which leads to a minimum inverted list size $n_{min}(4) = 50 \cdot 16^4$ of 3 million codes. The AVX-512 version of PQ Fast Scan would use 512-bit in-register shuffles, and

allow for small tables of 64 elements (64×8 bits). Lookup tables would therefore be divided in $P = 256/64 = 4$ portions, which would lead to a minimum inverted list size $n_{min}(4) = 50 \cdot 4^4$ of 12000 codes. Moreover, using minimum tables of 64 elements instead of 16 would allow grouping on 3 components instead of 4 components without impacting pruning power too strongly. The minimum inverted list size $n_{min}(3)$ of the AVX-512 version of PQ Fast Scan may be as low as 3000 codes. Therefore, the AVX-512 version of PQ Fast Scan would be compatible with almost all configurations of inverted indexes.

4.4.2 Applicability to Product Quantization Derivatives

As currently designed, PQ Fast Scan is only compatible with $m \times b = 8 \times 8$ product quantizers (64-bit codes). Minor adjustments may however make it compatible with other $m \times 8$ product quantizers, e.g., 16×8 product quantizers (128-bit codes). In all cases, it is only possible to use code grouping for $g = 4$ components (Section 4.2.2). For 16×8 product quantizers, this means that we would need to use the minimum tables technique on $m - g = 16 - 4 = 12$ tables. Experiments would be required to determine if PQ Fast Scan still has a high enough pruning power to offer a significant speedup in this scenario. Making PQ Fast Scan compatible with $m \times 8$ product quantizers with $m < 8$, e.g., 4×8 product quantizers (32-bit codes) would be easy. These cases are indeed less challenging than 8×8 product quantizers. Code grouping can still be used to build $g = 4$ (or less) small tables. This means that minimum tables have to be computed for only $m - g < 4$ tables and therefore guarantees that PQ Fast Scan will have a high enough pruning power. However, this type of codes is still rarely used. Making PQ Fast Scan compatible with b parameters other than $b = 8$ would not make much sense, as it would require changing the whole algorithm. PQ Fast Scan has been designed to work with lookup tables of 256 floating-point values, resulting from the use of 8-bit sub-quantizers.

In Section 2.3.4, we presented derivatives of product quantization, that have been recently introduced: Optimized Product Quantization (OPQ), Additive Quantization (AQ), Tree quantization (TQ) or composite quantization (CQ). Like product quantization, these approaches represent high-dimensional vectors by a combination of m centroids taken from m codebooks of 2^b centroids each. All these approaches offer a lower quantization error than product quantization. These approaches use (1) a different process to learn quantizer codebooks, and (2) a different ADC procedure from the one of product quantization. We discuss the applicability of PQ Fast Scan to these derivatives of product quantization.

The codebook learning process of OPQ, AQ, TQ and CQ has only been tested with 8-bit sub-quantizers ($m \times 8$ codes). PQ Fast Scan also relies on the use of 8-bit sub-quantizers, thus it requires no changes to the codebook learning process of these product quantization derivatives. Besides, OPQ and CQ use the same ADC procedure as product quantization. Therefore, the adaptation of PQ Fast Scan to OPQ or CQ is straightforward. The case of AQ and TQ is more challenging. The ADC process of AQ requires about $m + m^2/2$ table lookups and additions, while the ADC process of TQ requires about $2 \cdot m$ tables lookups and additions. In all cases, PQ Fast Scan groups codes on $g = 4$ components and computes minimum tables for the remaining tables. For AQ, this means that $g + g^2/2$ small tables would be built using code

grouping, and $(m - g) + (m - g)^2/2$ small tables would be built computing minimum tables. Likewise, for TQ, $2g$ tables would be built using code grouping and $2(m - g)$ small tables would be built computing minimum tables. In both cases, experiments are required to determine if this combination of code grouping and minimum tables provides a high enough pruning power. If this is the case, PQ Fast Scan would be able to provide a higher speedup for AQ and TQ than for PQ. As these derivatives require more additions ($m + m^2/2$ and $2m$) than product quantization, they would benefit more of the speedup offered by SIMD additions.

5 Quick ADC

Contents

5.1	Motivation	57
5.2	Presentation	58
5.2.1	Overview	58
5.2.2	SIMD Distance Computations	59
5.3	Evaluation	61
5.3.1	Experimental Setup	61
5.3.2	Exhaustive Search in SIFT1M	62
5.3.3	Non-exhaustive Search in SIFT1M	64
5.3.4	Non-exhaustive Search in GIST1M	65
5.3.5	Non-exhaustive Search in Deep1M	66
5.3.6	Non-exhaustive Search in SIFT1B	67
5.4	Discussion	68
5.4.1	Compatibility with Inverted Indexes	68
5.4.2	Applicability to Product Quantization Derivatives	68

5.1 Motivation

Even if product quantization is among the fastest nearest neighbor search approaches, we have shown that the procedure used to scan inverted lists of short codes has a low computational efficiency. We have proposed PQ Fast Scan, a highly efficient scan procedure that replaces slow cache accesses by fast SIMD in-register shuffles. Replacing cache accesses by SIMD in-register shuffles requires building small tables that fit SIMD registers. PQ Fast Scan achieves this result by (1) grouping codes, (2) computing minimum tables and (3) quantizing floating-point distances to 8-bit integers. We have shown that grouping codes imposes a minimum size on inverted lists. This makes it impossible to combine PQ Fast Scan with inverted indexes, another widespread search acceleration technique.

In this chapter, we introduce Quick ADC, a fast scan procedure that can be combined with inverted indexes. Like PQ Fast Scan, Quick ADC offers a 4-6 speedup over the conventional scan procedure. Quick ADC also builds on the idea of replacing costly cache accesses by cheap SIMD in-register shuffles. Unlike PQ Fast Scan, Quick ADC achieves this results by (1) using 4-bit sub-quantizers and (2) quantizing floating-point distances to 8-bit integers. Using 4-bit sub-quantizers makes it possible

to build small tables that fit SIMD registers, without imposing size constraints on inverted lists. On the flip side, using 4-bit sub-quantizers ($m \times 4$ codes) instead of the commonly used 8-bit sub-quantizers ($m \times 8$ codes) causes a slight decrease in recall. We however show that this decrease is generally small to negligible and unlikely to have a practical impact. More specifically, this chapter addresses the following points:

- We briefly present the key points of the design of Quick ADC. We show that using 4-bit sub-quantizers allows fast distance computations relying on SIMD in-register shuffles.
- We evaluate Quick ADC in a wide range of scenarios. We evaluate the recall of Quick ADC not only on SIFT descriptors (128 dimensions), but also on more challenging types of vectors such as deep neural codes (256 dimensions) or GIST descriptors (960 dimensions). We show that Quick ADC only causes a slight decrease in recall, especially when combined with inverted indexes and optimized product quantization. In all cases, Quick ADC yields 4-6 times better performance than the conventional scan procedure.
- We discuss the differences between PQ Fast Scan and Quick ADC. We review the applicability of Quick ADC to derivatives of product quantization.

5.2 Presentation

5.2.1 Overview

4-bit sub-quantizers Quick ADC relies on SIMD in-register shuffles for distance computations. We have demonstrated that SIMD in-register shuffles offer much better performance than cache accesses or SIMD gather operations. Thus, SIMD in-register shuffles perform 16 table lookups in 1 cycle while cache accesses only perform 1 table lookup in 1 cycle. However, SIMD in-register shuffles require lookup tables to be stored in SIMD registers, and limit their size to 16 elements of 8 bit each (Section 4.2.1). Quick ADC achieves this result by (1) imposing the use of 4-bit sub-quantizers and (2) quantizing floating point distances to 32-bit integers. Using 4-bit sub-quantizers inherently lead to lookup tables of 16 elements, without any additional modifications. An $m \times b$ product quantizer generates m lookup tables of 2^b elements each. An $m \times 8$ product quantizer (product quantizer with 8-bit sub-quantizers) generates lookup tables of $2^8 = 256$ floating-point values. As PQ Fast Scan uses 8-bit sub-quantizers, it has to perform additional modifications to shrink lookup tables of 256 values to 16 values (grouping codes, and computing tables of minimums). By contrast, an $m \times 4$ product quantizer (product quantizer with 4-bit sub-quantizers) generates lookup tables of $2^4 = 16$ floating-point values. Therefore, Quick ADC does not need to perform additional modifications to obtain lookup tables of 16 elements.

Floating-point quantization Although product quantizers that use 4-bit sub-quantizers inherently generate lookup tables of 16 elements, these elements are 32-bit floating-point distances. SIMD in-register shuffles require lookup tables of 16 elements of 8-bit each (16 $\times$ 8 bit). Therefore, like in PQ Fast Scan we quantize floating-point distances to 8-bit integers (Section 4.2.4). As there is no SIMD instruction

Algorithm 2 ANN Search with Quick ADC

```

1: function LOOKUP_ADD(comps,  $D^j$ , acc) ▷ Fig. 5.2
2:   r128 masked  $\leftarrow$  simd_and(comps, 0xf)
3:   r128 partial  $\leftarrow$  simd_shuffle(comps,  $D^j$ )
4:   return simd_add_saturated(acc, partial)
5: function QUICK_ADC_BLOCK(blk,  $\{D^j\}_{j=0}^{m-1}$ )
6:   r128 acc  $\leftarrow$  {0}
7:   for  $j \leftarrow 0$  to  $m/2 - 1$  do
8:     r128 comps  $\leftarrow$  simd_load(blk +  $j \cdot 16$ )
9:     acc  $\leftarrow$  LOOKUP_ADD(comps,  $D^{2j}$ , acc)
10:    comps  $\leftarrow$  simd_right_shift(comps, 4) ▷ Fig. 5.3
11:    acc  $\leftarrow$  LOOKUP_ADD(comps,  $D^{2j+1}$ , acc)
12:   return acc
13: function QUICK_ADC_SCAN(tlist,  $\{D^j\}_{j=0}^{m-1}$ , R)
14:   neighbors  $\leftarrow$  binheap(R)
15:   for blk in tlist do
16:     r128 acc  $\leftarrow$  QUICK_ADC_BLOCK(blk,  $\{D^j\}_{j=0}^{m-1}$ )
17:     EXTRACT_MATCHES(acc, neighbors)
18:   return neighbors

```

to compare 8-bit unsigned integers, we quantize distances to 8-bit integers, only using their positive range. Like in PQ Fast Scan, we quantize dsitances between a $qmin$ and $qmax$ bound intro $n = 127$ bins (0-126) uniformly. Values above $qmax$ are quantized to 127. We compute $qmin$ and $qmax$ as in PQ Fast Scan. Thus, $qmin$ is the minimum value across all lookup tables. To determine $qmax$ we scan *init* vectors to find a temporary nearest neighbor. We use the distance of the query vector to this temporary nearest neighbor as the $qmax$ bound.

5.2.2 SIMD Distance Computations

As the Quick ADC algorithm is simpler than PQ Fast Scan, we are able to detail how we utilize SIMD instructions to implement it. In the evaluation section, we use a 256-bit version of Quick ADC implemented with the new AVX 256-bit SIMD instruction set. Yet, for the sake of simplicity, we describe a 128-bit version of Quick ADC, and show how to generalize it to 256-bit SIMD at the end of the section. The 128-bit version of Quick ADC has also the advantage of being compatible with older Intel CPUs and ARM CPUs. The two versions are similar, as 256-bit SIMD in-register shuffles do not perform full-length shuffles on 256-bit (32×8 bit) lookup tables. Instead, 256-bit SIMD in-registers perform shuffles in two independent 128-bit tables, stored in two independent 128-bit lanes. The speedup provided by the 256-bit

a	b	c		p
$a_1 a_0$	$b_1 b_0$	$c_1 c_0$	...	$p_1 p_0$
...	...	...	...	...
$a_{m-1} a_{m-2}$	$b_{m-1} b_{m-2}$	$c_{m-1} c_{m-2}$	...	$p_{m-1} p_{m-2}$

Figure 5.1: Transposed block

5. Quick ADC

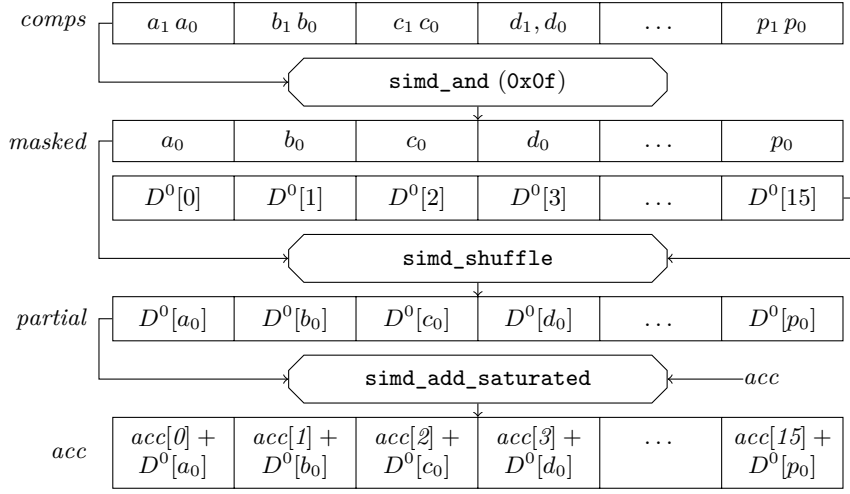


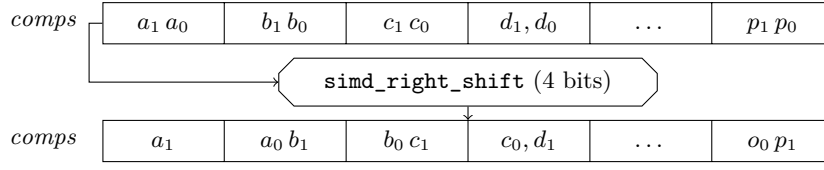
Figure 5.2: SIMD Lookup-add ($j = 0$)

version in comparison with the 128-bit is small: it is generally comprised between 10% and 15%.

