République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieure et de la Recherche Scientifique
Université Abderrahmane Mira de Bejaia
Faculté des Sciences Exactes
Département d'informatique

Mémoire de fin d'étude

En vue de l'obtention du diplôme de master professionnel en informatique
Spécialité : Génie Logiciel
Sur le thème :

Jointure paralléle sur les processeurs graphiques (GPU) avec CUDA

Réalisé par :

- Mr. ALILECHE Hakim

Encadré par :

- Mme. TAHAKOURT Zineb
- Mr. SIDER Abderrahmane

Membres du jury

Président :
- Mr. AMROUN Kamal

Examinateurs :
- Mr. Mir Foudil
- Mr. OUZEGANE Redouane

Année : 2016 - 2017

Remerciements

Tout d'abord et avant tout, Je remercie Dieu le tout puissant qui m'a donné sagesse, savoir et patience. Il m'a donné aussi le courage et la volonté afin de réaliser ce modeste travail et de le mener à terme.

J'adresse mes remerciements particulièrement à mes parents qui n'ont pas cessé de m'encourager et de m'aider de toutes les façons possibles durant tout le long chemin de mes études.

Je remercie spécialement mes deux encadreurs pour leur orientation et leur disponibilité durant tout ce temps de travail.

Je remercie tous les enseignants qui ont contribué à ma formation.

Je tiens aussi à remercier, à l'avance, les membres du jury qui veulent bien me faire l'honneur d'évaluer mon travail.

Je remercie chaleureusement mes frères et sœurs et toute ma famille, ainsi que mes amis et toutes les personnes ayant contribué de près ou de loin à la réalisation de ce modeste travail.

Dédicaces

Je dédie ce modeste travail en signe de respect et de reconnaissance :

A mes chers parents, pour leurs sacrifices et soutiens moral et matériel afin d'atteindre mon objectif,

A ma défunte sœur et à tous mes frères, à ma sœur ainsi qu'à toute ma famille,

A Mes amis Kamel, Massy, Abdelkader, Sofiane, Djaaffer, Youva, Mouh, Salah, Mustapha, Falio, Lounes, Nassim Boukhalfa, Alilou, Rabah, Mohand, Zahir et à tous mes amis,

A toutes les personnes qui ont contribué de près ou de loin à la réalisation de ce modeste travail.

Table des matières

Introduction générale

De nos jours, l'utilisation des bases de données et des bigdata ne cesse d'augmenter, le temps de leur traitement augmente proportionnellement. Du fait du temps de réponse relativement important que prend le traitement des données, il est nécessaire de réduire ce temps en améliorant les algorithmes et/ou en utilisant d'autres méthodes de traitement.

Il existe plusieurs opérations sur les bases de données, comme la projection, la sélection, la jointure.
La jointure est parmi les opérations les plus utilisées et les plus couteuses en terme de temps et de mémoire. Notre travail consiste à traiter ce problème, où il est primordial de trouver des solutions pour diminuer le temps de traitement de la jointure sur les bases de données.

Deux approches majeures de résolution sont utilisées aujourd'hui : la distribution et la parallélisation. La distribution consiste à diviser et répartir les données sur plusieurs machines, ceci est très couteux et exige un réseau fiable. La parallélisation consiste à diviser et répartir l'opération de jointure sur plusieurs coeurs et/ou processeurs dans une même machine.

L'approche que nous avons adoptée dans notre travail est la parallélisation, plus précisément le parallélisme de données "SIMD" [1], réalisable sur un seul ordinateur. Cette approche est plus intéressante pour nous, dans la mesure où la configuration matérielle requise est à notre portée, à savoir, les processeurs (CPU) multi-cours et les processeurs graphiques dédiés (GPU).

L'objectif de notre travail est de proposer un nouvel algorithme de jointure baptisé « **JoinByID** », qui calcule la jointure entre des relations (tables) en exploitant les valeurs des identifiants des tuples. Ceci donne un gain très intéressant en coût en termes de temps et d'espace mémoire.
Nous avons implémenté cet algorithme en utilisant la technologie $CUDA$ afin de paralléliser l'opération. Ceci donne également un gain intéressant en terme de temps de calcul.
Des résultats d'expérimentation très intéressants sont obtenus et sont présentés au dernier chapitre.

Pour une bonne conduite de ce travail nous avons procédé comme suit :

- Une étude approfondie sur le fonctionnement de CUDA et l'architecture CUDA [2].
- Une étude des différents algorithmes de jointure.
- Implémentation d'un algorithme de jointure (JoinByID) en séquentiel et un autre en parallèle.

1. Simple Instruction Multiple Data, du classement de Flynn
2. **CUDA** est une technologie crée par Nvidia pour exécuter des calculs généraux sur GPU

- Implémentation d'une application $CUDA$ avec le langage $C++$ sous *linux*.
- Expérimentation en utilisant un benchmark de *Renault*.[6]

Ce mémoire est divisé en deux parties, *l'état de l'art* en premier et *Conception et implémentation* en second.

La première partie : *L'état de l'art* Cette partie est la partie théorique du travail. Elle est répartie en quatre chapitres, à savoir : *les architectures d'ordinateurs*, *CUDA*, *les algorithmes de jointure* et *les travaux déjà faits* sur le sujet.

Dans le chapitre « les architectures d'ordinateurs », nous présentons les architectures selon classification de **Flynn** et les types d'architectures d'ordinateur.
Le chapitre « CUDA »donne un aperçu sur les GPU, puis définit la technologie $CUDA$ et son architecture, ainsi que le principe de son fonctionnement.
Le chapitre « Les algorithmes de jointure »présente les algorithmes de jointure les plus utilisés, leurs avantages et les inconvénients de chacun de ces algorithmes.
Le dernier chapitre de cette partie intitulé « Les travaux déjà faits »présente quelques travaux faits dans le domaine des jointures.

La seconde partie : *Conception et implémentation* Cette partie constitue l'aspect de notre contribution. Elle est répartie en deux chapitres, à savoir : la *conception* du projet et de son *implémentation*.

Le chapitre « Conception » contient la structure de données , la méthode de traitement et ainsi que l'algorithme « JoinById »utilisés dans notre projet.
Le chapitre « Implémentation »présente les différents modes de fonctionnement de « *JoinByID* », à savoir, *CPU*, *GPU* et *Hybrid*. Nous le déroulerons sur un benchmark [3] « normalized Renault mod 26 ext »de Renault et présenterons les résultats de l'expérimentation.
Nous terminons le mémoire par une conclusion et perspectives.

3. **benchmark :** (étalon) est un point de référence servant à mesurer les performances d'un système pour le comparer à d'autres.

Première partie

État de l'art

Chapitre 1 | Architectures des ordinateurs

1.1 Introduction

Avec l'évolution extraordinaire des ordinateurs, plusieurs classifications d'architectures d'ordinateurs sont apparues, parmi elles, la classification de *Flynn*.

Dans ce chapitre, nous présentons les principaux types d'architecture d'ordinateur ainsi que la taxonomie de *Flynn* et cela pour aider à bien comprendre le travail effectué et plus précisément le type d'architecture utilisé.

1.2 Types d'architectures

On distingue aussi les architectures des ordinateurs par leur mode de traitement. Ce mode de traitement peut être séquentiel, parallèle ou distribuée.

1.2.1 Architecture Séquentielle

Cette architecture désigne les machines qui traitent une seule instruction à la fois. Ces machines se composent d'un unique processeur doté d'un seul cœur (Unité de calcul). Ce processeur géré l'ordinateur et exécute séquentiellement les tâches (processus) désignées par l'ordonnanceur du système d'exploitation. Cette architecture était la plus utilisée jusqu'à l'avènement de l'architecture parallèle.

1.2.2 Architecture Parallèle

L'architecture parallèle désigne les machines dotées d'un processeur à plusieurs cœurs (multi-cœurs) et/ou plusieurs processeurs, traitant plusieurs instructions en même temps (en parallèle). Elle est aussi l'architecture la plus utilisée des GPU. Il existe plusieurs types d'architectures parallèles. Elle sont différentie selon de variante, à savoir : la source du parallélisme ou le type de mémoire.

Les sources du parallélisme On distingue deux sources de parallélisme :

- le parallélisme de données : consiste en l'exécution de la même opération par chaque unité de calcul sur des données différentes (Figure 1.1).

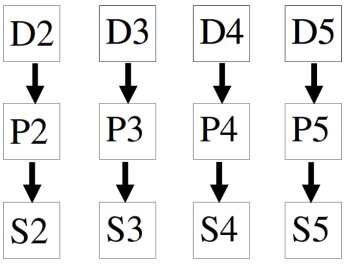

Figure 1.1 – Illustration du parallélisme de données

- Parallélisme de contrôle : consiste en l'exécution des opérations indépendantes simultanément sur plusieurs processeurs (Figure 1.2).

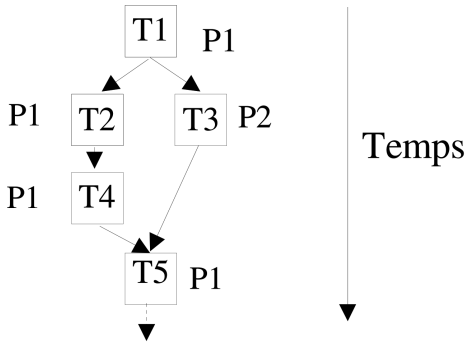

Figure 1.2 – Illustration du parallélisme de contrôles

Les types des mémoires On distingue deux types de mémoires utilisées dans Les architectures parallèles, à savoir :

- Architecture parallèle à mémoire partagée : Cette architecture parallèle dispose d'une mémoire globale constituée d'un ou plusieurs blocs. Ces blocs de mémoire sont partagés entre les différents processeurs. L'accès à ces blocs mémoire est équitable entre les processeurs et chaque bloc mémoire est vu comme étant un seul espace d'adressage par tous les processeurs. Les blocs mémoires peuvent être centralisés ou distribués sur plusieurs processeurs.

- Architecture parallèle à mémoire distribuée : Dans cette architecture, chaque unité est un calculateur complet intégrant à sa composition un processeur, une mémoire et un système d'entrée/sortie.

  Un processeur ne peut avoir accès qu'à sa propre mémoire privée. Les communications entre les processeurs se font à partir d'opérations d'entrée/sortie dont résulte des messages circulant dans des bus à travers le réseau.

1.2.3 Architecture Distribuée

L'architecture distribuée est un ensemble de machines (ordinateurs) dont les ressources ne se trouvent pas au même endroit ou sur la même machine. Elle se repose sur la possibilité d'utiliser des opérations qui s'exécutent sur des machines réparties sur le réseau et communiquent par messages au travers du réseau.
Cette architecture est utilisée dans différents domaines tels que : calcul distribué, pair à pair (Peer-to-Peer), architecture trois-tiers et client-serveur, ex (Figure 1.3). Internet est un exemple de system distribuée client-serveur.