Quick ADC operates on a block-transposed inverted list of codes, like PQ Fast Scan (Section 4.2.5) or the SIMD gather implementation of the conventional ADC Scan algorithm (Section 3.2). This is required as SIMD instructions operate on 16 codes at once. SIMD in-register shuffles are able to perform 16 tables lookups at once, but in the *same* lookup table. Therefore, to perform the first SIMD shuffle, we need to load the first component of 16 codes ($a_0, \dots, p_0$) in an SIMD register. This can be done efficiently only if these 16 components are contiguous in memory, which is why we need to block-transpose the inverted list. To block-transpose the inverted list, we first divide it into blocks of 16 codes ($a, \dots, p$). We then transpose each block independently. In a transposed block (Figure 3.5), we store the first components of 16 codes contiguously ($a_0, \dots, p_0$), followed by the second components of the same 8 code ($a_1, \dots, p_1$) etc., instead of storing all components of the first vector ($a_0, \dots, a_7$), followed by the components of the second vector ($b_0, \dots, b_7$) etc.

The `QUICK_ADC_SCAN` function (Algorithm 2, line 12) scans a block-transposed inverted list *tlst* using m quantized lookup tables $\{D^j\}_{j=0}^{m-1}$, where m is the number of sub-quantizers of the product quantizer. Each lookup table is stored in a distinct SIMD register. In Algorithm 2, SIMD instructions are denoted by the prefix `simd_`. SIMD instructions use 128-bit variables, denoted by `r128`. The function named `QUICK_ADC_SCAN` iterates over blocks *blk* of 16 codes (Algorithm 2, line 14). The `QUICK_ADC_BLOCK` function computes the distance between the query vector and the 16 codes ($a, \dots, p$) of the block *blk*.

Each block comprises $m/2$ rows of 16 bytes (128 bits). Each row stores the j th and $(j + 1)$ th components of 16 codes (Figure 5.1). The `QUICK_ADC_BLOCK` function iterates over each row (Algorithm 2, line 7), and loads it in the *comps* register sequentially (Algorithm 2, line 8). Two lookup-add operations are performed on each row

Figure 5.3: SIMD 4-bit Right Shift ($j = 0$)

(Algorithm 2, line 9 and line 11): one for the $(2j)$ th components, and one for $(2j+1)$ th components of the codes. Figure 5.2 describes the succession of operations performed by the LOOKUP_ADD function for the first row ($j = 0$). As each byte of the first row stores two components, e.g., the first byte of the first row stores a_1 and a_0 (Figure 5.2), we start by masking the lower 4 bits of each byte (and with $0x0f$), to obtain the first components ($a_0, \dots, p_0$) only. The remainder of the function looks up values in the D^0 table and accumulates distances in *acc* variable. Before the LOOKUP_ADD function can be used to process the second components ($a_1, \dots, p_1$), it is necessary that $(a_1, \dots, p_1)$ are in the lowest 4 bits of each byte of the register. We therefore right shift the *comps* register by 4 bits (Figure 5.3) before calling LOOKUP_ADD (Algorithm 2, line 10). The EXTRACT_MATCHES function (Algorithm 2, line 16), the implementation of which is not shown, extracts distances from the *acc* register and inserts them in the binary heap *neighbors*.

Among 256-bit SIMD instructions (AVX and AVX2 instruction sets) supported on recent CPUs, some, like in-register shuffles, operate concurrently on two independent 128-bit lanes. This prevents use of 256-bit lookup tables (32 8-bit integers) but allows an easy generalization of the 128-bit version of Quick ADC. While the 128-bit version of Quick ADC iterates on block rows one by one (Algorithm 2, line 7), the 256-bit version processes two rows at once: one row in each 128-bit lane. The number of iterations is thus reduced from $m/2$ to $m/4$. Lastly, instead of storing each D^j table in a distinct 128-bit register, the tables D^j and D^{2j} , $j \in \{0, \dots, m/2 - 1\}$, are stored in each of the two lanes of a 256-bit register.

5.3 Evaluation

5.3.1 Experimental Setup

We implemented Quick ADC in C++, using intrinsics to access SIMD instructions [33, 32]. While PQ Fast Scan is implemented using 128-bit SIMD (Section 4.3.1), we implemented Quick using 256-bit SIMD, which allows and additional performance boost of 10%-15%. Our implementation therefore uses the AVX and AVX2 instruction sets. We used the g++ compiler version 5.3, with the options `-O3 -ffast-math -m64 -march=native`. Exhaustive search and non-exhaustive search (inverted indexes, IVF) were implemented as described in [39]. We use the yael library and the ATLAS library version 3.10.2. We compiled an optimized version of ATLAS on our system. To learn product quantizers and optimized product quantizers, we used the implementation¹ of the authors of [10, 8].

1. <https://github.com/arbabenko/Quantizations>

Table 5.1: Systems

	CPU	RAM
workstation	Xeon E5-1650v3	16GB DDR4 2133Mhz
server	Xeon E5-2630v3	128GB DDR4 1866Mhz

Table 5.2: Datasets

	Base set	Learning set	Query set	Dim.
SIFT1M	1M	100K	10K	128
SIFT1B	1000M	100M (2M)	10K	128
GIST1M	1M	500K	1K	960
Deep1M	1M	300K	1K	256

Unless otherwise noted, experiments were performed on our workstation (Table 5.1). To get accurate timings, we processed queries sequentially on a single core. We evaluate our approach on two publicly available² datasets of SIFT descriptors, one dataset of GIST descriptors, and one dataset of deep features³ (Table 5.2). The Deep1M dataset consists of deep neural codes that were L_2 -normalized and PCA-compressed to 256 dimensions [10]. For SIFT1B, the learning set is needlessly large to train product quantizers, so we used the first 2 million vectors. We used the first 1000 queries from the query sets of SIFT1M and SIFT1B.

5.3.2 Exhaustive Search in SIFT1M

Using 16×4 Quick ADC (QADC) instead of 8×8 ADC causes a decrease in recall which is due to two factors: (1) use of 16×4 quantizers instead of 8×8 quantizers and (2) use of quantized lookup tables (Section 5.2.1). We evaluate the global decrease

2. <http://corpus-texmex.irisa.fr/>

3. <http://sites.skoltech.ru/compvision/projects/aqtq/>

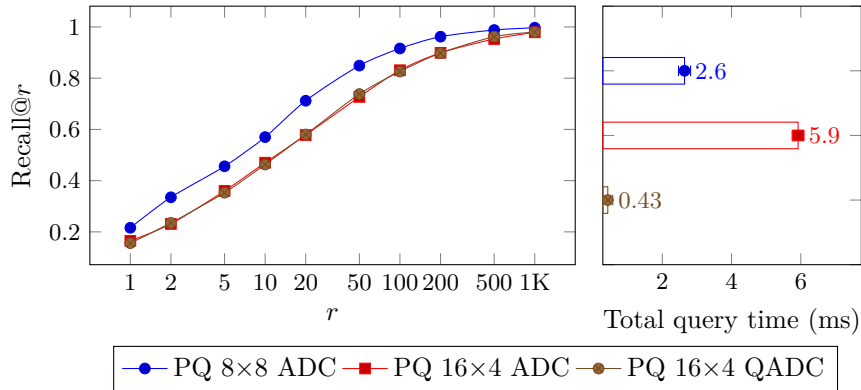


Figure 5.4: ADC and QADC response time and recall (PQ, SIFT1M, Exhaustive search)

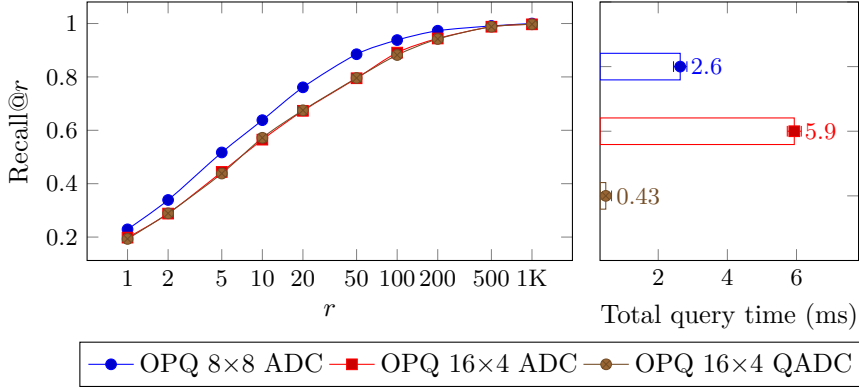


Figure 5.5: ADC and QADC response time and recall (OPQ, SIFT1M, Exhaustive search)

in recall caused by the use of 16×4 QADC instead of 8×8 ADC, but also the relative impact of factors (1) and (2). To do so, we use the SIFT1M dataset and follow an exhaustive search strategy. Because we do not use an inverted index, we encode the original vectors into short codes, and not residuals. This maximizes quantization error and thus represents a worst-case scenario for QADC. We scan $init = 200$ vectors to set the $qmax$ bound for quantization of lookup tables (Section 5.2.1).

Using 16×4 ADC causes a small decrease in recall (Figure 5.4). However, 16×4 QADC, which uses quantized lookup tables, does not further decrease recall in comparison with 16×4 ADC. Interestingly, the recall of 16×4 QADC is even higher than the recall of 16×4 ADC for some points e.g., the Recall@50 of 16×4 QADC is 0.738 against 0.727 for 16×4 ADC. This is because ADC already approximates real distances, as ADC computes distances between the query vector and *quantized* database vectors. QADC adds another layer of approximation, which can either compensate or worsen the error introduced by PQ and ADC. OPQ yields better results than PQ in all cases (Figure 5.5), which is consistent with [46, 24]. Moreover, the difference in recall between 8×8 ADC and 16×4 QADC is lower in the case of OPQ than in the case of PQ. OPQ optimizes the decomposition of the input vector space into m sub-spaces, which are used by the optimized product quantizer (Section 2.3.4). For $m = 16$, OPQ has more degrees of freedom than for $m = 8$ and is therefore able to bring a greater level of optimization.

For an exhaustive search in 1 million vectors, 16×4 QADC is ~ 14 times faster than 16×4 ADC and 6 times faster than 8×8 ADC (Figure 5.4, Figure 5.5) (85% decrease in response time). Response times for PQ and OPQ are similar (Figure 5.5). In practice, 8×8 ADC is much more common than 16×4 ADC [8, 10, 9, 46, 61], thus we only compare 16×4 QADC with 8×8 ADC in the remainder of this section. Overall, QADC therefore proposes trading a small decrease in recall, and almost negligible in the case of OPQ, for a large improvement in response time.

Non-exhaustive search offers both lower response time and higher recall than exhaustive search (Section 2.3.2) and is often preferred in practice. Therefore, in the

Table 5.3: Non-exhaustive search, SIFT1M, 64 bit

PQ	ADC *	R@100	Index	Tables	Scan	Total
SIFT1M, IVF, K=1024, ma=48						
PQ	ADC	0.951	0.023	0.35	0.2	0.57
	QADC	0.917	0.024	0.1	0.067	0.19
		-3.6%		-70%	-66%	-66%
OPQ	ADC	0.977	0.023	0.39	0.19	0.6
	QADC	0.956	0.023	0.18	0.068	0.28
		-2.1%		-53%	-65%	-54%
SIFT1M, IVF, K=256, ma=24						
PQ	ADC	0.949	0.008	0.18	0.3	0.48
	QADC	0.907	0.008	0.055	0.072	0.14
		-4.4%		-69%	-76%	-72%
OPQ	ADC	0.963	0.008	0.21	0.29	0.52
	QADC	0.949	0.008	0.089	0.073	0.17
		-1.5%		-59%	-75%	-67%

* ADC: 8×8 ADC, QADC: 16×4 QADC

remainder of this section, we evaluate QADC in the context of non-exhaustive search, for a wide range of scenarios: SIFT and GIST descriptors, PQ and OPQ, 64 and 128 bit codes, and varying index parameters. We show that in most cases, when combined with OPQ and inverted indexes, QADC offers a speedup of 3.3-4 (70-75% decrease in response time) for a small or negligible loss in recall.

5.3.3 Non-exhaustive Search in SIFT1M

Some approaches like PQ Fast Scan [2] or the inverted multi-index [9] have only been evaluated on large datasets. To show the versatility of QADC, we evaluate it on both large and small datasets, such as SIFT1M. Table 5.3 compares ADC and QADC in terms of Recall@100 (R@100), total ANN search time (Total), and the time spent in each of the search steps (Index, Tables, and Scan) detailed in Section 2.3.3. All times are in milliseconds (ms). OPQ requires a rotation of the input vector before computing lookup tables (Section 2.3.4). We include the time to perform this rotation in the Tables column.

When using inverted indexes, the parameters K , the total number of cells of the inverted index, and ma , the number of cells scanned to answer a query, impact response time and recall (Section 2.3.3). To answer a query $S \cdot ma/K$ vectors are scanned on average, where S is the size of the database, and ma tables are computed (Section 2.3.3). Recall increases with ma/K but so does response time. For a constant ma/K , a higher K provides better recall. We use an inverted index with $K = 1024$ cells, as in [39], and $ma = 48$. In this case, QADC offers a 66% decrease in scan time (Table 5.3), which is less than the 85% decrease for exhaustive search (Figure 5.4c). This is because when using inverted indexes, scanned codes are scattered across ma cells of relatively small size (on average, cells comprise $S/K \approx 976$ vectors). A

Table 5.4: Non-exhaustive search, GIST1M, 128 bit

PQ	ADC *	R@100	Index	Tables	Scan	Total
GIST1M, IVF, K=256, ma=24						
PQ	ADC	0.675	0.038	0.77	0.71	1.5
	QADC	0.515	0.038	0.26	0.15	0.45
		-24%		-67%	-79%	-71%
OPQ	ADC	0.918	0.039	1.7	0.73	2.5
	QADC	0.872	0.038	1.2	0.16	1.4
		-5%		-32%	-78%	-45%

* ADC: 16×8 ADC, QADC: 32×4 QADC

significant amount of time is therefore spent switching cells. Using larger cells solves this issue: for $K = 256$, $ma = 24$, QADC offers a 75% decrease in scan time. The scan time is not significantly increased (0.072ms versus 0.067ms) for $K = 256$, $ma = 24$ despite twice more vectors are scanned compared to $K = 1024$, $ma = 48$. However, for $ma = 24$, two times less lookup tables are computed, which significantly decreases response time. Therefore, in the context of QADC, the configuration $K = 256$, $ma = 24$ is more interesting.