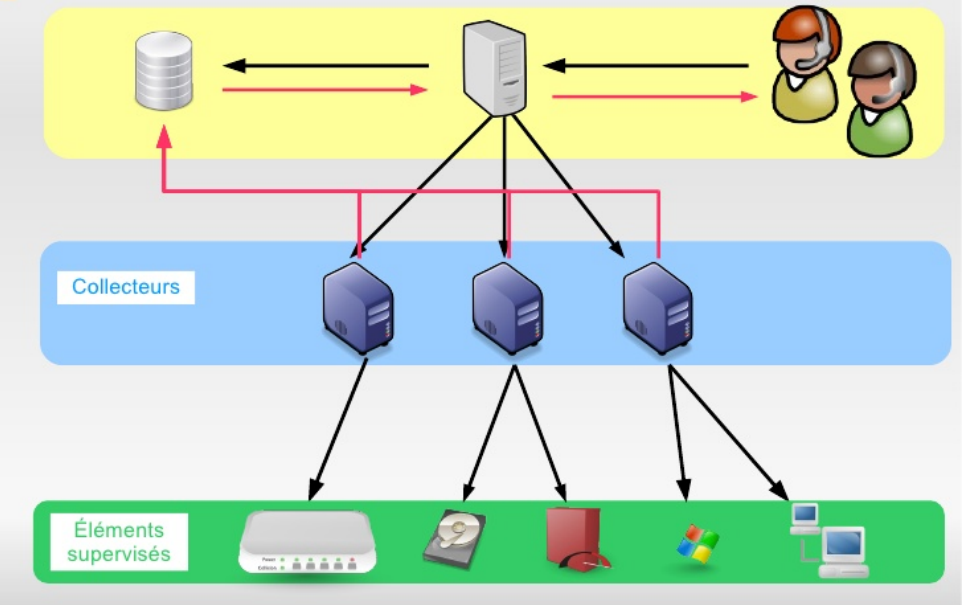

Figure 1.3 – Exemple d'architecture distribuée

1.3 Taxonomie de Flynn

Proposée par Michael Flynn en 1966 [1], la taxonomie de Flynn est une classification des architectures des ordinateurs.
Flynn classe les architectures d'ordinateur en quatre catégories selon le type d'organisation du flux des données et du flux d'instructions ; le flux peut être soit "Simple" ou "Multiple". Ces quatre catégories sont définies comme suit :

1.3.1 Simple Instruction Simple Data (SISD)

Il s'agit d'un ordinateur séquentiel qui n'exploite aucun parallélisme avec un seul flux d'instructions et un seul flux de données (Figure 1.4). Cette catégorie correspond à l'architecture de *Von Neumann*.

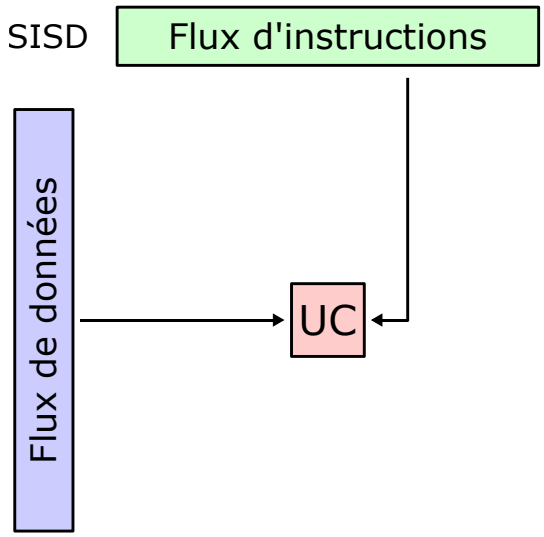

Figure 1.4 – Illustration de "Simple Instruction Simple Data"

1.3.2 Simple Instruction Multiple Data (SIMD)

Il s'agit d'ordinateurs utilisant le parallélisme au niveau de la mémoire avec un seul flux d'instructions pour un flux multiple de données (Figure1.5), par exemple le processeur vectoriel. Cette catégorie permet d'utiliser un ensemble de données (vecteur, matrice,...) qui seront traitées selon la même instruction en parallèle.

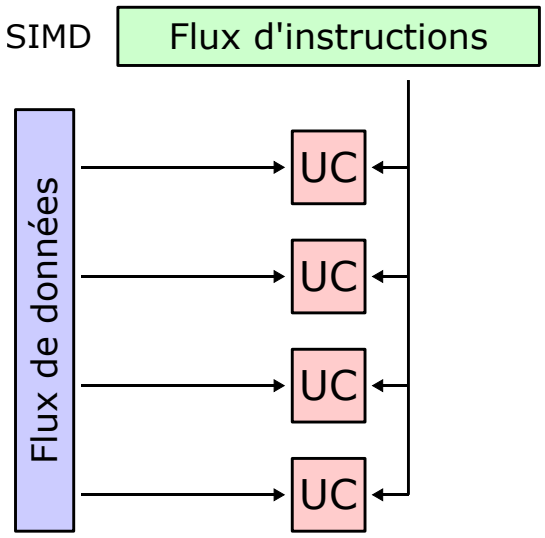

Figure 1.5 – Illustration de "Simple Instruction Multiple Data"

1.3.3 Multiple Instructions Simple Data (MISD)

Cette catégorie désigne un mode de fonctionnement des ordinateurs dotés de plusieurs unités arithmétiques et logiques fonctionnant en parallèle.
Dans ce mode de fonctionnement, plusieurs unités de calcul (UC) traitent la même donnée

en parallèle (Figure1.6). Il existe peu d'implémentations en pratique. Cette catégorie peut être utilisée dans le filtrage numérique et la vérification de redondance dans les systèmes critiques.

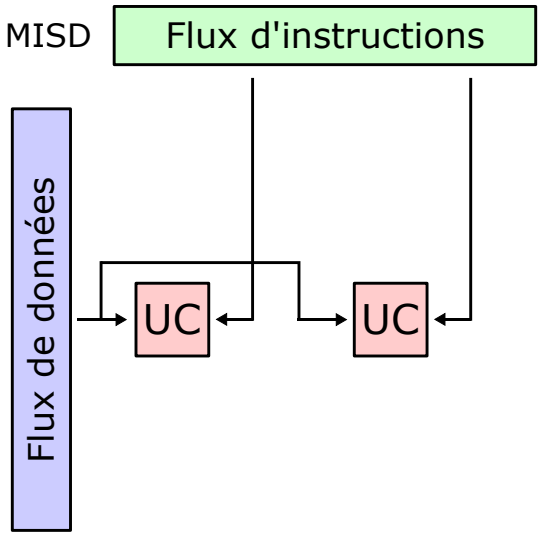

Figure 1.6 – Illustration de Multiple Instruction Simple Data

1.3.4 Multiple Instructions Multiple Data (MIMD)

Dans cette catégorie, plusieurs unités de calcul (multiprocesseurs), ayant chacune une mémoire distincte, traitent indépendamment des données différentes selon des instructions différentes (Figure1.7). Il s'agit de l'architecture parallèle la plus utilisée dont les deux principales variantes rencontrées, sont les suivantes :

MIMD à mémoire partagée Les UC (unités de calcul) ont accès à une mémoire commune (partagée) comme espace d'adressage global. Tout changement dans une case mémoire est perçu par les autres unités de calcul. Cette mémoire commune sert aussi à la communication entre les unités de calcul.
La synchronisation des UC peut se faire au moyen de :

- semaphores
- Verrous, ou Mutex (exclusion mutuelle)
- barrières de synchronisation

MIMD à mémoire distribuée Chaque unité de calcul possède sa propre mémoire et son propre système d'exploitation. Ce second cas de figure nécessite un middleware pour la synchronisation et la communication.

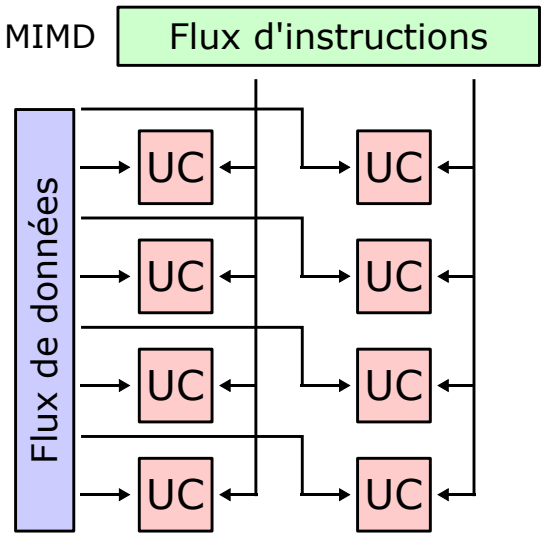

Figure 1.7 – Illustration de Multiple Instruction Multiple Data

1.4 Conclusion

Dans ce chapitre, nous avons présenté les architectures d'ordinateurs selon *Flynn* qui classe les ordinateurs en 4 catégories par l'organisation des flux d'instruction et de données dans les ordinateurs, où le flux peut être « *simple* »ou « *multiple* ». Nous avons aussi présenté les architectures par le mode de traitement des ordinateurs. Nous disposons de 3 types d'architectures majeures, à savoir : architectures séquentielles, parallèles et distribuées.

Chapitre 2 | CUDA

2.1 Introduction

En raison des difficultés apparues lors de l'utilisation des processeurs graphiques (GPU), pour paralléliser les calculs intensifs (scientifiques) avec les différences d'architectures physiques et logiques des GPU, la société *Nvidia*® [7] a unifié ses architectures et a créé CUDA « Compute Unified Device Architecture », en français « Calcul sur périphériques à architectures unifiées »pour faciliter l'utilisation et la programmation de ses *GPU*. Dans ce chapitre, nous présentons l'évolution des GPU ainsi que la naissance de CUDA et sont utilisations pour aider à mieux comprendre CUDA.

2.2 Les GPU

Un GPU (**G**raphic **P**rocess **U**nites) (unités de traitement graphique) est un circuit intégré qui assure les fonctions de calcul de l'affichage (graphique). Le *GPU* (processeur graphique) est souvent intégré sur une carte graphique.

Un cœur est un composant d'un processeur (dans un processeur multi-cœurs) capable d'exécuter des programmes de façon autonome. Toutes les fonctionnalités nécessaires à l'exécution d'un programme sont présentes dans un cœur : compteur ordinal, registres, unités de calcul, etc.

Un processeur graphique (*GPU*), contrairement au processeur centrale (*CPU*), (Figure 2.1), a généralement une structure hautement parallèle. Il comporte des centaines voire des milliers d'unités de traitement (cœurs), ce qui le rend efficace pour une large palette de tâches graphiques comme le rendu 3D, la gestion de la mémoire vidéo, le traitement du signal vidéo, etc.