Thanks to the use of 4-bit quantizers, which results in smaller and faster to compute lookup tables, QADC offers a 50-70% decrease in tables computation time. This decrease is lower with OPQ due to the time spent rotating the query vector. The loss in recall is significantly lower with OPQ (-1.5% to 2.1%) than with PQ (-3.6% to 4.4%). In most cases, QADC offers a close to 70% decrease in total response time.

5.3.4 Non-exhaustive Search in GIST1M

Due to their higher dimensionality GIST descriptors (960 dimensions) suffer a higher quantization error than SIFT descriptors (128 dimensions). To mitigate this issue, 128-bit codes can be used instead of 64-bit codes [46, 9]. Thus, here, we compare 16×8 ADC and 32×4 QADC. As for SIFT1M, we keep the parameters $K = 256$, $ma = 24$. Like for 64-bit codes, QADC provides a 75-80% decrease in scan time with 128-bit codes (Table 5.4). However, the decrease in recall when using QADC with PQ is higher for GIST descriptors (-24%) than for SIFT descriptors (-3.5% to 4.5%). However, using OPQ settles this issue and reduces the recall loss to 5%. OPQ should therefore be preferred for GIST descriptors. The decrease in response time is however less important for OPQ (-45%). This is because a large part of response time is spent computing tables in the case of OPQ. OPQ requires rotating the query vector before computing tables, the computational cost of which depends on the dimensionality of the query vector. For 960-dimensional GIST descriptors this cost is much higher than for 128-dimensional SIFT descriptors. Even in the less favourable case of GIST descriptors, Quick ADC still achieves a close to 50% decrease in response time.

Table 5.5: Non-exhaustive search, Deep1M, 64 bit

PQ	ADC *	R@100	Index	Tables	Scan	Total
Deep1M, IVF, K=256, ma=24						
PQ	ADC	0.772	0.015	0.24	0.33	0.58
	QADC	0.669	0.013	0.082	0.076	0.17
		-13%		-66%	-77%	-71%
OPQ	ADC	0.922	0.013	0.34	0.32	0.67
	QADC	0.902	0.013	0.16	0.08	0.25
		-2.2%		-53%	-75%	-62%

* ADC: 8×8 ADC, QADC: 16×4 QADC

Table 5.6: Non-exhaustive search, Deep1M, 128 bit

PQ	ADC *	R@100	Index	Tables	Scan	Total
Deep1M, IVF, K=256, ma=24						
PQ	ADC	0.922	0.015	0.33	0.58	0.93
	QADC	0.859	0.013	0.13	0.14	0.28
		-6.8%		-62%	-77%	-70%
OPQ	ADC	0.988	0.013	0.39	0.58	0.99
	QADC	0.984	0.013	0.24	0.14	0.39
		-0.4%		-39%	-76%	-60%

* ADC: 16×8 ADC, QADC: 32×4 QADC

5.3.5 Non-exhaustive Search in Deep1M

For the vectors of Deep1M, we evaluate QADC with 64-bit codes (Table 5.5) and 128-bit codes (Table 5.6). We keep the parameters $K = 256$ and $ma = 24$ used for SIFT1M and GIST1M. With 64-bit codes, QADC suffers higher loss of recall (13%) with deep features than with SIFT descriptors (3.6%). The increase in quantization error caused by the use of 4-bit quantizers (QADC) instead of 8-bit quantizers (ADC) is stronger for the PCA-compressed deep features than for SIFT descriptors. This is due to the higher dimensionality of the PCA-compressed deep features (256 dimensions, versus 128 dimensions for SIFT descriptors). As for GIST descriptors, using OPQ instead of PQ mitigates this issue. With OPQ, QADC incurs a 2.2% loss in recall, compared to a 13% loss with OPQ (Table 5.5). With OPQ, the decrease in response time offered by QADC is a slightly lower for the PCA-compressed deep features than for SIFT descriptors (-62% versus -67%). This is because the rotation matrix for PCA-compressed deep features is larger than the rotation matrix for SIFT descriptor. Thus, the time spent to compute lookup tables is increased. With 64-bit codes and OPQ, QADC offers a 60% decrease in response time for a 2.2% decrease in recall (Table 5.5). With 128-bit codes, the loss of recall caused by the use of QADC is lower than for than 64-bit codes, even for PQ (-6.8% versus -13%). For 128-bit codes and OPQ, QADC also offers a 60% decrease in response time. The loss of recall becomes negligible at 0.4%.

Table 5.7: Non-exhaustive search, SIFT1B, 64 bit

PQ	ADC *	R@100	Index	Tables	Scan	Total
SIFT1B, IVF, K=8192, ma=64						
PQ	ADC	0.746	0.09	0.42	23	23
	QADC	0.635	0.088	0.13	5	5.2
		-15%		-69%	-78%	-78%
OPQ	ADC	0.792	0.09	0.53	23	24
	QADC	0.712	0.087	0.22	4.9	5.2
		-10%		-58%	-79%	-78%
SIFT1B, IVF, K=65536, ma=64						
OPQ	ADC	0.806	0.52	0.51	4.2	5.2
	QADC	0.747	0.53	0.22	0.92	1.7
		-7.3%		-57%	-78%	-68%

* ADC: 8×8 ADC, QADC: 16×4 QADC

Table 5.8: Non-exhaustive search, SIFT1B, 128 bits

PQ	ADC *	R@100	Index	Tables	Scan	Total
SIFT1B, IVF, K=65536, ma=64						
OPQ	ADC	0.95	0.73	1.1	10	12
	QADC	0.94	0.72	0.49	2.2	3.4
		-1.1%		-57%	-78%	-72%

* ADC: 16×8 ADC, QADC: 32×4 QADC

5.3.6 Non-exhaustive Search in SIFT1B

For non-exhaustive search in 1 billion SIFT vectors, we test two inverted indexes configurations $K = 8192$, like in [55] and $K = 65536$, like in [9]. For this larger dataset, we scan $init = 1000$ vectors before quantizing lookup tables (Section 5.2.1). In all cases, QADC offers a close to 78-79% decrease in scan time (Table 5.7). For $K = 8192$, QADC also offers a 78% decrease in total query time but the loss in recall is relatively high (-15% for PQ, and -10% for OPQ). As in all other experiments, OPQ offers a much higher recall for a small to negligible increase in response time. Thus, from now on, we only report figures for OPQ only. The configuration $K = 65536$ is more interesting than $K = 8192$ as it allows both a lower response time and a higher recall, both for ADC and QADC. For $K = 65536$ and OPQ, QADC achieves a recall 0.747 in 1.7ms (68% decrease in response time). In comparison, in [9], the state-of-the-art OMulti-D-OADC system achieves the same recall in 2 ms.

With 64-bit codes (Table 5.7), the recall of ANN search in 1 billion vectors is relatively low, even for ADC. To offer better recall, it is possible to use 128-bit codes (Table 5.8). With 128-bit codes, the database uses 20GB of RAM, which exceeds the RAM capacity of our workstation. We therefore ran this experiment on a server (Table 5.1). This configuration allows QADC to achieve a recall of 0.94 in 3.4 ms.

In comparison, in [9], the OMulti-D-OADC system achieves a recall of 0.901 in 5 ms and a recall of 0.969 in 16 ms.

5.4 Discussion

5.4.1 Compatibility with Inverted Indexes

Contrary to PQ Fast Scan, Quick ADC imposes no constraints on the size of inverted lists. Quick ADC is therefore compatible with any type of inverted index. It is not only compatible with fine inverted indexes, but also with multi-indexes (Section 2.3.2). Besides, Quick ADC is compatible with $m \times 4$ product quantizers, for any value of m . In this regard, in Section 5.3, we tested PQ Fast Scan with 16×4 (64-bit codes) product quantizers and 32×4 (128-bit codes) product quantizers. Clearly, Quick ADC is not compatible with b parameters other than $b = 4$, as using 4-bit sub-quantizers is one of the key ideas of Quick ADC.

5.4.2 Applicability to Product Quantization Derivatives

We evaluated Quick ADC with Product Quantization (PQ) and Optimized Product Quantization (OPQ), but we did not evaluate it in the context of other product quantization derivatives presented in Section 2.3.4, namely Additive Quantization (AQ), Tree Quantization (TQ) and Composite Quantization (CQ). These approaches offer a higher accuracy than PQ or OPQ. They however use (1) a different process to learn quantizer codebooks, and (2) a different ADC procedure. We discuss the compatibility of Quick ADC with these approaches. Using 4-bit sub-quantizers instead of 8-bit sub-quantizers ($b = 4$ instead of $b = 8$) requires doubling the m parameter (i.e., $m' = 2m$) to maintain a similar accuracy (accuracy mainly depends on the $m \times b$ product, Section 2.3.1). AQ, TQ and CQ codebook learning processes have complexities in $m \cdot 2^b$ or $m^2 \cdot 2^{2b}$. Therefore, doubling m is unlikely to cause issues for practical values ($m \in 8, 16$, $m' \in 16, 32$). Still, experiments are required to determine if the codebook learning process remains tractable, especially for 128-bit codes ($m' = 32$). CQ uses the same ADC procedure as PQ or OPQ. Therefore, Quick ADC is fully compatible with CQ. The ADC procedure of AQ and TQ have complexities in $m + m^2/2$ and $2m$. For these two approaches, doubling m strongly increases the number of operations per distance computation. Even if Quick ADC makes table lookups and additions fast thanks to SIMD, such an increase in the number of operations would lessen its benefit. In conclusion, combining Quick ADC with CQ is more interesting than combining Quick ADC with TQ or AQ. The main drawback of Quick ADC is that it incurs a small decrease in accuracy. As CQ offers a better accuracy than OPQ, it would contribute to eliminate this drawback. The resulting solution would therefore be very competitive, as it would offer both superior performance, thanks to Quick ADC, and a high accuracy, thanks to CQ.

6 Derived Quantizers

Contents

6.1	Motivation	69
6.2	Presentation	70
6.2.1	Overview	70
6.2.2	Training Derived Quantizers	71
6.2.3	ANN Search with Derived Quantizers	73
6.3	Evaluation	76
6.3.1	Experimental Setup	76
6.3.2	Small Datasets, 64-bit Codes	76
6.3.3	Small Dataset, 32-bit Codes	79
6.3.4	Large Dataset, 64-bit Codes	80
6.4	Discussion	81
6.4.1	Limitations	81
6.4.2	Comparison with Other Quantization Approaches	82
6.4.3	Comparison with ADC+R	82

6.1 Motivation

For most current ANN search use cases, product quantization with 8-bit sub-quantizers offers a sufficient accuracy. With Quick ADC, we have shown that even 4-bit sub-quantizers can be used. We have proposed two highly efficient scan procedures for these use cases, PQ Fast Scan and Quick ADC. However, some emerging use cases require a higher accuracy. Thus, descriptors generated by deep neural networks are increasingly popular for multimedia similarity search. When they are not PCA-compressed, such descriptors have several thousand dimensions (usually 4096). For these very high-dimensional vectors, a higher quantization accuracy is required. Therefore, 16-bit quantizers are used to increase product quantization accuracy. In addition, there has been a recent interest in using 32-bit codes instead of the usual 64-bit codes [8]. For 32-bit codes, 16-bit quantizers are required as 8-bit quantizers have a too low accuracy. However, using 16-bit quantizer causes a threefold increase in search time, which often outweighs their accuracy advantage. We have shown that this threefold increase in response time is caused by the fact that lookup tables used for distance computations are stored in the L3 cache when using 16-bit sub-quantizers. On the contrary, when using 8-bit sub-quantizers, lookup tables are stored in the much faster L1 cache (Section 3.1).

In this chapter, we introduce a novel approach, derived quantizers, that makes 16-bit quantizers as fast as 8-bit quantizers, while retaining their accuracy. The key idea behind our approach is to derive 8-bit quantizers, named derived quantizers, from the 16-bit quantizers so that they share the same short codes. Therefore, our approach does not incur any increase in memory usage. The codebook of derived quantizers can be seen as an approximate version of the codebook of the high-resolution 16-bit quantizers. This allows us to design a two-pass nearest neighbor search procedure that provides both a low response time and a high-accuracy. More specifically, this chapter addresses the following points:

- We present in detail the algorithm we designed to derive fast 8-bit quantizers from high-resolution 16-bit quantizers. We also describe how our two-pass nearest neighbor search procedure exploits derived quantizers to offer both a low response time and a high accuracy.
- We evaluate derived quantizers in the context of product quantization and optimized product quantization. We show that our approach achieves close response time to 8-bit quantizers, while having the same accuracy as 16-bit quantizers.
- We discuss the differences between our approach and other approaches that provide a high accuracy, e.g., additive quantization or tree quantization. We show that because it provides a large increase in accuracy, without significantly impacting response time, our approach compares favorably to the state of the art.

6.2 Presentation

6.2.1 Overview

The key idea behind our approach is to use product quantizers with 16-bit sub-quantizers, and associate an 8-bit *derived quantizer* with each 16-bit sub-quantizer. *Derived quantizers* are used to compute *compact lookup tables*, which fit the L1 cache. Compact lookup tables are used to compute approximate distances between the query vectors and short codes. A candidate set of the r_2 closest vectors is built using the computed approximate distances. A precise distance evaluation is then performed for all vectors of the candidate set. This precise distance evaluation relies on large lookup tables, stored in the L3 cache and computed from the codebooks of the 16-bit sub-quantizers. Performing a precise distance evaluation is slow, as it requires accessing the L3 cache. However, as precise distance evaluation is only performed for the vectors of the candidate set, the overall ANN search speed is high.

We denote $PQ_{m \times \bar{b}, b}$ a product quantizer using m sub-quantizers of b bits each, associated with m derived quantizers of $\bar{b}$ bits each. We denote $OPQ_{m \times \bar{b}, b}$ an optimized product quantizer with the similar properties. In this paper, we focus on $PQ_{m \times 8, 16}$ and $OPQ_{m \times 8, 16}$, i.e., product quantizers and optimized product quantizers with 16-bit sub-quantizers and 8-bit derived quantizers.

A naive way of training the 8-bit quantizers would be to train the codebooks $\bar{\mathcal{C}}_j, j \in \{0, \dots, m-1\}$ of 8-bit quantizers in the same way that the codebooks $\mathcal{C}_j$ of

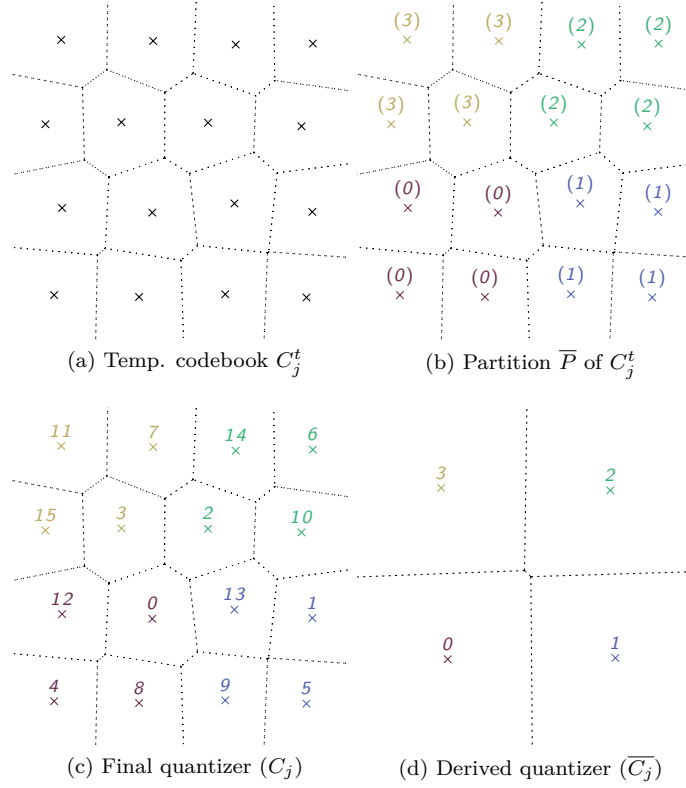


Figure 6.1: Derived quantizer training process

sub-quantizers are trained (Section 2.3.1). We would therefore need to encode high-dimensional vectors twice: once with the 16-bit sub-quantizers, and once with the 8-bit sub-quantizers. We would therefore obtain two codes per vector. This would double the memory use of the database, which would outweigh the benefit of our solution. Instead we *derive* the codebook of the 8-bit quantizers from the codebook of the 16-bit quantizers so that they share the same codes. Sub-quantizers are used to encode vectors and during ANN search, while the derived 8-bit quantizers are used exclusively for ANN search.

6.2.2 Training Derived Quantizers

A product quantizer uses m sub-quantizers, each having a different codebook $C_j, j \in \{0, \dots, m-1\}$. We derive the codebook of each derived quantizer $\bar{C}_j, j \in \{0, \dots, m-1\}$ from the codebook of the corresponding sub-quantizer. Thus, $\bar{C}_0$ is derived from C_0 etc.

The training process of the codebook of the j th sub-quantizer, with $k = 2^b$ centroids, and of the codebook of the j th derived quantizer, with $\bar{k} = 2^{\bar{b}}$ centroids is described in Algorithm 3. The KMEANS function is a standard implementation of the k-means algorithm. It takes a training set V_t and a parameter k , the number of clusters desired. It returns a codebook $\mathcal{C}$ and a partition P of the training set into k

6. Derived Quantizers

Algorithm 3 Training derived quantizers

```

1: function BUILD_QUANTIZERS( $V_t, \bar{k}, k$ )
2:    $\mathcal{C}_j^t, P \leftarrow \text{KMEANS}(V_t, k)$  ▷ Step 1
3:    $\bar{\mathcal{C}}_j, \bar{P} \leftarrow \text{KMEANS\_SAME\_SIZE}(\mathcal{C}_j^t, \bar{k})$  ▷ Step 2
4:    $\mathcal{C}_j \leftarrow \text{BUILD\_FINAL\_CODEBOOK}(\bar{P}, \bar{b})$  ▷ Step 3
5:   return  $\mathcal{C}_j, \mathcal{C}_j'$ 
6: function BUILD_FINAL_CODEBOOK( $\bar{P}, \bar{k}$ )
7:    $\bar{b} = \log_2(k)$ 
8:   for  $l \leftarrow 0$  to  $\bar{k}$  do
9:      $\bar{G} \leftarrow \bar{P}[\bar{i}]$ 
10:    for  $\bar{i} \leftarrow 0$  to  $|\bar{G}| - 1$  do
11:       $\mathcal{C}_j[\bar{i} \ll \bar{b} \mid l] = \bar{G}[\bar{i}]$  ▷  $\mid$  is binary or
▷  $\ll$  is bitwise left shift
12:   return  $\mathcal{C}_j$ 

```

clusters. We denote $P[i]$ the i th cluster of P . The `KMEANS_SAME_SIZE` function is a k-means variant which produces clusters G of identical sizes, i.e., $\forall G_l \in P, |G_l| = |G_0|$ [50]. For the sake of simplicity, Figure 6.1 illustrates the training process for $k = 16$ and $\bar{k} = 4$, although we use $k = 2^{16}$ and $\bar{k} = 2^8$ in practice. The training process takes three steps, described in the three following paragraphs.

Step 1. Train a temporary codebook $\mathcal{C}_j^t$ using the `KMEANS` function. Figure 6.1a shows the result of this step. Each point represents a centroid of $\mathcal{C}_j^t$. Vectors of the training set V_t are not shown. Implicitly, centroids of $\mathcal{C}_j^t$ have an index associated with them, which is their position in the list $\mathcal{C}_j^t$, i.e., the index of $\mathcal{C}_j^t[i]$ is i . As the indexes of centroids are not used in the remainder of the training process, they are not shown.

Step 2. Partition $\mathcal{C}_j^t$ into $\bar{k}$ clusters using the `KMEANS_SAME_SIZE` function. To do so, $\mathcal{C}_j^t$ is used as the training set argument for `KMEANS_SAME_SIZE`. Figure 6.1b shows the partition $\bar{P} = (\bar{G}_l), l \in \{0, \dots, \bar{k} - 1\}$ of $\mathcal{C}_j^t$. The number in parentheses above each centroid in the index of the cluster it has been assigned to i.e., the number l such that the centroid belongs to $\bar{G}_l$. The codebook $\bar{\mathcal{C}}_j$ returned by `KMEANS_SAME_SIZE` is the codebook of the derived quantizer, shown on Figure 6.1d. Each centroid $\mathcal{C}_j[l]$ of the derived quantizer is the centroid of the cluster $\bar{G}_l$. We obtain "centroids of centroids" because we used as codebook of centroids as the training set.

Step 3. Build the final codebook $\mathcal{C}_j$ by reordering the centroids of the temporary codebook $\mathcal{C}_j^t$. This reordering, or re-assignment of centroids indexes, is what allows sub-quantizers and derived quantizers to share the same code. The order of centroids in $\mathcal{C}_j$ must be such that the lowest $\bar{b}$ bits of the index assigned to each centroid of $\mathcal{C}_j$ corresponds to the cluster $\bar{G}_l$ it has been assigned to in step 2. If we denote $\text{low}_{\bar{b}}(i)$, the lower b bits of the index i , the order of centroids must obey the property:

$$\begin{aligned} \forall i \in \{0..k-1\}, \forall l \in \{0..\bar{k}-1\}, \\ \text{low}_{\bar{b}}(i) = l \Leftrightarrow \mathcal{C}_j[i] \in G_l \end{aligned} \tag{P1}$$

The `BUILD_FINAL_CODEBOOK` function produces an assignment of centroid indexes which obeys property P1 (Algorithm 3, line 11). Figure 6.1c shows the final assignment of centroid indexes. In this example, $k = 16$ and $\bar{k} = 4$ ($b = 4$ and $\bar{b} = 2$). The centroids belonging to cluster 1 (01 in binary) have been assigned the indexes 9 (1001), 13 (1101), 1 (0001), and 5 (0101). The lowest $\bar{b} = 2$ bits of 9, 13, 1 and 5 are 01, which matches the partition number (1, or 01). This property similarly holds for all partitions and all centroids.

This joint training process allows the derived quantizer $\overline{\mathcal{C}}_j$ to be used as an approximate version of $\mathcal{C}_j$ during the NN search process. To encode a vector $x \in \mathbb{R}^d$ into a short code, the codebooks $\mathcal{C}_j$ are used. The code c resulting from the encoding of vector x is such that for all $j \in \{0, \dots, m-1\}$, $\mathcal{C}_j[c[j]]$ is the closest centroid of x^j in $\mathcal{C}_j$. Our training process ensures that the centroid $\overline{\mathcal{C}}_j[\text{low}_{\bar{b}}(c[j])]$ is close to x^j . In other words, the centroid index assigned by the quantizer $\mathcal{C}_j$ remains meaningful in the derived quantizer $\overline{\mathcal{C}}_j$.

6.2.3 ANN Search with Derived Quantizers

ANN Search with derived quantizers takes two passes. The first pass builds a candidate set of $r2$ vectors, while the second pass reranks the candidate set to produce a final result set of r vectors. The first pass takes three steps: first, lookup tables are computed (Step 1.1) and quantized (Step 1.2). The full database is then scanned to build the candidate set (Step 1.3). The second pass takes a single step (Step 2.), which consists in performing a precise distance evaluation for all vectors of the candidate set, using the quantizers $\{\mathcal{C}_j\}_{j=0}^{m-1}$. Each step is detailed in the following paragraphs

Step 1.1 Compute a set of m short lookup tables, $\{\overline{D}_j\}_{j=0}^m$ from the derived quantizers $\{\overline{\mathcal{C}}_j\}_{j=0}^m$. Compact lookup tables are computed using the same `COMPUTE_TABLES` function used in the conventional ANN search procedure (Section 2.3.3, Algorithm 1). The compact lookup table $\overline{D}_j$ consists of the distances between the sub-vectors y^j and all centroids of the codebook $\overline{\mathcal{C}}_j$.

Step 1.2 Quantize floating-point distances in compact lookup tables $\{\overline{D}_j\}_{j=0}^{m-1}$ to 8-bit integers in order to build quantized lookup tables $\{Q_j\}_{j=0}^{m-1}$. This quantization step is necessary because our scan procedure (Step 1.3) uses a data structure optimized for fast insertion, `capped_buckets`, which requires distances to be quantized to 8-bit integers. We follow a quantization procedure similar to the one used in [2]. We quantize floating-point distances uniformly into 255 (0-254) bins between a $qmin$ and $qmax$ bound. All distances above $qmax$ are quantized to 255. We use the minimum distance across all lookup tables as the $qmin$ bound. To determine $qmax$, we compute the distance between the query vector and the $r2$ first vectors of the database. The greatest distance is used as the $qmax$ bound. Once $qmin$ and $qmax$ have been set, quantized lookup tables are computed as follows:

$$\forall j \in \{0, \dots, m-1\}, \forall i \in \{0, \dots, \bar{k}-1\},$$

$$Q_j[i] = \left\lfloor \frac{\overline{D}_j[i] - qmin}{qmax - qmin} \right\rfloor \cdot 255$$

Algorithm 4 ANN Search with derived quantizers

```

1: function NNS_DERIVED( $\{\mathcal{C}^j\}_{j=0}^m, \{\overline{\mathcal{C}^j}\}_{j=0}^m, db, y, r, r2$ )
2:    $\{\overline{D}^j\}_{j=0}^m \leftarrow \text{COMPUTE\_TABLES}(y, \{\overline{\mathcal{C}^j}\}_{j=0}^m)$  ▷ Step 1.1
3:    $\{Q_j\}_{j=0}^m \leftarrow \text{QUANTIZE}(\{\overline{D}^j\}_{j=0}^m, db, r2)$  ▷ Step 1.2
4:    $cand \leftarrow \text{SCAN}(db, \{Q_j\}_{j=0}^m, r2)$  ▷ Step 1.3
5:   return RERANK( $db, cand, \{\mathcal{C}^j\}_{j=0}^m, y, r, r2$ ) ▷ Step 2

6: function SCAN( $db, \{Q_j\}_{j=0}^m, r2$ )
7:    $cand \leftarrow \text{capped\_buckets}(r2)$ 
8:   for  $i \leftarrow 0$  to  $|db| - 1$  do
9:      $c \leftarrow db[i]$  ▷  $i$ th short code
10:     $d \leftarrow \text{ADC\_LOW\_BITS}(c, \{Q_j\}_{j=0}^m)$ 
11:     $cand.put(d, i)$ 
12:   return  $cand$ 

13: function ADC_LOW_BITS( $c, \{Q_j\}_{j=0}^m$ )
14:    $d \leftarrow 0$ 
15:   for  $j \leftarrow 0$  to  $m - 1$  do
16:      $d \leftarrow d + Q_j[\text{low}_{\overline{b}}(c[j])]$ 
17:   return  $d$ 

18: function RERANK( $db, cand, \{\mathcal{C}^j\}_{j=0}^m, y, r, r2$ )
19:    $i_{bucket} \leftarrow 0$ 
20:    $count \leftarrow 0$ 
21:    $neighbors \leftarrow \text{binheap}(r)$ 
22:    $\{D_j\}_{j=0}^{m-1} \leftarrow \{-1\}$ 
23:   while  $count < r2$  do
24:      $bucket \leftarrow cand.get\_bucket(i_{bucket})$ 
25:     for all  $i \in bucket$  do
26:        $d \leftarrow \text{ADC\_RERANK}(db[i], \{D_j\}_{j=0}^{m-1}, \{\mathcal{C}^j\}_{j=0}^m)$ 
27:        $neighbors.add((i, d))$ 
28:      $count \leftarrow count + bucket.size$ 
29:      $i_{bucket} \leftarrow i_{bucket} + 1$ 
30: function ADC_RERANK( $c, \{D_j\}_{j=0}^{m-1}, \{\mathcal{C}^j\}_{j=0}^m$ )
31:    $d \leftarrow 0$ 
32:   for  $j \leftarrow 0$  to  $m - 1$  do
33:     if  $D_j[c[j]] = -1$  then
34:        $D_j[c[j]] \leftarrow \|y^j - \mathcal{C}^j[c[j]]\|^2$ 
35:      $d \leftarrow d + D_j[c[j]]$ 
return  $d$ 