À l'arrivée des *GPGPU* (**G**eneral-**P**urpose computing on **G**raphics **P**rocessing **U**nits, en français : calcul générique sur un processeur graphique), qui permettent de faire des calculs génériques sur un *GPU*, les *GPU* peuvent être utilisés aussi pour les calculs de tous types (pas uniquement graphiques).
Avec le parallélisme massif des *GPU*, la technologie *GPGPU* rend les calculs généraux plus rapides.

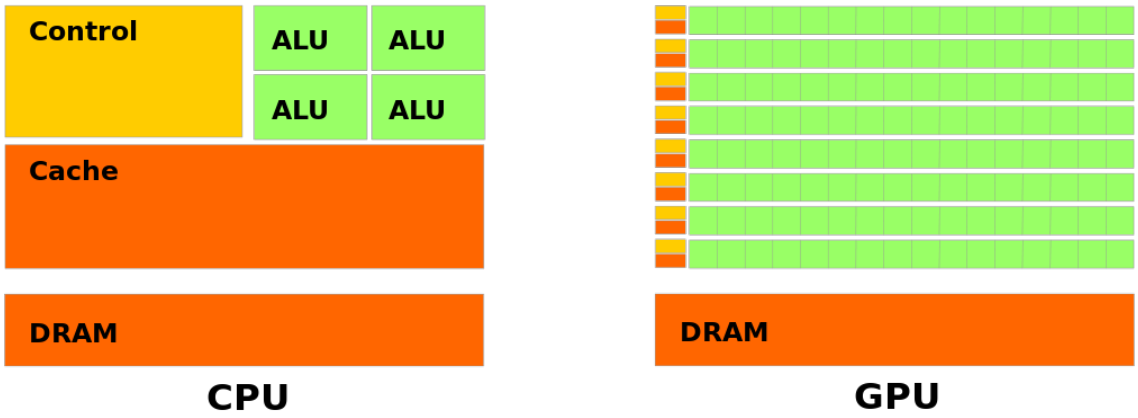

FIGURE 2.1 – Comparaison entre CPU et GPU.

Dans ce qui suit, nous nommerons par *GPU* les processeurs graphiques de Nvidia.

2.3 CUDA

CUDA (Compute Unified Device Architecture) est une technologie de GPGPU de Nvidia® qui permet d'utiliser un processeur graphique (GPU) pour exécuter des calculs généraux à la place du processeur (CPU).

CUDA permet de programmer des GPU avec les langages C/C++. Développée par Nvidia® pour ses cartes graphiques GeForce 8 Séries. *CUDA* utilise un pilote unifié utilisant une technique de streaming (flux continu). Le premier kit de développement pour CUDA a été publié le 15 février 2007.

2.4 L'architecture CUDA

Même si les *GPU* se composent de plusieurs composants, *CUDA*, avec son architecture unifiée, organise les exécutions et de la mémoire et permet de gérer ces *GPU* avec simplicité et de la même manière pour tous modèles de *GPU*. *CUDA* organise les composants du *GPU* utilisables comme suit :

2.4.1 Les unités de calcul (cœurs)

Compte tenu de l'enorme quantité d'unités de calcul sur les *GPU*, *CUDA* a organisé l'exécution de ces unités comme suit :

- **Thread**
  Un thread est l'exécution des instructions (d'un programme ou d'une fonction) sur un cœur (unité de traitement) d'un *GPU*. Il dispose d'un espace mémoire privé appelé « mémoire locale », utilisé pour les appels de fonction, la liste des registres alloués et autre.

- **Thread bloc (bloc de threads)**
  Un "*bloc* de *threads*" est un ensemble d'exécutions simultanées de *threads* qui peuvent coopérer entre eux par la barrière de synchronisation et la mémoire partagée.
  La taille d'un *bloc* de *threads* est définie par le programmeur. Un *bloc* de *threads* est identifié par un numéro unique "*id*" dans la grille à laquelle il appartient.

- **Grid (grille)**
  Une *grille* est un ensemble de *blocs* de *threads* qui exécutent la même fonction (*kernel*), lisent les entrées de la «*mémoire globale*», écrivent les résultats à la «*mémoire globale*» et synchronisent les appels du *kernel* dépendants.

- **Warp**
  Un *warp* est une chaîne de 32 threads reliés au même flux d'instructions. Les *threads* d'un *warp* exécutent la même instruction au même moment.

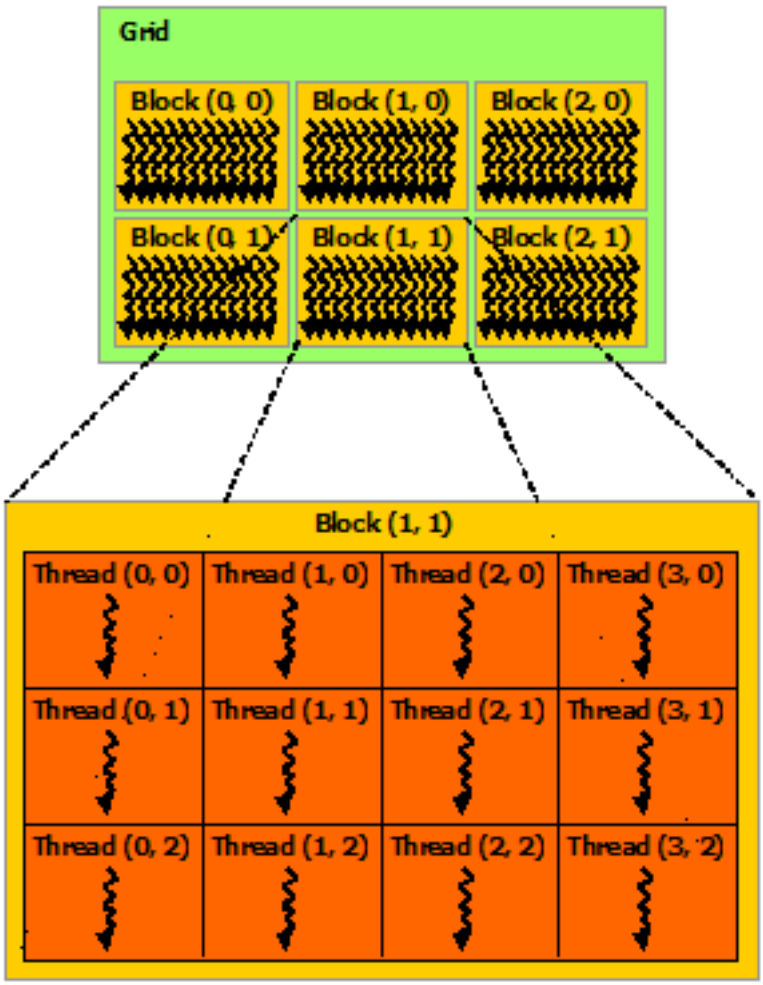

FIGURE 2.2 – L'organisation des *threads* sur *CUDA*

2.4.2 Mémoires

CUDA répartie les mémoires disponibles d'un *GPU* sur plusieurs mémoires, certaines de ces mémoires sont optimisées pour des données précises. Les mémoires utilisables avec CUDA sont :

- **La mémoire globale** La mémoire globale est la plus grande mémoire d'un *GPU*, où les fonctions, leurs paramètres et d'autres données sont stockés. Cette mémoire est accessible par toutes les unités de calcul (*threads*). Cette mémoire, comparée aux autres, est la plus lente en terme de temps d'accés.

- **Les mémoires : constante et texture** Ces deux mémoires sont utilisées en lecture seule (read-only), de façon qu'elles ne peuvent pas être modifiées par les *threads* du *GPU*. Ces mémoires sont relativement limitées en capacité.

- **La mémoire shared (partagée)** C'est une mémoire partagée entre les *threads* d'un même bloc, sa capacité est entre 48Ko et 112Ko. Cette mémoire est parmi les

plus rapides en terme de temps d'accès.

- **La mémoire locale** C'est une mémoire propre à chaque *thread*. Sa capacité est entre 16Ko et 32Ko. Cette mémoire est également parmi les plus rapides en terme de temps d'accès.

- **Les Registres** Un registre est une mémoire de taille de 32 bits, allouable par les *threads* du même *bloc*. Le nombre maximum de registres par *thread* est de 32 mille à 128 mille registres selon la capabilité du *GPU*. Les registres sont les plus rapides en temps d'accès.

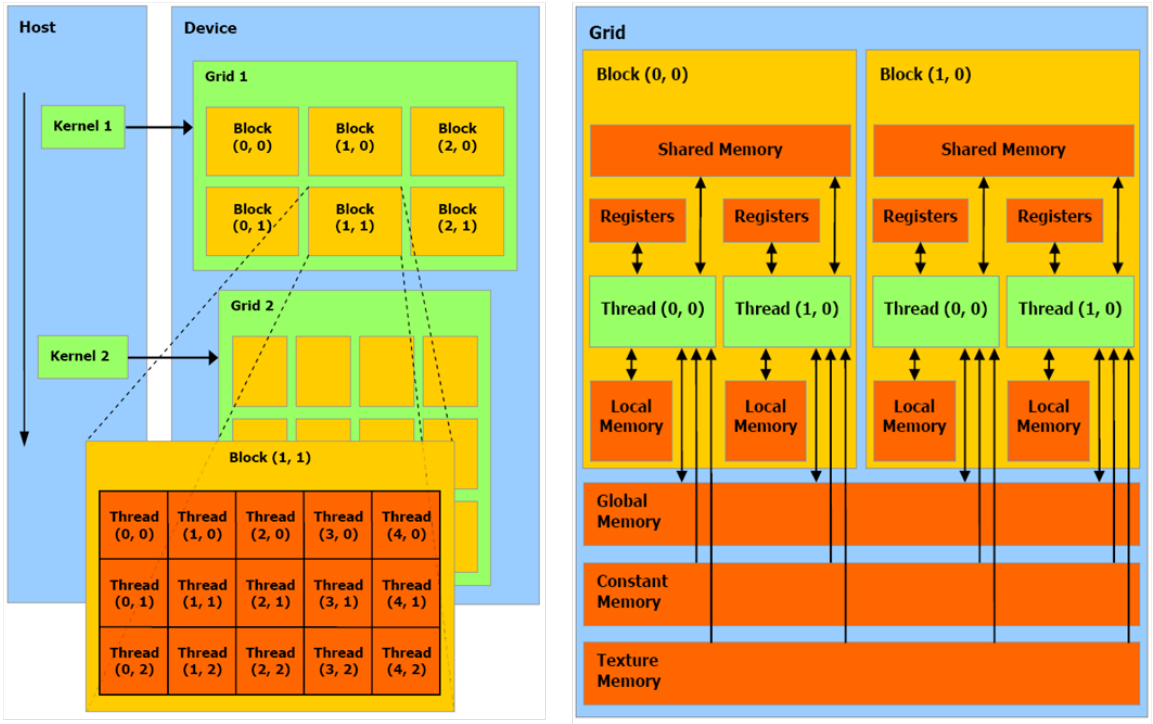

FIGURE 2.3 – Hiérarchie logique *CUDA*

2.5 Le principe de fonctionnement de CUDA

La plate-forme CUDA est conçue pour fonctionner avec des langages de programmation tels que C, C ++ et Fortran. Ceci rend plus facile, pour les intéressés de la programmation parallèle, l'accès aux ressources GPU, contrairement aux API antérieures.

Un programme *CUDA* démarre son exécution dans le *CPU* et utilise la mémoire centrale *RAM*, puis, arrivé à la partie parallèle, le programme passe par plusieurs étapes (Figure 2.4).

Les étapes utilisées par *CUDA* pour exécuter la partie parallèle sur le *GPU* sont comme suit :

2.5.1 Étape 01 : Copie vers GPU

Après avoir défini et préparé les données à charger dans la mémoire GPU, CUDA copie ces données, ainsi que les paramètres de la fonction à exécuter *kernel*, de la mémoire centrale (*RAM*), vers la mémoire globale du *GPU*. Si les textures et les constantes sont définies, elles sont alors chargées de la mémoire centrale (*RAM*) vers leurs mémoires respectives sur le *GPU*.

2.5.2 Étape 02 : chargement des instructions

Instruire les unités de calcul du *GPU* des instructions de la fonction *kernel* à exécuter qui sont préalablement chargées dans la mémoire *GPU*. Par ailleurs, le nombre de *threads* et de *blocs* à utiliser (choisis par le programmeur) sont communiqués au *GPU*.

2.5.3 Étape 03 : Exécution sur *GPU*

Après avoir reçu les instructions et les données, le *GPU* répartit les *threads* en *blocs* et définit les *warps* puis exécute en parallèle les instructions dans chaque cœur du *GPU*.

2.5.4 Étape 04 : Récupération des résultats

Au terme de l'exécution de tous les *threads*, on procède à la copie des résultats du *GPU* vers la mémoire centrale *RAM*. Le *CPU* prend la relève et termine l'exécution du programme.

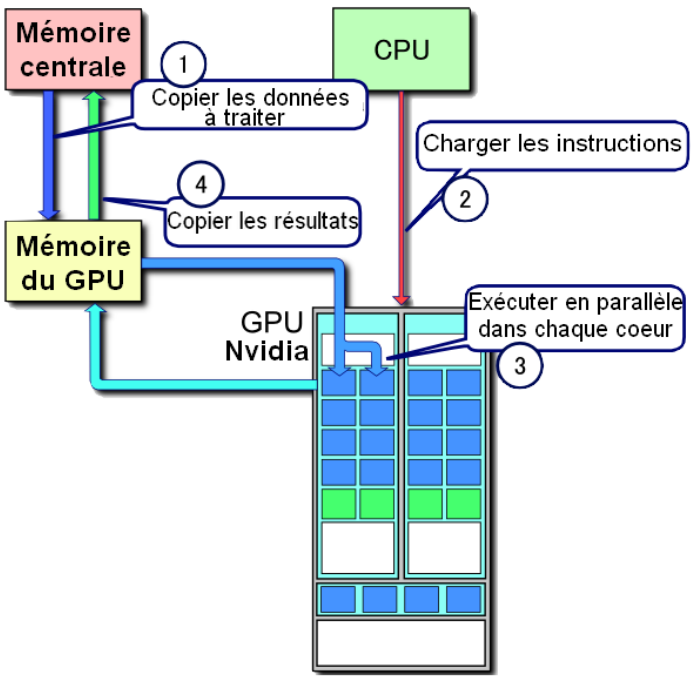

FIGURE 2.4 – Schéma du principe de l'utilisation de CUDA

2.6 Conclusion

Dans ce chapitre, nous avons décrit les GPU, l'impact de la technologie $GPGPU$ sur GPU, et la naissance de $CUDA$ du côté de la société *Nvidia* qui utilise cette technologie pour permettre de faire des calculs de tous types sur les processeurs graphiques (GPU) de Nvidia. Nous avons présenté également dans ce chapitre l'architecture globale de $CUDA$ et le principe d'utilisation de $CUDA$.

Chapitre 3 | Algorithmes de jointure

3.1 Introduction

En informatique, et plus particulièrement en Base de données relationnelle, la jointure est l'opération binaire, qui permet de fusionner deux tables (relations), de sorte à ne laisser que les tuples ayant leurs correspondants dans les deux tables. On symbolise la jointure entre deux relations R et S par $R \bowtie S$.
Plus formellement, la sémantique de la jointure naturelle est définie comme suit :

$$R \bowtie S = \{\ t \cup s \mid t \in R \wedge\ s \in S \wedge\ \{\forall a \in t \cap r\ :\ val(t,a) = val(s,a)\}\}.$$

La figure 3.1 illustre un exemple de jointure.
où :

- La table R contient 3 attributs A, B et C,
- La table S contient 3 attributs B, D et E.

Dans cet exemple la jointure naturelle se fait sur l'attribut en commun B et donne la table résultante $R \bowtie S$ de 5 attributs, à savoir : A, B, C, D et E.

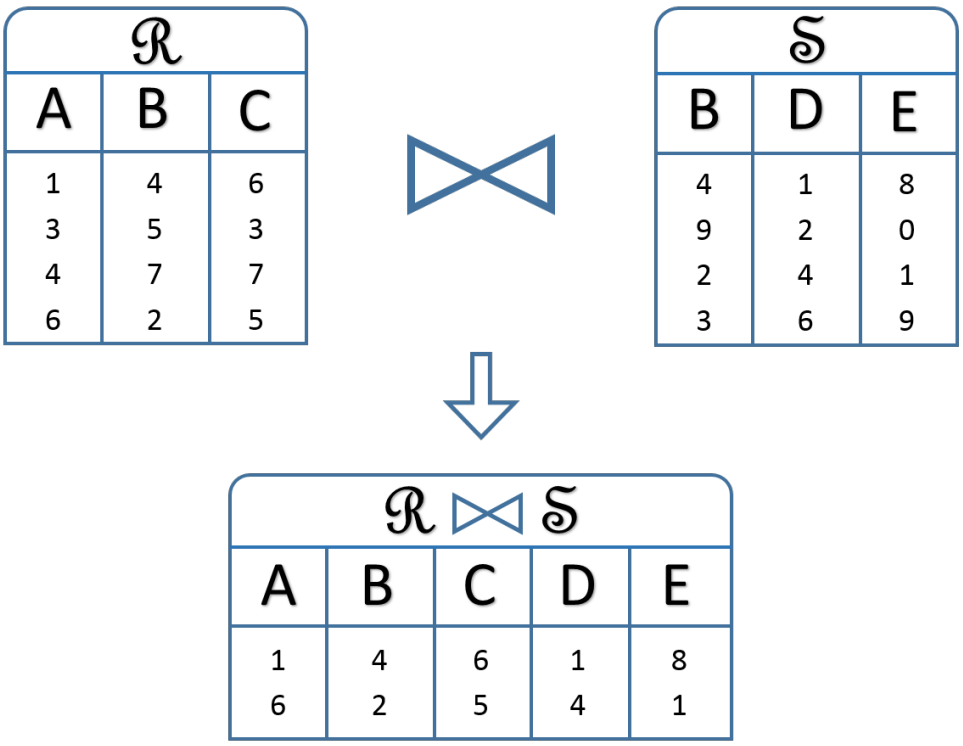

Figure 3.1 – Exemple de jointure

Il existe plusieurs algorithmes de jointures ; les algorithmes principalement utilisés sont :
Supposons d'abord deux relations R et S.

3.2 Algorithme de jointure par boucles imbriquées

3.2.1 Principe

Le principe de cet algorithme consiste à comparer les valeurs des attributs, communs aux deux relations, de chaque tuple de la relation R avec tous les tuples de la relation S, (ce qui nous donne des couples de tuples) puis fusionner les tuples de chaque couple comparés dont les valeurs des attributs en commun sont identiques.

3.2.2 Pseudo Code

L'algorithme 1 suivant, représente la jointure par *boucles imbriquées* :

Algorithm 1 Algorithme de jointure par Boucles imbriquées.

```
1: Relation joinBI(Relation R, Relation S)
2: Var :
3:    R_esult : Relation ;                    // La relation résultante de la jointure.
4: Begin
5:    foreach tuple r in R do
6:       foreach tuple s in S do
7:          if r = s then
8:             add(R_esult, fusion(r, s)) ; // Ajouter à R_esult le tuple résultant de la fusion
               des tuples r et s.
9:          end if
10:      end for
11:   end for
12:   return  R_esult ;
13: End.
```

3.2.3 Avantages et inconvénients

Cet algorithme comporte en soi des avantages précieux. Et, comme il n'éxiste pas d'algorithme parfait, il comporte également des inconvénients.
Quelques avantages liés à l'utilisation de cet algorithme sont :

- Facile à implémenter,
- Ne consomme pas davantage de mémoire comparé à quelques autres algorithmes,
- N'utilise pas d'autre traitement comme trie ou le hachage.
- Facile à paralléliser dans plusieurs langages.

- Très efficace si l'une des deux relations entre entièrement en mémoire.[8]

Les inconvénients d'utilisation de cet algorithme sont :

- La jointure par boucles imbriquées est inefficace sur de très grandes tables.[8]
- Beaucoup de comparaisons inutiles. il compare tous les tuples aveuglement, ce qui le rend très lent, ex : dans le cas où : le résultat est uniquement la fusion des premier tuples des deux relations.

3.3 Algorithme de Jointure par Tri-Fusion

3.3.1 Principe

Cet algorithme facilite la jointure, mais utilise un autre traitement qui est le tri.
On commence par trier les deux relations R et S sur les attributs à joindre dans l'ordre croissant (resp. décroissant).
On initialise un pointeur pour chacune des deux relations, puis on les pointe sur les premiers tuples de chaque relation. On avance le pointeur de R si la valeur du tuple pointé est inférieure (resp. supérieure) à celle du pointeur de S, et on fait de même pour le pointeur de S. Dès lors qu'on a des valeurs identiques, on fait la fusion des deux tuples pointés, puis on sauvegarde le tuple résultant.

3.3.2 Pseudo Code

L'algorithme 2 suivant, représente la jointure par *tri-fusion* :

Algorithm 2 Algorithme de jointure par Tri-Fusion.

```
1: Relation joinTF(Relation R, Relation S)
2: Var :
3:    R_esult : Relation ;                    // La relation résultante de la jointure.
4:    p_R : Pointer of tuple ;
5:    p_S : Pointer of tuple ;
6: Begin
7:    trier(R) ;                              // trier la relation R (croissant)
8:    trier(S) ;                              // trier la relation S (croissant)
9:    p_R ← R.first ;                         // p_R Pointe début de R
10:   p_S ← S.first ;                         // p_S Pointe début de S
11:   while p_S ≠ NULL and p_S ≠ NULL do
12:      if p_R.val = p_S.val then
13:         add(R_esult, fusion(p_R, p_S)) ;
14:         p_R ← p_R.next ;
15:      end if
16:      if p_R.val < p_S.val then
17:         p_R ← p_R.next ;
18:      end if
19:      if p_S.val < p_R.val then
20:         p_S ← p_S.next ;
21:      end if
22:   end while
23:   return R_esult ;
24: End.
```

3.3.3 Avantages et inconvénients

Cet algorithme avec son option de tri apporte des avantages à la jointure dans certains cas et engendre des inconvénients dans d'autres.
Les avantages liés à l'utilisation de cet algorithme sont :

- D'accélérer la jointure dans le cas ou les tables (relations) jointes ne sont pas très grandes,
- D'éviter les comparaisons inutiles avec la fonctionnalité du tri,

- De faire un seul parcours des deux tables, sauf dans le cas des doublons.