```

Step 1.3 Scan the full database to build a candidate set of $r2$ vectors using the quantized lookup tables (Algorithm 4, line 6). Our scan procedure is similar to the scan procedure of the conventional ANN search algorithm (Algorithm 1), apart from two differences. The first difference is that the ADC_LOW_BITS function is used to compute distances in place of the ADC function. The ADC_LOW_BITS function masks the lowest $\overline{b}$ bits of centroids indexes $c[j]$ to perform lookups in quantized tables (Algorithm 4, line 16), instead of using the full centroids indexes to access the full lookup tables. The second difference is that candidates are stored in a data

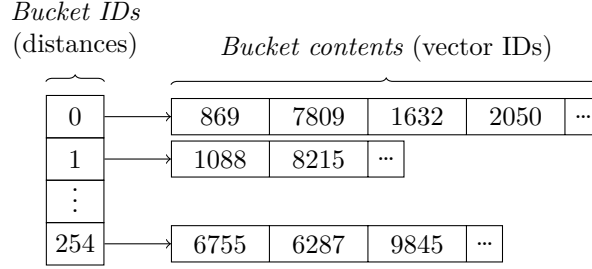
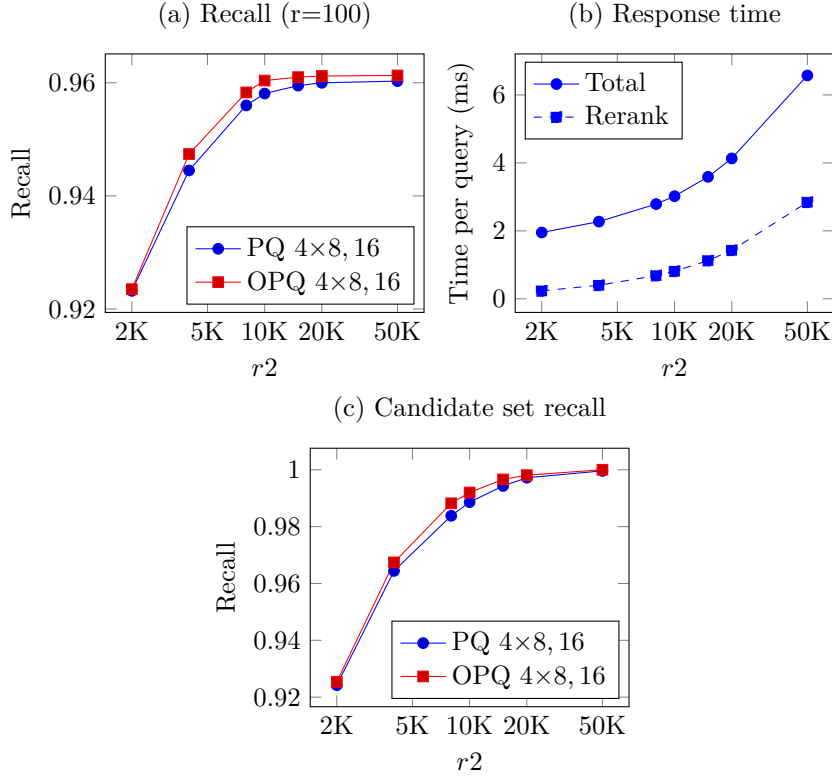


Figure 6.2: Capped buckets data structure

structure optimized for fast insertion, capped buckets (Algorithm 4, line 7), instead of a binary heap. Capped buckets consists of an array of buckets, one for each possible distance (0-254). Each bucket is a list of vector IDs. The put operation (Algorithm 4, line 11) involves retrieving the bucket d , and appending the vector ID i to the list. Because adding a vector to a capped buckets data structure requires much less operations than adding a vector to a binary heap, it is much faster. It however requires distances to be quantized to 8-bit integers, which are used as bucket IDs. This not an issue, as quantizing distances to 8-bit integers has been shown not to impact recall significantly [2]. On the contrary, a fast insertion is highly beneficial because the candidate set is relatively large (typical $r_2=10K-200K$) in comparison with the final result set (typical $r=10-100$). Therefore, many insertions are performed in the candidate set. To avoid the capped_buckets structure to grow indefinitely, we maintain an upper bound on distances (i.e., bucket IDs). Vectors having distances higher than the upper bound are discarded instead of being inserted in the capped_bucket. We used the distance (i.e., bucket ID) of the r_2 -th farthest vector in the capped_buckets as upper bound.

Step 2. Extract r_2 vectors from the capped buckets, and perform a precise distance evaluation using the full quantizers $\{\mathcal{C}_j\}_{j=0}^{m-1}$. This precise distance evaluation is used to build the result set of nearest neighbors, named *neighbors* (Algorithm 4, line 21). We iterate over capped buckets by increasing bucket ID i_{bucket} (Algorithm 4, line 24 and line 29). We process bucket 0, bucket 1, etc. until r_2 vectors have been processed. When processing a bucket, we iterate over all vectors IDs stored in this bucket (Algorithm 4, line 25), and compute precise distances using the ADC_RERANK function. This function is similar to the ADC function used in the conventional search process (Algorithm 1). However, here, we do not pre-compute the full lookup tables $\{D_j\}_{j=0}^{m-1}$ but rely on a dynamic programming technique. We fill all tables with the value -1 (Algorithm 4, line 22), and compute table elements on demand. Whenever, the value -1 is encountered during a distance computation (Algorithm 4, line 33), it means that this table element has not yet be computed. The appropriate centroid to sub-vector distance is therefore computed and stored in the appropriate lookup table (Algorithm 4, line 34). This strategy is beneficial because it avoids computing the full $\{D_j\}_{j=0}^{m-1}$ tables, which is costly (Table 3.2). In the case of PQ $m \times 8, 16$, a full lookup table D_j comprises a large number of elements $k = 2^{16} = 65536$, but only a small number are accessed (usually 5%-20%). This is because vectors of the candidate set are relatively close to the query vector, and thus their short codes tend to have similar indexes.

Figure 6.3: Impact of r_2 on recall and response time (SIFT1M, 64-bit codes)

6.3 Evaluation

6.3.1 Experimental Setup

All ANN search methods evaluated in this section were implemented in C++. We use gcc version 5.3, with the options `-O3 -ffast-math -m64 -march=native`. For linear algebra primitives, we use the ATLAS library version 3.10.2, of which we compiled an optimized version for our system. We trained the full codebooks $\mathcal{C}_j$ of product quantizers and optimized product quantizers using the implementation¹ of the authors of [8, 10]. We use two datasets²: a small dataset of 1 million vectors (SIFT1M), and a large dataset of 100 million vectors (SIFT100M), consisting of the first 100 million vectors of the BIGANN dataset. We experiment with 32-bit codes and 64-bit codes. All experiments were run on workstation equipped with an Intel Xeon E5-1650v3 CPU and 16 GiB of RAM (DDR4 2133Mhz).

6.3.2 Small Datasets, 64-bit Codes

Our ANN search procedure utilizing derived quantizers takes place in two passes: (1) a candidate set of r_2 vectors is built by scanning the full database, and (2) a

1. <https://github.com/arbabenko/Quantizations>

2. <http://corpus-texmex.irisa.fr/>

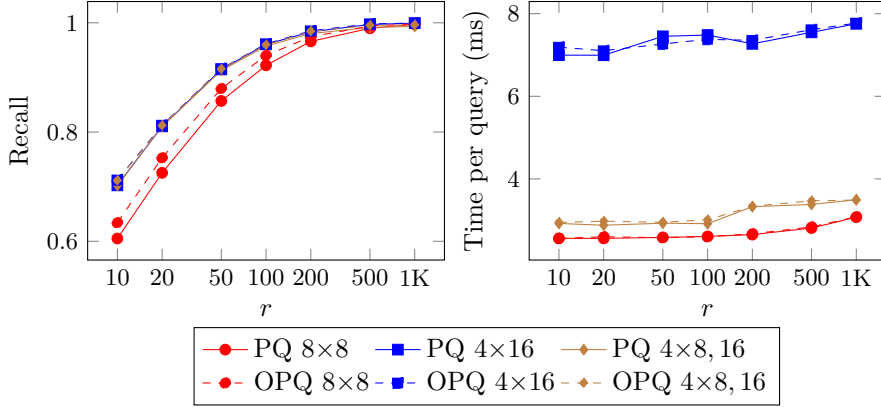
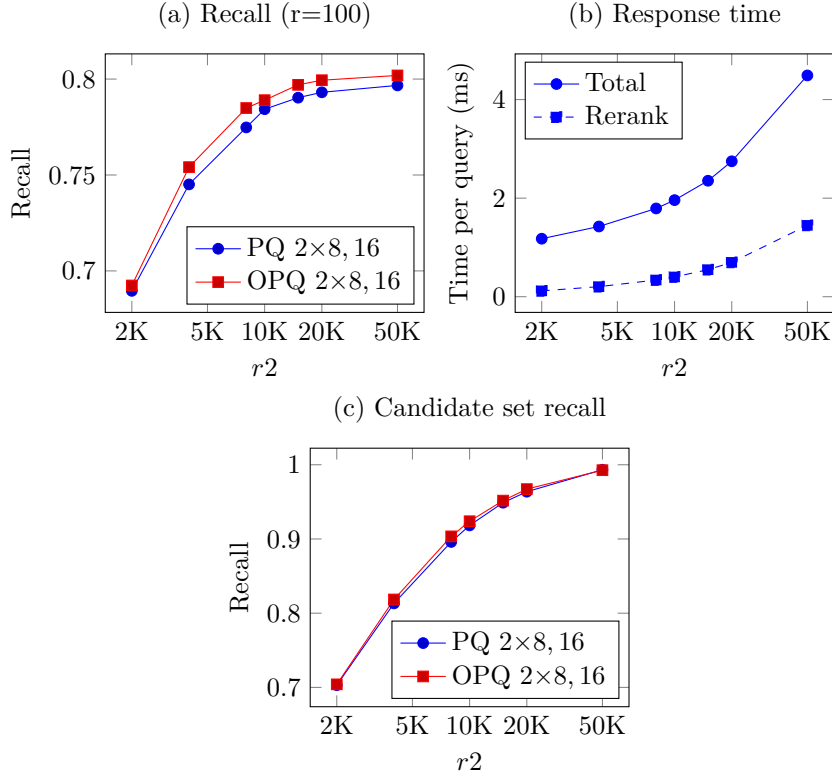


Figure 6.4: Recall and response time (SIFT1M, 64-bit codes)

final result set of r nearest neighbors is built by reranking the candidate set using high-accuracy quantizers. Both the parameters r and r_2 impact response time and recall. By contrast, the conventional ANN search procedure depends on a single parameter: r , the number of nearest neighbors in the final result set. In the remainder of this section we denote our approach (16-bit quantizers with derived 8-bit quantizers) $PQ_{m \times 8, 16}$ for product quantization, and $OPQ_{m \times 8, 16}$ for optimized product quantization.

Impact of r_2 We first study the impact of r_2 on recall and accuracy. To do so, we set $r = 100$ and vary r_2 between 2K and 50K. The recall increases with r_2 , until a point where it stabilizes (Figure 6.3a). The total response time is also very sensitive to r_2 ; it suffers a threefold increase between $r_2 = 2K$ and $r_2 = 50K$ (Figure 6.3b). This increase in response time comes from two factors: (1) an increased time spent building the candidate set (first pass) and (2) an increased time spent reranking vectors of the candidate set (second pass). Reranking the vectors of the candidate set requires performing lookups in the full lookup tables D_j , which is costly as they are stored in slow cache levels (Section 6.2.3). Moreover, as the elements of the tables D_j are computed on dynamically, a large candidate set means more costly centroid-to-subvector distances will need to be computed. Thus, the reranking time strongly increases with r_2 (Figure 6.3b) and plays a major role in the increase of the response time. The other cause of the increase in response time is the increased time spent building the candidate set (first pass). The number of distance computations performed in the first pass is independent of r_2 (Algorithm 4). However, the time spent in the first pass still increases with r_2 because a large r_2 means a large candidate set. Thus, more insertions in the candidate set are performed, which increases response time.

Due to its significant impact on both response time and recall, it is essential to fine-tune r_2 to obtain both a high recall and low response time. To determine r_2 , we measure the recall of the first pass of our ANN search procedure depending on r_2 (Figure 6.3c). We only execute the first pass of our ANN search procedure i.e., we build the candidate set but we do not rerank it. We use this candidate set as result set, and measure the recall. From Figure 6.3a and Figure 6.3c, it may seem that the first pass of our ANN search procedure (Figure 6.3c) achieves a higher recall

Figure 6.5: Impact of r_2 on recall and response time (SIFT1M, 32-bit codes)

than the full procedure (Figure 6.3). However, this is not the case. In the case of the full procedure (Figure 6.3a), the result set comprises $r = 100$ vectors, while in the case of the first pass (Figure 6.3c), the candidate set is used as result set, and thus comprises $r_2 = 2K-50K$ vectors. Our goal is to select r_2 values as small as possible but which preserve the recall of PQ4×16. For a given r value, we set r_2 such that the recall of the first pass (Figure 6.3c) is 5-10% higher than the recall@ r of PQ4×16 (Figure 6.4). For $r \leq 100$, PQ4×16 achieves a recall@ $r \leq 0.9$. Therefore, we set $r_2 = 9000$ (recall of first step is 0.98) for $r \leq 100$. For $r > 100$, PQ4×16 achieves very high recalls. Thus, for these cases we set $r_2 = 120000$ (recall of first step is 0.99).

Comparison with (O)PQ4×16 and (O)PQ8×8 PQ4×8, 16 and OPQ4×8, 16 offer the same recall as PQ4×16 and OPQ4×16 (Figure 6.4), while maintaining a response time close to PQ8×8 and OPQ8×8 (Figure 6.4), thus demonstrating the effectiveness of derived quantizers. Compared to (O)PQ4×8, 16, (O)PQ4×16 is 2.5 times faster, while offering the same recall. At $r = 50$, PQ4×16 offers a 7% increase in recall for a 196% increase in response time, compared to PQ8×8. By contrast, PQ4×8, 16 also offers a 7% increase in recall at $r = 50$, but the increase in response time is limited to 13%. The response time of (O)PQ4×8, 16 increases at $r = 200$ because we increased r_2 from 9000 to 12000 at this point. This increase is necessary for (O)PQ4×8, 16 to offer the same recall as (O)PQ4×16 beyond $r = 200$ (Section

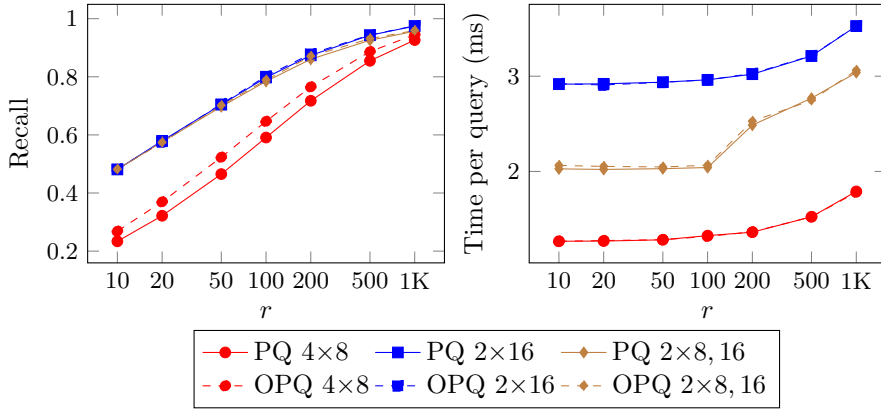


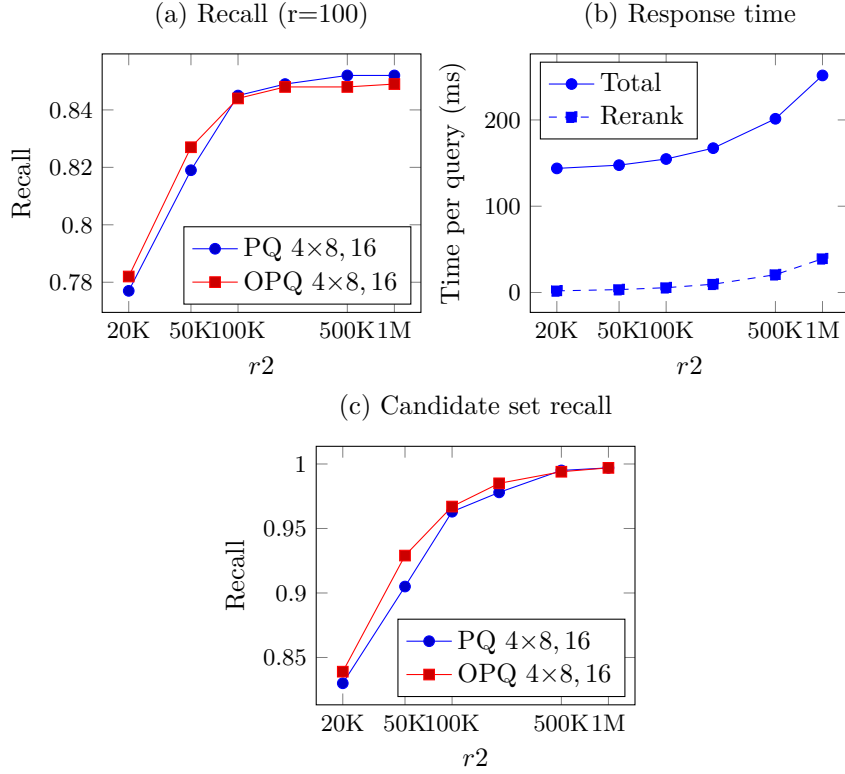
Figure 6.6: Recall and response time (SIFT1M, 32-bit codes)

6.3.2). Lastly, we observe that OPQ8×8 offers a higher recall than PQ4×16, but OPQ4×16 does not offer a higher recall than PQ4×16 (Figure 6.4). This is because OPQ optimizes the distribution of information between the m sub-spaces of a product quantizers. The greater the number m of sub-spaces, the larger is the gain provided by OPQ. For PQ8×8 ($m=8$ sub-spaces), OPQ is able to provide a noticeable gain, but for PQ4×16 ($m=8$ sub-spaces), the gain is too small to be visible.

6.3.3 Small Dataset, 32-bit Codes

Like with 64-bit codes, both recall and response time increase with r^2 (Figure 6.5a and Figure 6.5b). However, the second pass (reranking pass) represents a greater part of the total time for 32-bit codes than for 64-bit codes (Figure 6.5b). This is because the first pass (building the candidate set) is faster for 32-bit than for 64-bit codes, while the second pass has the same cost for the two code sizes. The first pass is faster for 32-bit codes (PQ2×8,16) because each distance computation requires $m = 2$ memory accesses instead of the $m = 4$ cache accesses required for 64-bit codes (PQ4×8,16). Even with the conventional search process, PQ2×16 (Figure 6.6) is faster than PQ4×16 (Figure 6.4). This is partly due to the lower number of cache accesses, but also to the fact that for PQ2×16, lookup tables fit the L2 cache, which is faster than the L3 cache used by PQ4×16 (Table 3.2, Section 3.1). Overall, the higher cost of the reranking step and the relatively low cost of PQ2×16 make the difference in response time between PQ2×16 and PQ4×16 lower. We determine r^2 values using the rule describe in Section 6.3.2. We set $r^2 = 10000$ for $r \leq 100$, $r^2 = 15000$ for $r = 200$, $r^2 = 17000$ for $r = 500$ and $r^2 = 20000$ for $r = 1000$.

Like for 64-bit codes, (O)PQ2×8,16 offers the same recall as (O)PQ2×16 (Figure 6.6). However, the relative difference in response time is higher for (O)PQ2×8,16 than for (O)PQ2×16 (Figure 6.6). At $r = 50$, PQ2×16 offers a 52% increase in recall for a 123% increase in response time, compared to PQ4×8. PQ2×8,16 also offers a 50% increase in recall at $r = 50$, for a 53% increase in response time. If our approach leads to a higher increase in response time for 32-bit codes than for 64-bit codes (50% versus 13%), it also leads to a higher increase in recall (50% versus 7%). This is because 32-bit codes lead to a higher quantization error than 64-bit codes.

Figure 6.7: Impact of r_2 on recall and response time (SIFT100M, 64-bit codes)

Thus, any technique to reduce the quantization error has more impact on 32-bit codes than on 64-bit codes.

6.3.4 Large Dataset, 64-bit Codes

To conclude our evaluation section, we evaluate our approach on a large dataset of 100 million vectors. Like for the small dataset, recall and response time increase with r_2 (Figure 6.7a and Figure 6.7b). However, contrary to the small dataset, the reranking time (second pass) only represents a small fraction of the response time in this case. Building the candidate set (first pass) requires performing a distance evaluation for all codes stored in the database. As the large is 100 times larger than the small database, the cost of this step is multiplied by 100. By contrast, the size of the candidate set is only multiplied by 10 ($r_2 = 20K-10M$ for the large dataset versus $r_2 = 2K-50K$). As a consequence, the second pass has less impact on response time for the large dataset than for the small dataset. In addition, response time increases more slowly with r_2 for the large dataset than for the small dataset. For the small dataset, the response time is multiplied by 3.4 between $r_2 = 2K$ and $r_2 = 50K$, while for the large dataset, the response time is multiplied by 1.8 between $r_2 = 20K$ and $r_2 = 1M$. Thus, it is less necessary to fine-tune r_2 . We set $r_2 = 300K$ for all r values.

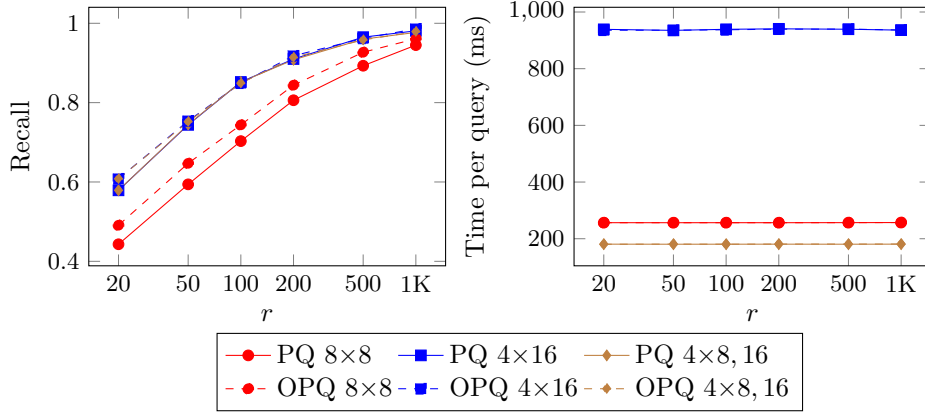


Figure 6.8: Recall and response time (SIFT100M, 64-bit codes)

Like for the small dataset, (O)PQ4×8, 16 provides the same recall as (O)PQ4×16 (Figure 6.8). However, the response time of (O)PQ4×8, 16 is lower than the response time of (O)PQ8×8 instead of being slightly higher (Figure 6.8). At $r = 50$, PQ4×16 offers a 25% increase in recall for a 270% increase in response time. PQ4×8, 16 offers a 25% increase in recall *and* a 42% decrease in response time. With PQ4×8, 16, the first pass of our ANN search procedure consists in performing distance computations between the query vector and 4×8 codes. Therefore, the first pass of our ANN search procedure is about two times faster than the conventional search procedure with 8×8 codes. For the small dataset, the time gain is outweighed by the large amount of time spent in the second pass. For the large dataset, the second pass only represents a fraction of the total response time (Figure 6.7b). The overall response is thus lower than the response time of the conventional ANN search procedure in the case of larger datasets.

6.4 Discussion

In this section, we discuss the strengths and weaknesses of our approach and compare it with the state of the art.

6.4.1 Limitations

As shown in Section 6.3, the use of 8-bit derived quantizers combined with 16-bit quantizers does not significantly increase response time while providing a substantial increase in recall, especially on large datasets. However, our approach still requires encoding vectors with 16-bit quantizers, and training 16-bit quantizers. This impacts both the vector encoding time and the codebook training time. For instance, on our workstation, encoding 1 million vectors with a 4×16 product quantizer takes 150 seconds (0.15 ms/vector), while encoding 1 million vectors with 8×8 product quantizer takes 0.91 seconds (0.00091 ms/vector). Similarly, training the codebooks of a 4×16 product quantizer takes 4 minutes (50 k-means iterations, 100K SIFT descriptors), while training the codebooks of a 8×8 product quantizer takes 5 seconds. Nonetheless, it is unlikely that any of this disadvantages would have a practical impact. Codebooks are trained once and for all, therefore training time does not

have much impact, as long as it remains tractable. As regards encoding time, even with 4×16 product quantizers, encoding a vectors into its short code takes much less time than an ANN query (0.15ms versus 2-200ms). Moreover, many real-world workloads are read-mostly: ANN queries are performed more often than vectors are added to the database.

6.4.2 Comparison with Other Quantization Approaches

In this chapter, we combined derived quantizers with Product Quantization (PQ) and Optimized Product Quantization (OPQ). In Section 2.3.4, we presented derivatives of product quantization that have been recently introduced: Optimized Product Quantization (OPQ), Additive Quantization (AQ), Tree Quantization (TQ) or Composite Quantization (CQ). Like derived quantizers, these approaches decrease quantization error to offer a higher ANN search accuracy.

Our approach compares favorably to these new quantization approaches as it does not significantly increase response time and only moderately increases encoding time. On the contrary, AQ and TQ result in a more than twofold increase in ANN search time [10]. Moreover, AQ and TQ increase vector encoding time by multiple orders of magnitude. Thus encoding a vector into a 64-bit code takes more than 200ms (1333 times more than our approach) for AQ, and 6ms for TQ (40 times more than our approach) [9]. Even if encoding time is not as important as query response time, such an increase makes AQ and TQ hardly tractable for large datasets. Thus, AQ and TQ have not been evaluated on datasets of more than 1 million vectors [8, 10]. More importantly, our approach is essentially orthogonal to the approach taken by AQ, TQ or CQ. Therefore, derived quantizers may be combined with AQ, TQ or CQ, but only if the codebook learning processes remain tractable.

The codebooks learning process of AQ has a time complexity in $m^2 \cdot 2^{2b}$, thus this process is likely to be intractable for 16-bit quantizers ($b = 16$). For TQ, the codebook learning process has a complexity in $m \cdot 2^b$ and is thus likely to be tractable, but this needs to be confirmed by experiments. For its part, CQ requires data structures with a size in $m^2 \cdot 2^{2b}$. However, these data structures are mainly used to speed up computations, so it is possible to eliminate these data structures at the expense of a moderate increase in computation time. Adapting CQ to 16-bit quantizers is therefore a possible research direction. Combining the ADC process of AQ, TQ or CQ with derived quantizers is likely to be feasible. Using 16-bit quantizers enables halving the m parameter ($m' = m/2$) for a constant code size. This makes it easy to adapt the ADC procedures, even if they have complexities in $m + m^2/2$ or $2m$.