Les inconvénients pouvant être rencontrés lors de l'utilisation de cette algorithme sont :

- Lenteur lors des tris des tables (relations),
- Plus de lenteur encore lors d'un retour arrière, dans le cas de doublons (valeurs identiques d'un ou plusieurs tuples de la relation R avec plusieurs tuples de la relation S).

3.4 Algorithme de Jointure par Hachage

3.4.1 Principe

Cet algorithme est plus sophistiqué, il se compose de deux parties.
Premièrement, on crée une table de hachage H_S de la relation S (la plus petite de préférence) avec un tableau associatif comportant des paires (clé, val), où : *clé* est le haché des valeurs des attributs en commun entre les deux relations du tuple. *val* est une liste des numéros de lignes des tuples qui ont les mêmes valeurs donc la même *clé* (haché).
Deuxièmement, on parcourt, par boucles imbriquées, chaque tuple de la relation R. On hach les valeurs des attributs en commun entre les deux relations du tuple, puis on vérifie avec les *clés* de H_S l'existence de ce haché dans la table de hachage H_S. Si le haché existe, on ajoute, à la table résultat, les tuples résultant de la fusion du tuple de la relation R avec chaque tuple dont le numéro de ligne est référencé dans la liste *val* de H_S correspondante au haché.

3.4.2 Pseudo Code

L'algorithme 3 suivant, représente la jointure par *hachage* :

Algorithm 3 Algorithme de jointure par Hachage.

```
1: Relation joinH(Relation R, Relation S)
2: Var :
3:    R_esult : Relation ;                    // La relation résultante de la jointure.
4:    H_S : Map of couples (Integer, List of integer) ;   // La table associative de
      hachage de S.
5:    h : Value of hash ;
6: Begin
7:    // Part 1
8:    foreach tuple t in S do
9:       H_S[hash(t)].push_back(t) ;
10:   end for
11:   // Part 2
12:   R_esult = newRelation ;        // initialise la relation R_esult qui sera le résultat de la
      jointure.
13:   foreach tuple t in R do
14:      h ← hash(t) ;
15:      if exist H_S[h] then
16:         foreach tuple s in H_S[h] do
17:            add(R_esult, fusion(t, s)) ;
18:         end for
19:      end if
20:   end for
21:   return R_esult ;
22: End.
```

3.4.3 Avantages et inconvénients

Grâce au hachage qu'il utilise, cet algorithme apporte des avantages à la jointure et entraine des inconvénients reliés à cette technique utilisée.
les avantages liés à l'utilisation de cette algorithme sont :

- Diminue considérablement le temps de comparaisons, plus spécialement dans le cas de jointure des petites tables (relations).
- Diminue le nombre de comparaisons effectuées lors de la jointure.

Les inconvénients découlant de l'utilisation de cet algorithme sont :

- Consomme beaucoup de mémoire, pour la réalisation de la table de hachage.
- Prend beaucoup de temps pour le hachage des tables volumineuses (grand nombre d'attributs ou de tuples).

3.5 Conclusion

Dans ce chapitre, nous avons présenté trois principaux algorithmes de jointure ainsi que leurs avantages et leurs inconvénients. Nous avons vu que l'algorithme par boucles imbriquées présentait l'intérêt de simplicité mais présente quand même l'inconvénient de prendre beaucoup de temps avec des très grandes tables à cause des comparaisons inutiles faites lors de la jointure. L'algorithme par tri-fusion est plutôt plus adapté pour des petites tables.
Enfin l'algorithme de jointure par hachage qui présente l'avantage de rapidité suite au hachage qui réduit le nombre de comparaisons dans le processus de jointure, mais qui prend beaucoup de temps pour le hachage des tables volumineuses.

Chapitre 4 | Les travaux déjà faits

4.1 Introduction

De nos jours, le nombre de personnes qui s'intéressent au développement technologique et à la science augmente. Dans le domaine de la jointure aussi où on trouve chaque année plusieurs projets de recherche sur ce sujet.
Nous allons présenter dans ce chapitre quelques travaux effectués dans le domaine des jointures de base de données.

4.2 Jointure sur CPU multi-coeurs [4]

À l'université of Wisconsin-Madison, 3 personnes ont implémenté et évalué un algorithme de jointure parallèle basé sur le hachage (Hash Join) en utilisant des processeurs multi-coeurs.

4.2.1 Implémentation de l'algorithme

L'implémentation de cet algorithme est répartie en 3 phases, à savoir, la phase partition, la phase build (construction) et la phase probe (sondage) :

La phase Partition La phase de partition est une étape facultative dans cet algorithme. Elle est utilisée uniquement si la table de hachage ne peut pas être chargée entièrement dans la mémoire principale.
Dans cet implémentation, une partition est une liste chainée des buffers de sortie (hachés). Un buffer de sortie est une structure composée de quatre éléments : un entier spécifiant la taille du bloc de données, un pointeur au début du bloc de données, un pointeur vers l'espace libre dans le bloc de données et un pointeur vers le produit de sortie suivant qui est initialement mis à zéro. En cas d'overflow (dépassement de mémoire), un buffer de sortie vide est ajouté au début de la liste, puis ce dernier est relié au buffer saturé. La localisation de l'espace libre consiste à vérifier le premier buffer de la liste. Pendant la phase de partitionnement, tous les threads commencent à lire les tuples à partir de la relation R, via un curseur.
Chaque thread fonctionne sur un grand lot de tuples à la fois, afin de minimiser les frais généraux de synchronisation sur le curseur de balayage d'entrée.
Chaque thread examine un tuple, puis extrait la *clé* k et calcule la fonction hachage de partition $hp(k)$. Ensuite, il écrit le tuple pour partitionner $Rhp(k)$ en utilisant un algorithme des deux algorithmes « algorithme bloquant »ou « algorithme non bloquant ».
Lorsque le curseur de R est à court de tuples, l'opération de partitionnement progresse pour traiter les tuples de la relation S. Encore une fois, chaque tuple est examiné, la clé de jointure k est extraite et le tuple est écrit dans la partition $Shp(k)$.

La phase de partitionnement se termine lorsque tous les tuples de S sont partitionnés. Notez que les algorithmes de partitionnement sont classés « non bloquants »s'ils produisent des résultats sur le flan et s'ils numérisent l'entrée en une seule fois, sinon ils sont classés « bloquants ». Un algorithme de partitionnement est classé « bloquants »s'il produit des résultats après avoir effectué l'entrée entière et s'il la numérise en plus d'une fois. Les trois concepteurs reconnaissent que l'opérateur de jointure en général n'est jamais vraiment non bloquant, car il bloquera pendant la phase de construction.

La phase Build (construction) Soit n le nombre de threads, La phase de construction se déroule comme suit :
Si la phase de partition a été omise, tous les threads sont affectés au travail sur la relation R. Si le partitionnement a été effectué, chaque thread i est assigné à travailler sur les partitions $Ri + 0 * n$, $Ri + 1 * n$, $Ri + 2 * n$, etc. Par exemple, une machine à quatre coeurs a $n = 4$, le thread 0 fonctionnera sur les partitions R0, R4, R8, ..., le thread 1 sur R1, R5, R9, ..., etc.

Ensuite, une table de hachage vide est construite pour chaque partition de la relation d'entrée R. Pour réduire le nombre de pertes de cache qui sont engagées pendant la phase suivante (sonde), chaque zone de cette table de hachage est dimensionné de sorte qu'il fonctionne sur quelques lignes de cache. Chaque thread analyse chaque tuple t dans sa partition, extrait la clé de jointure k, puis supprime cette clé à l'aide d'une fonction de hachage h. Ensuite, le tuple t est ajouté à la fin du sceau de hachage h(k), en créant une nouvelle zone de hash si nécessaire. Si la phase de partition a été omise, tous les threads partagent la table de hachage, et les écritures à chaque zone de hachage doivent être protégées par un loquet. La phase de construction est terminée lorsque tous les n threads ont traité toutes les partitions assignées.

La phase Probe (sondage) La phase de la sonde, programme, pendant la phase de construction, le travail sur les n threads d'une manière similaire, à savoir, si aucun partitionnement n'a été effectué, tous les threads seront affectés à S et ils se synchronisent avant d'accéder au curseur de lecture pour S. Sinon, le thread i est affecté aux partitions $Si + 0 * n$, $Si + 1 * n$, $Si + 2 * n$, etc. Pendant cette phase, chaque thread lit chaque tuple s de la partition qu'on lui a assigné et extrait la clé k. Il vérifie alors si la clé de chaque tuple r stocké dans le seau $h(k)$ correspond à k. Cette vérification est nécessaire pour filtrer les éventuelles collisions de hachage. Si les clés correspondent, les tuples r et s sont fusionné pour former le tuple de sortie. ce tuple est écrit dans un fichier de sortie qui est privé au thread.

4.3 Jointure distribuée utilisant map/reduce [5]

4.3.1 Le paradigme Map/reduce

Map/Reduce est un « modèle de programmation et une implémentation associée pour le traitement et la génération de grands ensembles de données ». Il a d'abord été développé chez Google par Jeffrey Dean et Sanjay Ghemawat. Leur motivation a été suscitée du fait de la multitude de calculs, effectués tous les jours dans Google, qui impliquaient d'énormes quantités de données d'entrées. Ces calculs se sont généralement révélé conceptuellement simples.
Et lorsque les systèmes distribués sont apparus, un certain nombre de problèmes, comme la distribution des données, le partage ou la parallélisation du calcul, rendent difficile le développement du calculateur réel.
Dean et *Ghemawat* ont vu le besoin d'une abstraction qui aiderait le programmeur à se concentrer sur le calcul en cours sans avoir à se préoccuper des complications d'un système distribué comme la tolérance aux pannes, l'équilibrage de charge, la distribution de données et la parallélisation des tâches. Et c'est exactement ce pour quoi *Map/Reduce* a été conçu. Un cadre simple mais puissant qui permet au programmeur d'écrire des unités de travail simples en tant que carte et à réduire les fonctions. Le cadre s'occupe alors automatiquement du partitionnement et de la parallélisation de la tâche sur un grand groupe de machines commerciales peu coûteuses. Il prend en charge tous les problèmes mentionnés précédemment, à savoir la tolérance aux pannes, l'équilibrage de charge, la distribution de données et la parallélisation des tâches.
L'idée a été rapidement reprise par la communauté open source, la Fondation Apache Software, spécialement, et s'est développée en un projet open source et ensuite dans un cadre et une implémentation intégrale appelée « Hadoop ». Hadoop est libre à télécharger et se vante maintenant d'une très grande communauté de programmeurs et d'entreprises qui comprend de grandes entreprises Web 2.0 comme *Yahoo*.