6.4.3 Comparison with ADC+R

The idea of building a relatively large candidate set and then reranking it to obtain a final result set was inspired by the ADC+R approach [55]. The first major difference between our approach and ADC+R is that ADC+R uses 8-bit sub-quantizers, while our approach uses 16-bit sub-quantizers combined with 8-bit derived quantizers. This allows our approach to offer a significantly higher accuracy. ADC+R mainly aims at making ANN search with 8-bit sub-quantizers faster. On the contrary, derived quantizers aim at providing a higher accuracy (16-bit sub-quantizers), while maintaining a constant speed (8-bit derived quantizers). Besides, ADC+R has been superseded

by other approaches such as multi-indexes (Section 2.3.2). The second major difference between our approach is that the techniques they use to build the candidate set and rerank it are completely unrelated. The key idea of ADC+R is to use two product quantizers: a first product quantizer pq_1 to encode vectors into short codes, and a second product quantizer pq_2 to encode the error vector $r(x) = x - pq_1(x)$ resulting from the quantization by pq_1 . A vector x is encoded into two codes. To maintain memory use constant, ADC+R uses $m/2$ sub-quantizers for pq_1 and pq_2 . Thus, a vector $x \in \mathbb{R}^d$ is encoded into two 4×8 codes, instead of a single 8×8 code. The candidate set is built using the conventional ANN search (Section 2.3.3) algorithm and the codes produced by pq_1 . To rerank the candidate set, approximations $\hat{x} = pq_1(x) + pq_2(r(x))$ of the input vectors x need to be rebuilt. The distance between the query vector y and approximations $\hat{x}$ are then computed in $\mathbb{R}^d$. These two operations are costly, therefore making reranking slow. Therefore, reranking a candidate set of 20000 vectors takes about 40ms with ADC+R. On the contrary, our approach encodes the vectors with a single product quantizer pq , which is built in such a way that codes can be either scanned with 16-bit sub-quantizers, or 8-bit sub-quantizers. This allows our approach to rerank a candidate set of 20000 vectors in less than 1.5ms.

7 Conclusion and Perspectives

Contents

7.1	Summary of the Thesis	85
7.1.1	Context	85
7.1.2	Contributions	86
7.2	Perspectives	87
7.2.1	Application to Product Quantization Derivatives	87
7.2.2	Exploitation of Future Hardware	89
7.2.3	Generalization to Other Algorithms	90

7.1 Summary of the Thesis

7.1.1 Context

In this thesis, we have proposed several contributions on the topic of nearest neighbor search in high dimensionality and at large scale. This problem is of particular importance in the current context of mass use of online services. The recent development of mass internet access, online social networks, and smartphone applications has enabled corporations to collect data produced by millions or even billions of users. This trend of large scale data collection, known as *big data* is often characterized by three adjectives: *volume*, *velocity* and *variety*. Distributed computing platforms like Hadoop and Spark, have been recently introduced to address the high volume of datasets and the high velocity at which new data is produced. This thesis focuses on the *variety* aspect of big data: users not only upload textual data (messages, chats, blog posts) but also multimedia data (image and video files).

High-dimensional nearest neighbor search is of particular importance in multimedia databases. Multimedia objects (images, videos, audio files) can be represented as high-dimensional feature vectors that capture their contents. Finding similar multimedia objects then comes down to finding multimedia objects that have similar feature vectors. Many other applications, such as recommender systems or scientific data processing require efficient high-dimensional nearest neighbor search techniques. However, nearest neighbor search remains a challenging problem, especially at large scale and in high-dimensional spaces. Thus, the notorious curse of dimensionality makes *exact* nearest neighbor search intractable. The current research focus is therefore on *approximate* nearest neighbor search solutions.

Product quantization is one the most efficient approximate nearest neighbor search approaches, and is at the basis of all contributions of this thesis. The key advantage of product quantization is that it compresses high-dimensional vectors into short codes, which enables large databases to be stored entirely in RAM. Product Quantization can therefore answer nearest neighbor queries without accessing the SSD or HDD, unlike other approaches such as LSH. This approach makes it possible to achieve low response times, because RAM typically has a 100-1000 times lower latency than SSDs. The contributions of this thesis bet on exploiting the capabilities of modern CPUs to further decrease response times offered by product quantization.

7.1.2 Contributions

Product quantization compresses high-dimensional vectors into short codes, and stores the entire database in RAM in the form of lists of short codes. To answer nearest neighbor queries, product quantization scans these lists for nearest neighbors. Product quantization therefore computes the distance between the query vector and every code in the lists. This process, named Asymmetric Distance Computation (ADC), remains CPU intensive, and does not exploit the capabilities of modern CPUs. On a single-core, ADC is able to scan lists of short codes at 2-3 GB/s while the theoretical single-core memory bandwidth is 12-16 GB/s. This demonstrates that ADC is strongly CPU bound. In this thesis, we first analyzed the factors that limit the performance of ADC (Performance Analysis). Based on this analysis, we subsequently proposed three contributions (PQ Fast Scan, Quick ADC, Derived Quantizers) that increase the performance of ADC in different scenarios.

Performance Analysis To compute distances, ADC relies on a set of cache-resident lookup tables. Each distance computation requires a small number of table lookups and additions. In the first part of our analysis, we have shown that even when lookup tables fit the fastest cache levels, cache accesses limit the performance of ADC. Moreover, our study demonstrates that the performance of ADC strongly depends on the cache level in which lookup tables are stored. The product quantization parameter b , the number of bits per quantizer (typical $b = 8 - 16$ bits), affects both the accuracy of nearest neighbor search and the cache level in which lookup tables are stored. For instance, we show that using $b = 16$ bits per quantizer yields a higher accuracy than using $b = 8$ bits per quantizer, but also causes lookup tables to be stored in the L3 cache instead of the L1 cache. This in turn translates into a threefold increase in response time. In the second part of our analysis, we show that ADC cannot be efficiently implemented in SIMD, including using `gather` instructions introduced in recent Intel Haswell CPUs. Our study concludes that both the large number of cache accesses it performs, and the impossibility of an SIMD implementation limit the performance of ADC. These limitations call for a modification of the ADC procedure to increase performance.

PQ Fast Scan The first solution we proposed to overcome the limitations of the conventional ADC algorithm is PQ Fast Scan. PQ Fast Scan achieves a 4-6 times speedup over ADC, while returning the exact same results. The key idea behind PQ Fast Scan is to replace costly cache accesses by much faster SIMD in-register shuffles. Using SIMD in-register shuffles requires lookup tables to be stored in SIMD registers, which are much smaller than cache. PQ Fast Scan therefore builds small tables that

fit SIMD registers. These small tables are used to compute lower bounds on distances. Lower bounds are then used to prune unneeded distance computations, thus limiting the number of cache accesses. To build small tables, PQ Fast Scan modifies the layout of the database in memory. This makes it incompatible with inverted indexes, a search acceleration technique commonly used in combination with product quantization.

Quick ADC The second solution we proposed is Quick ADC. Like PQ Fast Scan, Quick ADC achieves a 4-6 times speedup over the conventional ADC algorithm. Quick ADC also builds on the idea of replacing cache accesses by SIMD in-register shuffles. However, contrary to PQ Fast Scan, Quick ADC is compatible with inverted indexes. Instead of building small tables that can be used to compute lower bounds, Quick ADC uses 4-bit quantizers ($b = 4$). Using 4-bit quantizers ($b = 4$) instead of the commonly used 8-bit quantizers makes it easy to build lookup tables that fit SIMD registers, but causes a slight decrease in recall. While this decrease is generally small, it may still have a practical impact in very large databases. To eliminate this issue, we combine Quick ADC with optimized product quantization, a derivative of product quantization that yields a better accuracy. We show that when combined with optimized product quantization, the decrease of accuracy caused by Quick ADC is always small, including in the case of very large databases.

Derived Quantizers For most current use cases of nearest neighbor search, product quantization with 8-bit quantizers, or even 4-bit quantizers, offers a sufficient accuracy. Newer use cases may however require 16-bit quantizers. For instance, there has been a recent interest in using 32-bit codes, instead of 64-bit codes, to achieve higher compression ratios. In this case, the accuracy offered by 8-bit quantizers is too low and 16-bit quantizers become necessary. The main drawback of this approach is that 16-bit incur a threefold increase in response time. To tackle this issue, we have proposed derived quantizers. Derived quantizers make 16-bit quantizers as fast as 8-bit quantizers, while retaining their accuracy. The key idea behind our approach is to compute 8-bit derived quantizers that approximate the 16-bit high-resolution quantizers. To answer nearest neighbor queries, a small candidate set is built by scanning the database using the fast 8-bit derived quantizers. This candidate set is then reranked using the high-resolution 16-bit quantizers.

7.2 Perspectives

The contributions presented in this thesis open both short term and more long-term perspectives. Among short-term perspectives is the application of the fast scan techniques we developed to derivatives of product quantization, such as additive quantization or composite quantization. After evaluating the applicability of our techniques to these recent derivatives of product quantization, we discuss two long-term perspectives. We first review how our techniques could be adapted to exploit future hardware. We then examine how they can be generalized to other algorithms that rely heavily on lookup tables or other database algorithms.

7.2.1 Application to Product Quantization Derivatives

In this thesis, we proposed three contributions (PQ Fast Scan, Quick ADC, Derived Quantizers) that increase the speed of the ADC procedure in different scenarios. We

7. Conclusion and Perspectives

tested these solutions in the context of product quantization and optimized product quantization. Product Quantization (PQ) and Optimized Product Quantization (OPQ) use the same ADC procedure: m table lookups and m additions. Recently, compositional quantization models inspired by product quantization have been introduced by the research community. Additive Quantization (AQ), Tree Quantization (TQ) and Composite Quantization (CQ) are among the most promising approaches (Section 2.3.4). All these approaches offer a lower quantization error than product quantization or optimized product quantization. They however sometimes use a different ADC procedure. For instance, AQ requires about $m + m^2/2$ table lookups and additions, and TQ requires $2 \cdot m$ table lookups and additions. Moreover, all these new approaches use a different process to learn quantizer codebooks from the one of PQ. So far, these approaches have only been tested with 8-bit quantizers ($b = 8$). Some of our solutions use 4-bit quantizers (Quick ADC), or 16-bit quantizers (Derived Quantizers). Adapting the codebook learning processes of these new approaches to other quantizer sizes may therefore open challenges. In this section, we recap the research opportunities opened by the application of our solutions to these new compositional quantization models.

PQ Fast Scan PQ Fast Scan relies on the use of 8-bit sub-quantizers, and all these new quantization approaches have also been tested with 8-bit sub-quantizers. Therefore, there is no need to adapt the codebook learning process of AQ, TQ or CQ. In addition, CQ uses a similar ADC procedure to the one of PQ and OPQ. Therefore, adapting PQ Fast Scan to CQ is straightforward. We have shown that the case of TQ or AQ is more challenging (Section 4.4.2). The ADC procedure of TQ and AQ requires more table lookups ($m + m^2/2$ or $2 \cdot m$), than the ADC procedure of PQ or OPQ. This means that a different combination of code grouping and minimum tables has to be used. Determining if PQ Fast Scan has a high enough pruning power under these conditions is therefore a possible research direction. If this is the case, PQ Fast Scan would offer a high speedup for AQ and TQ. The SIMD implementation of additions allowed by PQ Fast Scan would highly benefit AQ and TQ, as they require a high number of additions ($m + m^2/2$ or $2 \cdot m$).

Quick ADC Quick ADC uses 4-bit sub-quantizers while the recently introduced compositional quantization approaches have been tested with 8-bit sub-quantizers only. This means that it is necessary to adjust the codebook learning process of AQ, TQ or CQ. Using 4-bit sub-quantizers instead of 8-bit sub-quantizers requires doubling the m parameter to maintain a comparable accuracy. This impacts the complexity of the codebook learning process of AQ, TQ and CQ. We have shown that this process should however remain tractable (Section 5.4.2), but experiments are required to determine if it is the case in practice. We have also shown that Quick ADC is unlikely to strongly benefit the ADC procedure of AQ and TQ. For these two quantization approaches, doubling the m parameter causes a large increase in the number of additions ($m + m^2/2$ or $2 \cdot m$), lessening the benefit of Quick ADC (Section 5.4.2). In conclusion, the most interesting research opportunity consists in combining Quick ADC and CQ, as its ADC procedure requires only m additions. The resulting solution would offer both superior performance, thanks to Quick ADC, and a high accuracy, thanks to CQ.

Derived Quantizers The key idea of derived quantizers is to use 16-bit quantizers instead of the common 8-bit quantizers to increase accuracy. Therefore, derived quantizers have the same goal as AQ, TQ or CQ. Because they increase accuracy without impacting response time, derived quantizers compare favorably to these approaches. More importantly, the approach of derived quantizers is orthogonal to these other approaches. Derived quantizers may therefore be combined with AQ, TQ or CQ, but only if the codebook learning processes remain tractable for 16-bit sub-quantizers. We have shown that the codebook learning process of TQ and CQ is likely to remain tractable, while the codebook learning process of AQ is likely to be intractable (Section 6.4.2). Therefore, the most interesting research opportunity is to combine derived quantizers with TQ or CQ. Lastly, using 16-bit quantizers makes it possible to divide the m parameter by 2 ($m' = m/2$), while maintaining a greater accuracy. Thus the ADC procedures can be easily adapted, as they have complexities in $m + m^2/2$ or $2m$.

7.2.2 Exploitation of Future Hardware

Future SIMD instruction sets The upcoming Intel Xeon Skylake Purley, expected in 2017, will include support for the new AVX-512 SIMD instruction set [34, 38]. The AVX-512 instruction set will provide twice as wide SIMD registers (512 bits) as the current AVX2 SIMD instruction set (256 bits). In addition, AVX-512 will offer new SIMD instructions that do not exist yet in AVX2. For instance, AVX-512 will offer a full-width SIMD in-register shuffle allowing 512-bit small tables (64×8 bits). In comparison, the current AVX2 instruction set only offers half-width SIMD in-register shuffles, allowing lookups in 256-bit small tables (16×8 bits). AVX-512 will increase the applicability of PQ Fast Scan, and the accuracy of Quick ADC. PQ Fast Scan requires inverted lists of at least 3 million codes to be efficient, hindering the compatibility with inverted indexes (Section 4.2.2). This size constraint stems from the use of code grouping. More specifically, we have shown that the minimum size of inverted lists depends on the size of small tables stored in SIMD registers (Section 4.4.1). For 128-bit small tables (16×8), the minimum size is 3 million codes, while for 512-bit small tables (64×8), the minimum size would be 3000-12000 codes. Thus, AVX-512 would make PQ Fast Scan compatible with most inverted indexes configurations. Quick ADC causes a slight drop in accuracy because it uses 4-bit quantizers ($b = 4$) instead of the more common and more accurate 8-bit quantizers ($b = 8$). Relying on 512-bit small tables would allow Quick ADC to use 6-bit quantizers ($b = 6$), thus reducing the drop in accuracy. In addition to improving applicability or accuracy, AVX-512 (512-bit SIMD) will allow process two times more data per cycles than AVX2 (256-bit SIMD), or four times more data per cycles than SSE (128-bit SIMD). The benefit of AVX-512 is therefore twofold: increase in accuracy or applicability and improvement in performance.

Other Architectures So far, we have implemented and evaluated PQ Fast Scan on current Intel CPUs. However, other architectures, such as ARM CPUs are used on a massive scale in consumer devices such as smartphones or tablets [5]. ARM CPUs might also gain traction in the datacenter market in the future. PQ Fast Scan and Quick ADC require support for SIMD in-register shuffles, and SIMD saturated adds. In addition, they require an SIMD width of at least 128 bit. The ARM NEON instruction set is a 128-bit SIMD instruction set included in ARM Cortex-A CPUs, ARM's range of CPUs for smartphones and tablets [6, 7]. ARM NEON supports

7. Conclusion and Perspectives

in-register shuffles (VTBL and VTBX instructions) as well as saturated adds (VQADD instruction). Therefore, implementing PQ Fast Scan and Quick ADC with ARM NEON instructions is a possible research direction. Moreover, experiments would allow assessing the speedup obtained on ARM processors. In August 2016, ARM has announced its intent to provide very wide SIMD engines (up to 2048 bits) in its future generation of CPUs via the Scalable Vector Extensions (SVE) instruction set [15]. By allowing larger small tables, SVE would bring similar benefits as AVX-512 to ARM CPUs.

In 2013, IBM announced its intent to offer the POWER technology for licensing, and to allow third-parties to include POWER CPUs in their products [56]. So far, POWER CPUs were almost exclusively used in mainframes manufactured by IBM itself. This has triggered a surge of interest for the POWER architecture, and the OpenPOWER Foundation [48] was created to coordinate the efforts of all parties interested in POWER CPUs. The OpenPOWER Foundation now includes major players such as Google or NVIDIA, and the first POWER servers manufactured by third-parties are appearing. POWER servers might therefore enter the market in the near future. Like ARM, the POWER architecture offers 128-bit SIMD. In particular, SIMD in-register shuffles (`vperm`) and 8-bit saturated additions (`vaddubs`) are supported [30]. PQ Fast Scan and Quick ADC may therefore be implemented on POWER CPUs.

7.2.3 Generalization to Other Algorithms

When designing PQ Fast Scan and Quick ADC, we developed different techniques that exploit SIMD to accelerate nearest neighbor search. As a medium to long-term research perspective, these techniques may be used beyond the context of nearest neighbor search. More specifically, these techniques can be useful to algorithms that rely on lookup tables, or database workloads that process large amounts of data. In this section, we discuss how the techniques we developed in the context of PQ Fast Scan and Quick ADC can be generalized to other workloads.

SIMD in-registers shuffles The main idea behind PQ Fast Scan and Quick ADC is to store lookup tables in SIMD registers, while storing them in the L1 cache is generally considered as best practice for efficiency. Therefore, any algorithm relying on lookup tables is a candidate for applying this idea. Among practical uses of lookup tables is query execution in compressed databases. Compression schemes, either generic [49, 26, 1, 53] or specific (e.g., SAX for time series [40]), have been widely adopted in database systems. In the case of dictionary-based compression (or quantization), the database stores short codes. A dictionary (or codebook) holds the actual values corresponding to the short codes. Query execution then relies on lookup tables, derived from the dictionary. In this case, storing lookup tables in SIMD registers allows for better performance. If lookup tables are small enough (16 entries), they may be stored directly in SIMD registers, after quantization of their elements to 8-bit integers. Otherwise, it is possible to build small tables for different types of queries. For top-k queries, it is possible to build small tables enabling computation of lower or upper bounds. Like in PQ Fast Scan, lower bounds can then be used to limit L1-cache accesses. To compute upper bounds instead of lower bounds, maximum tables can be used instead of minimum tables. For approximate aggregate queries

(e.g., approximate mean), tables of aggregates (e.g., tables of means) can be used instead of minimum tables.

Saturated Integer Arithmetic Another idea behind PQ Fast Scan is to use 8-bit saturated arithmetic. This idea can be applied for queries which do not use lookup tables, such as queries executed on uncompressed data. Top-k queries require exact score evaluation for a small number of items, so 8-bit arithmetic can be used to discard candidates. Similarly, 8-bit arithmetic may provide enough precision for approximate queries. In the context of SIMD processing, 8-bit arithmetic allows processing 4 times more data per instruction than 32-bit floating-point arithmetic and thus provides a significant speedup.

Bibliography

- [1] D. J. Abadi, S. R. Madden, and M. C. Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. In *SIGMOD*, 2006.
- [2] F. André, A.-M. Kermarrec, and N. Le Scouarnec. Cache locality is not enough: High-Performance Nearest Neighbor Search with Product Quantization Fast Scan. *PVLDB*, 9(4), 2015.
- [3] Apache Software Foundation. Apache Hadoop. <https://hadoop.apache.org/>, 2016. Accessed 30 June 2016.
- [4] Apache Software Foundation. Apache Spark. <https://spark.apache.org/>, 2016. Accessed 30 June 2016.
- [5] ARM. ARM Processors. <https://www.arm.com/products/processors>. Accessed 02 October 2016.
- [6] ARM. NEON. <http://www.arm.com/products/processors/technologies/neon.php>. Accessed 02 October 2016.
- [7] ARM. NEON Instructions. <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0489e/CJAJIIGG.html>. Accessed 02 October 2016.
- [8] A. Babenko and V. Lempitsky. Additive Quantization for Extreme Vector Compression. In *CVPR*, 2014.
- [9] A. Babenko and V. Lempitsky. The Inverted Multi-Index. *TPAMI*, 37(6), 2015.
- [10] A. Babenko and V. Lempitsky. Tree quantization for large-scale similarity search and classification. In *CVPR*, 2015.
- [11] H. Bay, T. Tuytelaars, and L. Van Gool. SURF: Speeded Up Robust Features. In *ECCV*, 2006.
- [12] J. S. Beis and D. G. Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *CVPR*, 1997.
- [13] J. L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *CACM*, 18(9), 1975.
- [14] CERN. Computing. <https://home.cern/about/computing>. Accessed 27 June 2016.
- [15] I. Cutress. ARM Announces ARM v8-A with Scalable Vector Extensions: Aiming for HPC and Data Center. *Anandtech*, August 2016.

- [16] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. *SCG*, 2004.
- [17] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.
- [18] Facebook. Facebook Q1 2016 Results. <https://investor.fb.com/financials/?section=annualreports>, 2016. Accessed 27 June 2016.
- [19] Fortune. Amazon’s recommendation secret. <http://fortune.com/2012/07/30/amazons-recommendation-secret/>, 2016. Accessed 27 June 2016.
- [20] Friedman, Jerome H. and Bentley, Jon Louis and Finkel, Raphael Ari. An Algorithm for Finding Best Matches in Logarithmic Expected Time. *TOMS*, 3(3), 1977.
- [21] K. Fukunaga and P. M. Narendra. A Branch and Bound Algorithm for Computing k-Nearest Neighbors. *TC*, C-24(7), 1975.
- [22] J. Gan, J. Feng, Q. Fang, and W. Ng. Locality-sensitive Hashing Scheme Based on Dynamic Collision Counting. In *SIGMOD*, 2012.
- [23] T. Ge, K. He, Q. Ke, and J. Sun. Optimized Product Quantization for Approximate Nearest Neighbor Search. In *CVPR*, 2013.
- [24] T. Ge, K. He, Q. Ke, and J. Sun. Optimized Product Quantization. *TPAMI*, 36(4), 2014.
- [25] A. Gionis, P. Indyk, and R. Motwani. Similarity Search in High Dimensions via Hashing. In *VLDB*, 1999.
- [26] G. Graefe and L. D. Shapiro. Data Compression and Database Performance. In *SAC*, 1991.
- [27] R. Gray. Vector Quantization. *IEEE ASSP Magazine*, 1(2), 1984.
- [28] T. Hey, S. Tansley, and K. Tolle, editors. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009.
- [29] J. Hofmann, J. Treibig, G. Hager, and G. Wellein. Comparing the Performance of Different x86 SIMD Instruction Sets for a Medical Imaging Application on Modern Multi- and Manycore Chips. In *WPMVP*, 2014.
- [30] IBM. *Power ISA Version 2.07*, May 2013.
- [31] Instagram Blog. Celebrating a Community of 400 Million. <http://blog.instagram.com/post/129662501137/150922-400million>, 2015. Accessed 27 June 2016.
- [32] Intel. Intel Intrinsic Guide. <https://software.intel.com/sites/landingpage/IntrinsicsGuide/>.
- [33] Intel. *User and Reference Guide for the Intel C++ Compiler 15.0*.
- [34] Intel. Intel AVX-512 instructions. <https://software.intel.com/en-us/blogs/2013/avx-512-instructions>, 2013. Accessed 02 October 2016.

-
- [35] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, June 2016.
 - [36] Intel. *Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 1: Basic Architecture*, September 2016.
 - [37] Intel. *Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z*, September 2016.
 - [38] Intel. *Intel Architecture Instruction Set Extensions Programming Reference*, September 2016.
 - [39] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *TPAMI*, 33(1), 2011.
 - [40] J. Lin, E. Keogh, L. Wei, and S. Lonardi. Experiencing SAX: a novel symbolic representation of time series. *Data Mining and Knowledge Discovery*, 15(2), 2007.
 - [41] Y. Liu, J. Cui, Z. Huang, H. Li, and H. Shen. SK-LSH: An Efficient Index Structure for Approximate Nearest Neighbor Search. *PVLDB*, 7(9), 2014.
 - [42] S. Lloyd. Least squares quantization in PCM. *Transactions on Information Theory*, 28(2), 1982.
 - [43] D. Lowe. Object recognition from local scale-invariant features. In *ICCV*, 1999.
 - [44] M. Muja and D. G. Lowe. Scalable Nearest Neighbour Algorithms for High Dimensional Data. *TPAMI*, 36(X), 2014.
 - [45] D. Nister, H. Stewenius, D. Nist, and H. Stew. Scalable Recognition with a Vocabulary Tree. In *CVPR*, 2006.
 - [46] M. Norouzi and D. J. Fleet. Cartesian K-Means. In *CVPR*, 2013.
 - [47] A. Oliva and A. Torralba. Modeling the Shape of the Scene: A Holistic Representation of the Spatial Envelope. *IJCV*, 42(3), 2001.
 - [48] OpenPOWER Foundation. <http://openpowerfoundation.org/>.
 - [49] M. A. Roth and S. J. Van Horn. Database compression. *ACM SIGMOD Record*, 22(3), 1993.
 - [50] E. Schubert. Same-size k-means variation, 2012. <http://elki.dbs.ifi.lmu.de/wiki/Tutorial/SameSizeKMeans>.
 - [51] C. Silpa-Anan and R. Hartley. Optimised KD-trees for fast image descriptor matching. In *CVPR*, 2008.
 - [52] J. Sivic and A. Zisserman. Video Google: a text retrieval approach to object matching in videos. In *ICCV*, 2003.
 - [53] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O’Neil, et al. C-store: a column-oriented DBMS. In *VLDB*, 2005.

- [54] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Quality and Efficiency in High Dimensional Nearest Neighbor Search. In *SIGMOD*, 2009.
- [55] R. Tavenard, H. Jegou, M. Douze, and L. Amsaleg. Searching in one billion vectors: Re-rank with source coding. In *ICASSP*, 2011.
- [56] J. Walton. IBM Offers POWER Technology for Licensing, Forms OpenPOWER Consortium. *Anandtech*, August 2013.
- [57] R. Weber, H.-J. Schek, and S. Blott. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. In *VLDB*, 1998.
- [58] Y. Xia, K. He, F. Wen, and J. Sun. Joint Inverted Indexing. In *ICCV*, 2013.
- [59] YouTube Engineering and Developers Blog. Machine learning for video transcoding. <https://youtube-eng.blogspot.fr/2016/05/machine-learning-for-video-transcoding.html>, 2016. Accessed 27 June 2016.
- [60] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *NSDI*, 2012.
- [61] T. Zhang, C. Du, and J. Wang. Composite Quantization for Approximate Nearest Neighbor Search. In *ICML*, 2014.

AVIS DU JURY SUR LA REPRODUCTION DE LA THESE SOUTENUE

Titre de la thèse:

Exploiting modern hardware for high-dimensional nearest neighbor search

Nom Prénom de l'auteur : ANDRE FABIEN

Membres du jury :

- Monsieur TRIANTAFILLOU Peter
- Monsieur MOSTEFAOUI Achour
- Monsieur THOMAS Gaël
- Madame KERMARREC Anne-Marie
- Monsieur LE SCOUARNEC Nicolas

Président du jury : M. Achour MOSTEFAOUI

Date de la soutenance : 25 Novembre 2016

Reproduction de la these soutenue

Thèse pouvant être reproduite en l'état

~~Thèse pouvant être reproduite après corrections suggérées.~~

Fait à Rennes, le 25 Novembre 2016

Signature du président de jury

Le Directeur,

M'hamed DRISSI



A. Mostefoui