4.3.2 Algorithmes utilisés

Ce projet traite les jointure avec un algorithme adapté au paradigme *Map/Reduce*, il est réparti en trois phase, à savoir : map, process, reduce.

1. **Map :** Dans cette phase, les relations sont découpé et mappée (cartographiée) en ajoutant des identifiants des relations au tuples.

2. **Process :** Dans cette phase, les tuples sont réparti de manière aléatoire sur les nouds (ordinateurs), puis la comparaison ce fait sur ces nouds en même temps, puis combiner les résultats.

3. **Reduce :** les résultats des nouds de la phase *process* sont regroupé sur un seul noud (père) qui fusion les résultats et les regroupe dans une même table (relation).

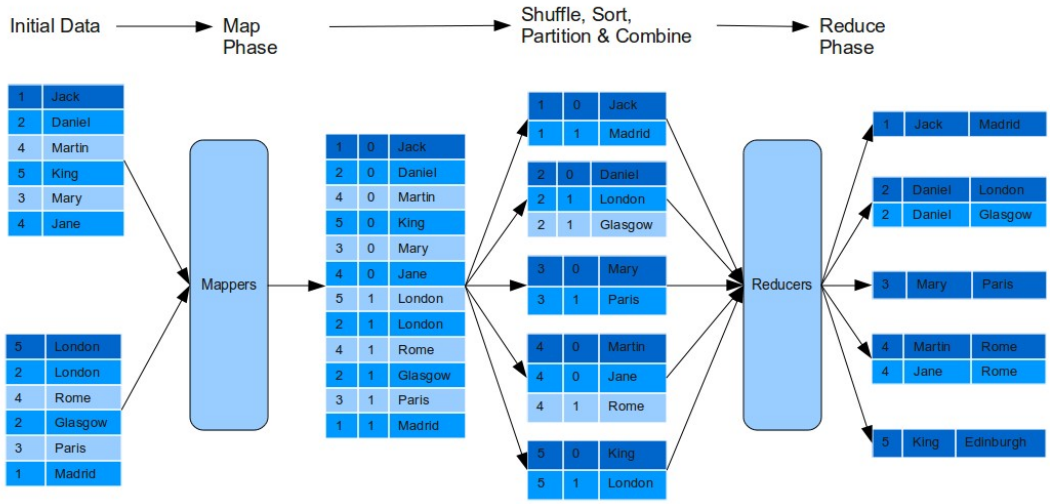

Figure 4.1 – Jointure par Map Reduce

4.4 Conclusion

Dans ce chapitre nous avons présenté quelques travaux effectués dans le domaine des jointures de base de données, à savoir :
le projet « Jointure sur CPU multi-cour »[4] de Spyros Blanas, Yinan Li et Jignesh M. Patel de l'university de Wisconsin-Madison qui traitent les jointures avec les CPU multi-coeurs et utilisent l'algorithme « hash-join ».
Ainsi que le projet « Jointure distribuée utilisant map/reduce »[5] de Jairam Chandar de l'université de Edinburgh qui a traité les algorithmes avec map/reduce une technologie pour les systèmes distribués.

Deuxième partie

Conception et Réalisation

Chapitre 5 | Conception

5.1 Introduction

Dans les chapitres précédents, nous avons présenté les différents algorithmes les plus utilisés pour le calcul de jointure des tables dans les bases de données relationnelles. Nous avons vu que cette opération est couteuse en termes de temps de calcul et d'espace.
Nous avons évoqué également quelques travaux traitant ce problème de différentes approches : parallèle et distribuée.

Dans cette partie du mémoire nous présentons les détails de notre travail qui consiste à proposer un nouvel algorithme de jointure , puis l'implémentation et l'expérimentation de cet algorithme dans 3 modes, à savoir :
Le mode *CPU* qui s'exécute en séquentiel sur le *CPU*, le mode *GPU* qui s'exécute en parallèle sur le *GPU*, et enfin le mode *Hybrid* qui s'exécute en parallèle et à la fois sur le *CPU* et le *GPU*.
Nous utilisons pour la parallélisation de cet algorithme la technologie *CUDA* sur les *GPU* de *Nvidia*.

5.2 Structure de données

Nous utilisons dans notre projet, un benchmark de *Renault* comme base de données pour l'expérimentation de l'algorithme. Ce benchmark est utilisé dans les *CSP*, il est en langage "XML" au format "XCSP 2.1", il comporte uniquement des entiers comme données (Figure 5.1).

```
<?xml version="1.0" encoding="UTF-8"?>
<instance>
<presentation maxConstraintArity="10" format="XCSP 2.1"/>
<domains nbDomains="15">
 <domain name="D0" nbValues="9">0..8</domain>
    ⋮
 <domain name="D14" nbValues="7">0..6</domain>
</domains>
<variables nbVariables="111">
 <variable name="V0" domain="D0"/>
    ⋮
 <variable name="V110" domain="D3"/>
</variables>
<relations nbRelations="142">
<relation name="R0" arity="2" nbTuples="1" semantics="conflicts">
0 0
</relation>
    ⋮
<relation name="R141" arity="4" nbTuples="3" semantics="conflicts">
0 1 1 5|1 0 0 9|1 1 0 3
</relation>
</relations>
<constraints nbConstraints="159">
 <constraint name="C0" arity="2" scope="V8 V9" reference="R0"/>
    ⋮
 <constraint name="C158" arity="10" scope="V0 V13 V24 V35 V57 V68 V89 V92 V95 V99" reference="R92"/>
</constraints>
</instance>
```

Figure 5.1 – Structure d'un benchmark de Renault (format XML) .

Notre modélisation de la base de données est une structure de données orientée objet, répartie en 4 classes principales, à savoir : *Constraints*, *Relations*, *Domains* et *Variables*.

Ces classes sont définies comme suit :

- La classe *Constraints*
  Cette classe contient la liste des *relations* ainsi que la liste des variables (attributs) de chaque *relation*,

- La classe *Variables*
  Cette classe contient la liste des *variables* (attributs) ainsi que le *domaine* de chaque *variable* (*attribut*),

- La classe *Domains*
  Cette classe contient la liste des *domaines* ainsi que les intervalles de données (domaine numérique) de chaque *variable* (*attribut*). (Ex D1 : 0..50, ie. les *variables* du *domaine* D1 ont des valeurs comprises entre 0 et 50),

- La classe *Relations*
  Cette classe contient l'ensemble des valeurs des *relations* , ainsi que les informations sur leurs dimensions.

Les relations entre ces classes sont comme illustrées dans le diagramme des classes (Diagramme 5.2) suivant :

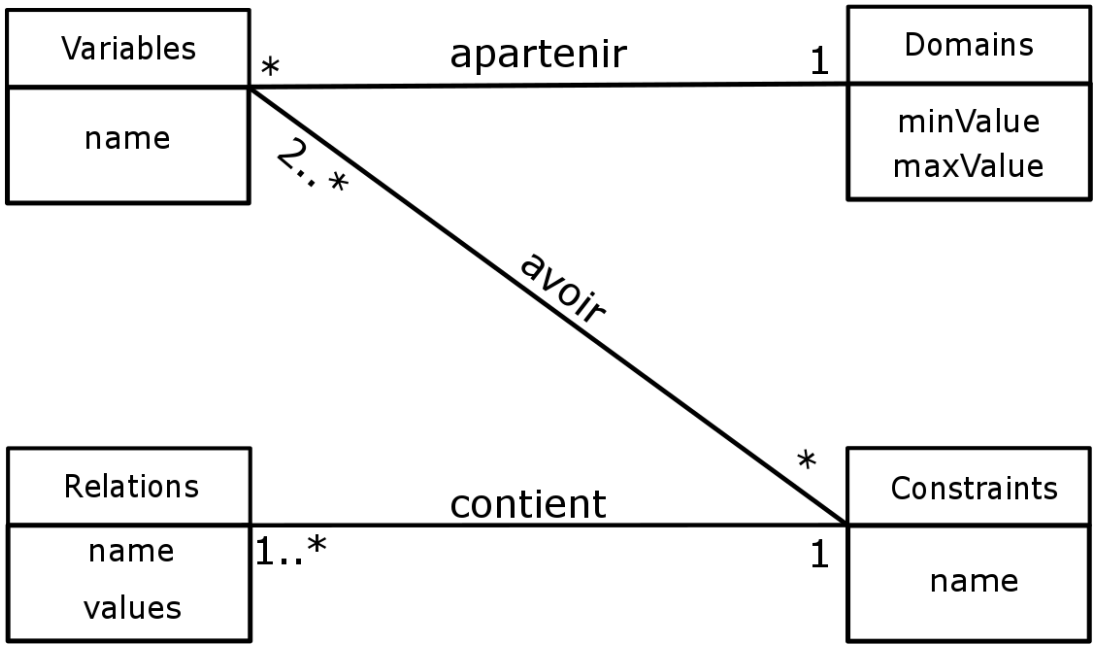

Figure 5.2 – Diagramme de classes

5.3 Méthode de traitement

À la conception du projet, nous avons opté pour l'utilisation du parallélisme comme approche majeure de résolution du problème de jointure. Parmi les divers modes de parallélisation existants, c'est la parallélisation des données (SIMD) qui a attiré notre attention en particulier, et cela pour diverses raisons :

- Elle correspond parfaitement aux exigences du problème de jointure, où une seule et même opération (jointure) se fait sur des données différentes,
- Elle est exploitable en utilisant les processeurs graphiques (GPU) et les processeurs centraux (CPU)à multi-cœurs,
- Son implémentation sur CUDA est relativement facile.

La parallélisation dans cette conception est effectuée au moment de la comparaison des attributs des deux relations (tables) jointes, où plusieurs tuples (lignes) sont comparées au même temps (concurrent), ce qui implicitement augmente la vitesse de traitement des données donc diminue le temps de traitement de la jointure.

5.4 L'algorithme *JoinById* proposé

Afin d'arriver à une solution, nous avons implémenté un nouvel algorithme qui s'adapte au parallélisme. Cet algorithme a la particularité de n'utiliser que ce qui est nécessaire. Les algorithmes cités dans le chapitre 3 utilisent, lors de la jointure, les tuples à comparer dans leur totalité, en incluant les attributs qui ne participent pas lors des comparaisons des tuples. Contrairement à ces algorithmes, l'algorithme *JoinById* proposé calcule d'abord l'ensemble des attributs en commun « *Commons* »avec toutes les relations à joindre, puis il fait le traitement des jointures sur les tables en ne tenant compte que des attributs appartenant à l'ensemble « *Commons* ». Ceci réduit considérablement la quantité de mémoire utilisée surtout dans le cas des relations volumineuses.

5.4.1 Principe de l'algorithme *JoinById*

Le concept de base de cet algorithme est, lors d'une jointure de plusieurs relations, d'extraire uniquement les attributs en commun avec toutes les relations jointes. Ceci est fait en projetant les deux relations avec l'ensemble des attributs en commun de toutes les relations à joindre, puis en comparant les tuples et enfin en terminant la jointure avec la fusion des tuples comparés de valeurs égales.
Cet algorithme est composé de 4 étapes qui sont :

Étape 1 détermination de l'ordre de jointure Dans cette étape, après avoir procédé à la lecture du fichier "XML" (base de données), à l'analyse (parsing) et à l'extraction des données qui seront organisées en objets (Constraints, ...), nous déterminons l'ordre de jointure. Cette étape donne comme résultat une liste des relations ordonnées d'une façon plus ou moins optimale. Cette liste sera prise par le processus de jointure comme l'ordre de jointure à suivre par la suite (dans l'étape 2).

L'ordre de jointure est très important. Cet ordre peut permettre, dans certains cas, de gagner beaucoup en terme de temps et de mémoire.
En partant du principe que, plus il y a des attributs en commun, moins il y a de chance d'avoir des jointures (similitudes) (similarité), cette opération ce résume on 3 étapes qui sont :

- Déterminer la contrainte (relation) qui a le plus d'attributs, on l'appelle « *biggest* »,
- Déterminer le nombre d'attributs en commun entre chaque relation et la relation « *biggest* », puis faire une liste qui contiendra la référence de chaque relation ainsi que le nombre d'attributs en commun qu'a cette relation avec la relation « *biggest* »,
- Trier cette liste par ordre décroissant. L'ordre de jointure est déterminé par l'ordre des relations référencées dans cette liste.

Au cours de cette étape nous déterminons aussi l'ensemble des attributs en commun avec toutes les relations à joindre « *Commons* ».

Étape 2 projection Dans cette étape, nous faisons la projection, sur les deux relations à joindre, des attributs en commun entre ces deux relations. Ceci est réalisé en extrayant les attributs inclus dans l'ensemble des attributs en commun entre ces deux relations ainsi que les valeurs de ces attributs dans chaque relation.

Cette étape est répartie comme suit :

- Détermination des n attributs en commun entre les 2 relations en cours de jointure,
- Création de la relation R' contenant les attributs en commun ainsi que leurs valeurs de la première relation R et de la relation S' contenant les attributs en commun ainsi que leurs valeurs de la seconde relation S.

Étape 3 comparaisons Après avoir projeté les deux relations sur les attributs qui sont en commun entre eux, nous procédons à la comparaison des relations résultantes de la projection.
La comparaison entre les deux relations est comme suit :

- Prendre chaque tuple de la relation R' avec chaque tuple de la relation S',
- Comparer les valeurs des attributs des deux tuples, un à un,
- Ajouter les *identifiants* (ID des tuples, références des tuples ou numéros de lignes) des deux tuples au Tableau IDs et ce dans le cas où la valeur de chaque attribut de la relation R' a la même valeur que celle du même attribut de la relation S'.
- Mettre à jour le tableau *GlobalIDs*, en éliminant les lignes non disponibles dans la première colonne du tableau IDs, ajouter ensuite la deuxième colonne du tableau IDs au tableau *GlobalIDs*. Ceci dans le cas où le tableau *GlobalIDs* est déjà créé, Sinon on crée le tableau qui sera une copie identique du tableau IDs.

Étape 4 génération et fusion Le déroulement de cette étape est conditionné par la file des relations réstantes à joindre.

1. Dans le cas où la file des relations n'est pas vide (il reste au moins une relation à joindre), on effectue une *génération* en utilisant le tableau *GlobalIDs* pour générer la projection des attributs de l'ensemble « *Commons* »sur la relation résultant de la précédente comparaison. Pour cela nous procédons comme suit :

- Parcourir le tableau *GlobalIDs*. Pour chaque ligne de ce tableau, parcourir chaque colonne de cette ligne.

- sélectionner le tuple indiqué, par le numéro de la colonne courante qui détermine la relation (ex : la colonne 1 indique la relation « *biggest* »qui est la première relation jointe), et par la valeur de cette colonne dans cette ligne qui indique l'identifiant (ID, référence ou numéro de ligne) du tuple dans la relation indiqué.

- projeter les attributs de « *Commons* »sur la relation du tuple sélectionné, puis prendre ce tuple après la projection.

- fusionner les tuples pris de chaque colonne d'une ligne et ajouter le tuple résultant de la fusion de chaque ligne à la nouvelle relation qui sera considérée comme la relation R dans la prochaine jointure.

2. Dans le cas où la file des relation est vide (il n'y a plus de relations à joindre), en effectue une *génération* et *fusion*. Pour cela nous procédons comme suit :

- Parcourir le tableau *GlobalIDs*. Pour chaque ligne de ce tableau, parcourir chaque colonne de cette ligne.

- Sélectionner le tuple indiqué par le numéro de la colonne courante qui détermine la relation et par la valeur de cette colonne dans la ligne courante qui indique l'identifiant (numéro de ligne ou la référence) du tuple dans la relation indiquée.

- Fusionner les tuples d'une ligne et ajouter le tuple résultant de la fusion de chaque ligne à la relation R_{final} qui sera le résultat de jointure de toutes les relations.

5.4.2 Les variantes de l'algorithme *JoinById*

Nous avons défini deux variantes de cet algorithme, à savoir : la variante séquentielle et la variante parallèle.

La variante séquentielle de l'algorithme Cette variante utilise l'algorithme défini auparavant. Cette variante est exécutée sur un processeur central sans parallélisme comme un *CPU* sans multithreading, ni de multi-cœurs.
Les comparaisons des tuples dans l'étape 3 de l'algorithme se font d'une manière consécutive, l'une après l'autre.

La variante parallèle de l'algorithme Cette variante est plus complexe. Adaptée pour le parallélisme de données, cette variante est conçue pour l'utilisation sur les processeurs graphiques *GPU* utilisant le concept du *GPGPU* et les processeurs *CPU* multi-coeurs et multi-threads. Le parallélisme s'effectue dans l'étape 3 de l'algorithme *JoinById*, où les comparaisons des tuples se font d'une manière parallèle (concurrent), de façon que

chaque *thread* exécute une comparaison indépendamment des autres et les *threads* s'exécutent au même temps.
L'étape 3 « comparaison »de cette variable pour les GPU supportant la technologie $CUDA$ est comme suit :

1. Charger en mémoire GPU les relations R' et S'.
2. Déterminer la dimension des blocs et le nombre de *threads* du GPU qui exécuteront les comparaisons en parallèle.
3. Assigner à chaque *thread* du GPU un tuple (ligne) de la relation (table) R'.
4. Lancer l'exécution des *threads* en parallèle, de sorte que, chaque *thread* compare séquentiellement les valeurs de chaque attribut (colonne) du tuple qui lui est assigné, avec les valeurs correspondantes de chaque tuple (ligne) de la relation (table) S'.
5. En cas de concordance parfaite, c'est à dire que pour un tuple (ligne) de relation (table) R', les valeurs des attributs (colonnes) de ce tuple (ligne) sont égales aux valeurs des attributs (colonnes) d'un tuple (ligne) de la relation (table) S', on sauvegarde dans le tableau IDs de 2 colonnes, l'*identifiant* (ID, référence, numéros de ligne) du tuple de R' comparé dans la première colonne et l'*identifiant* du tuple de S' comparé dans la deuxième colonne.
6. À la fin de la comparaison, copier le tableau IDs vers la mémoire centrale (RAM).

Cas exceptionnel A la fin de l'étape 3, si le tableau IDs obtenu est vide (aucun tuple n'a été obtenu après les comparaisons), l'algorithme se termine directement avec une relation vide comme résultat.

5.4.3 L'algorithme *JoinByID*

L'algorithme de jointure sur les bases de données *JoinByID* est défini comme suit :

Algorithm 4 Algorithme de jointure proposé « JoinByID ».

1: Relation **joinByID**(List of Relation Rs)
2: **Var :**
3: R_{final} **: *Relation*** ;
4: R, Rs_{ordred} **: *List of Relation*** ;
5: $Commons$, c **: *List of Attribut*** ;
6: i, n, n_R **: *Integer*** ;
7: IDs, $GlobalIDs$ **: *Matrix of 2 column of Integer*** ;
8: **Begin**
9: //Étape 1
10: $Rs_{ordred} \leftarrow$ **getOptimalOrder**(Rs) ;
11: $Commons \leftarrow$ **getCommonAttributes**(Rs) ;
12: //Étape 2
13: $R \leftarrow Rs_{ordred}[1]$;
14: $i \leftarrow 2$;
15: $n \leftarrow$ **sizeof(**Rs]
16: **while** $i <= n$ **do**
17: $S \leftarrow Rs_{ordred}[2]$;
18: $c \leftarrow$ **getCommonAttributes**(R, S) ;
19: $R' \leftarrow$ **project**(c, R) ;
20: $S' \leftarrow$ **project**(c, S) ;
21: //Étape 3
22: $IDs \leftarrow$ **compare**(R', S') ;
23: **if** IDs **is** *empty* **then**
24: **return** *empty* ;
25: **end if**
26: **if** $i = 1$ **then**
27: $GlobalIDs \leftarrow IDs$;
28: **else**
29: **update**($GlobalIDs, IDs$) ;
30: **end if**
31: **end while**
32: $R_{final} \leftarrow$ **genAndFuse**($GlobalIDs, Rs_{ordred}$) ;
33: **return** R_{final} ;
34: **End.**

5.5 Conclusion

Dans ce chapitre, nous avons présenté la structure de données résultant de la modélisation de la base de données d'un benchmark de Renault, le diagramme des classes représentant la base de données utilisée. Nous avons spécifié le mode de traitement utilisé dans le projet, à savoir, le parallélisme de données ($SIMD$). Nous avons aussi proposé un algorithme de jointure et expliqué son principe de fonctionnement, qui se compose de 4 étapes, à savoir, la détermination de l'ordre de jointure, la projection, la comparaison et la génération et fusion des résultats. Enfin nous avons présenté les variantes de cet algorithme.

Chapitre 6 | Implémentation

6.1 Introduction

Après la conception, nous passons à l'implémentation et à la réalisation du projet. Dans ce chapitre nous présentons l'ensemble des outils utilisés pour la réalisation ainsi que les modes d'utilisation proposés par l'application résultant du projet.
Dans ce chapitre nous présentons l'implémentation de l'algorithme de jointure $JoinById$; à la fin, nous présentons des informations sur les données utilisées du benchmark de *Renault*, les résultats de l'expérimentation avec les données du benchmark ainsi que l'interprétation de ces résultats.

6.2 Présentation des outils de développement

Nous allons présenter brièvement les outils et langages que nous avons utilisés. CUDA est déjà présenté en détail dans le chapitre 2, nous allons donc présenté les autres outils utilisé, comme NVCC, NSight et langages C/C++.

Avant de présenter les outils de développement, nous présentons les caractéristiques de la machine sur laquelle nous avons travaillé :

6.2.1 Caractéristiques de la machine

Afin de réaliser et d'expérimenter le projet, un minimum de matériels est requis. D'autres matériels sont recommandés pour une meilleure performance d'exécution. Parmi ces matériels, nous avons :

- Une carte graphique Nvidia avec support CUDA est requise, nous avons utilisé la carte graphique Nvidia GeForce GT 540M avec support CUDA de 1GB de mémoire.
- Un puissant processeur est recommandé, nous avons utilisé un processeur Intel Core i7 2640M.
- Une grande mémoire vive (RAM) est recommandée, nous avons utilisé une mémoire RAM de 4 Géga octets de mémoire .

6.2.2 logiciels et outils

Au cours du développement nous avons utilisé des programmes, des logiciels et des outils nécessaires à la réalisation de l'application qui est l'implémentation de l'algorithme de jointure $JoinById$. Parmi ces programmes et logiciels utilisés, nous avons :

- Le Système d'exploitation Ubuntu 16.

- L'IDE NSight basé sur Eclipse [1].
- Le SDK, CUDA Toolkit 8.0.
- Le débogueur C++ de GNU, gcc.
- Le parseur (analyseur) de fichiers *XML* : RapidXML.
- L'éditeur de texte Sublime text.

6.3 les Modes d'utilisation

À la réalisation du projet, on à créé 3 modes de traitement des données (jointure), à savoir : les modes CPU, GPU et Hybrid [2].

6.3.1 Mode CPU

Ce mode traite les données de façon séquentielle par le processeur central (CPU), c'est à dire que les tuples (lignes) des relations (tables) jointes sont comparés l'un après l'autre sans concurrence (sans parallélisme). Ce mode utilise l'algorithme proposé dans sa première variante séquentielle.

6.3.2 Mode GPU

Ce mode réalise le traitement des relations (comparaisons des tuples) en parallèle dans le processeur graphique (GPU). Le reste est réalisé sur le CPU, c'est à dire que ce mode démarre dans CPU, exécute les étapes de l'algorithme à l'exception de l'étape où les tuples sont comparés. Cette étape est exécutée dans le GPU où plusieurs comparaisons sont effectuées au même temps par les cœurs du processeurs graphique (GPU). Le nombre de comparaisons effectuées en parallèle dépend des performances du processeur et le nombre de cœurs qu'il inclus (GPU).

6.3.3 Mode Hybride

Ce mode est un peu spécial, il mixe entre les deux précédents modes, « CPU »et « GPU », mais en parallélisant le traitement dans le CPU, de manière à partager la jointure entre les coeurs du "CPU". Ceci est fait en assignant une partie de la jointure à chaque cœur (core) et le "GPU" prend la partie restante. Ce mode fonctionne de la manière suivante : on divise la première relation à joindre de sorte que chaque cœur du CPU prend un nombre fini de lignes (tuples) et le GPU prend le reste et ils exécutent la

1. la version basée sur eclipse est uniquement pour linux.
2. **Hybrid** est un mode qui utilise à la fois le CPU et le GPU.

jointure selon la variante parallèle de l'algorithme $JoinById$. La jointure est effectuée en parallèle entre le GPU et les coeurs du CPU. Comme ce mode mixe et exécute les deux modes CPU et GPU en parallèle, Il donne une plus grande puissance de calcul.

6.4 Présentation des données utilisées

Dans ce projet, nous avons utilisé un benchmark de Renault comme base de données, le benchmark 26. Nous présentons quelques informations sur cette base de données utilisé.

6.4.1 Informations globales

les informations globales de la base de données utilisé sont résumées dans le tableau 6.1 suivant :

Table 6.1: Les caractéristiques globales de la base de données du benchmark 26 de *Renault*.

Les informations générales sur la base de données				
Nombre de relations	Nombre min de tuples	Nombre max de tuples	Nombre min d'attributs	Nombre max d'attributs
159	2	10	3	48721

Le tableau 6.1 montre quelques informations sur la base de données, tel que le nombre de relations que contient la base de données. Il montre aussi le plus petit nombre de tuples que peut contenir une relation (min) et le plus grand nombre de tuples qu'une relation peut avoir (max). Il montre aussi le nombre d'attributs (colonnes) minimal et maximal d'une relation (table) dans la base de données utilisée.

6.4.2 Informations sur les relations (tables) de la base de données

Le benchmark de Renault utilisé contient des relations (tables) de diverses tailles, où ces relations comportent de petits nombres jusqu'à de très grands nombres de tuples, et cela avec différents nombres d'attributs par relation.
les caractéristiques de ces relations sont décrites dans le tableau 6.2 suivant :

Table 6.2: Les caractéristiques des relations de la base de données du benchmark 26 de *Renault*.

La liste des relations de la base de données			
Relations	Nombre d'attributs	Nombre de tuples	Liste des attributs
C0	2	3	V8, V9

Table 6.2: (Suite)

La liste des relations de la base de données			
Relations	Nombre d'attributs	Nombre de tuples	Liste des attributs
C1	2	3	V9, V8
C2	2	3	V12, V14
C3	2	3	V14, V12
C4	2	4	V23, V52
C5	2	4	V52, V23
C6	2	7	V25, V52
C7	2	7	V52, V25
C8	2	42	V28, V46
C9	2	42	V46, V28
C10	2	42	V31, V46
C11	2	42	V46, V31
C12	2	42	V34, V46
C13	2	42	V46, V34
C14	2	3	V36, V41
C15	2	3	V41, V36
C16	2	3	V36, V81
C17	2	3	V81, V36
C18	2	42	V39, V46
C19	2	42	V46, V39
C20	2	42	V46, V50
C21	2	42	V50, V46
C22	2	42	V46, V56
C23	2	42	V56, V46
C24	2	42	V46, V69
C25	2	42	V69, V46
C26	2	42	V46, V70
C27	2	42	V70, V46
C28	2	42	V46, V82
C29	2	42	V82, V46
C30	2	3	V51, V75
C31	2	3	V75, V51

Table 6.2: (Suite)

La liste des relations de la base de données			
Relations	Nombre d'attributs	Nombre de tuples	Liste des attributs
C32	2	3	V64, V87
C33	2	3	V87, V64
C34	2	3	V64, V88
C35	2	3	V88, V64
C36	2	3	V76, V78
C37	2	3	V78, V76
C38	2	3	V79, V81
C39	2	3	V81, V79
C40	2	10	V80, V85
C41	2	10	V85, V80
C42	3	29	V0, V23, V44
C43	3	42	V0, V26, V52
C44	3	378	V0, V38, V46
C45	3	378	V0, V46, V53
C46	3	378	V0, V46, V83
C47	3	25	V1, V24, V95
C48	3	77	V4, V24, V90
C49	3	164	V4, V41, V46
C50	3	248	V5, V32, V106
C51	3	97	V17, V33, V54
C52	3	99	V24, V41, V81
C53	3	125	V24, V45, V95
C54	3	1050	V24, V46, V57
C55	3	25	V24, V79, V95
C56	3	25	V24, V90, V95
C57	3	9	V42, V78, V99
C58	3	488	V101, V104, V109
C59	3	211	V101, V105, V108
C60	3	247	V101, V105, V110
C61	3	175	V102, V103, V107
C62	3	997	V102, V106, V109

Table 6.2: (Suite)

La liste des relations de la base de données			
Relations	Nombre d'attributs	Nombre de tuples	Liste des attributs
C63	3	461	V102, V107, V109
C64	3	224	V103, V104, V109
C65	3	139	V103, V105, V108
C66	3	409	V104, V105, V107
C67	4	61	V0, V6, V9, V13
C68	4	91	V0, V13, V17, V58
C69	4	966	V0, V13, V41, V46
C70	4	70	V0, V13, V58, V98
C71	4	58	V0, V15, V43, V44
C72	4	43	V0, V19, V23, V43
C73	4	408	V0, V24, V40, V95
C74	4	455	V0, V24, V64, V81
C75	4	406	V0, V24, V94, V95
C76	4	199	V6, V24, V68, V81
C77	4	199	V13, V24, V64, V95
C78	4	53	V37, V50, V80, V88
C79	4	117	V37, V85, V91, V105
C80	4	1966	V101, V102, V104, V107
C81	4	1430	V101, V102, V108, V109
C82	4	1655	V101, V104, V105, V108
C83	4	1682	V101, V104, V105, V109
C84	4	1176	V101, V104, V107, V109
C85	4	115	V101, V105, V106, V107
C86	4	1156	V101, V105, V106, V108
C87	4	769	V101, V107, V108, V110
C88	4	1006	V103, V104, V106, V108
C89	4	343	V103, V105, V107, V109
C90	4	1985	V104, V106, V107, V108
C91	5	984	V0, V6, V13, V24, V95
C92	5	135	V0, V12, V13, V43, V57
C93	5	146	V0, V13, V15, V43, V57

Table 6.2: (Suite)

La liste des relations de la base de données			
Relations	Nombre d'attributs	Nombre de tuples	Liste des attributs
C94	5	365	V0, V13, V21, V24, V95
C95	5	14238	V0, V13, V24, V46, V71
C96	5	336	V0, V13, V24, V74, V95
C97	5	394	V0, V13, V24, V87, V95
C98	5	8694	V4, V13, V24, V46, V95
C99	5	1396	V6, V13, V24, V95, V99
C100	6	94	V0, V5, V13, V89, V92, V99
C101	6	80	V0, V9, V13, V89, V92, V99
C102	6	48	V0, V10, V13, V89, V92, V99
C103	6	48	V0, V11, V13, V89, V92, V99
C104	6	48	V0, V13, V14, V89, V92, V99
C105	6	48	V0, V13, V18, V89, V92, V99
C106	6	50	V0, V13, V19, V89, V92, V99
C107	6	60	V0, V13, V20, V89, V92, V99
C108	6	54	V0, V13, V22, V89, V92, V99
C109	6	57	V0, V13, V23, V89, V92, V99
C110	6	48721	V0, V13, V24, V46, V88, V95
C111	6	80	V0, V13, V25, V89, V92, V99
C112	6	60	V0, V13, V26, V89, V92, V99
C113	6	153	V0, V13, V29, V89, V92, V99
C114	6	50	V0, V13, V30, V89, V92, V99
C115	6	93	V0, V13, V37, V89, V92, V99
C116	6	48	V0, V13, V42, V89, V92, V99
C117	6	54	V0, V13, V44, V89, V92, V99
C118	6	48	V0, V13, V47, V89, V92, V99
C119	6	81	V0, V13, V48, V89, V92, V99
C120	6	79	V0, V13, V52, V89, V92, V99
C121	6	48	V0, V13, V59, V89, V92, V99
C122	6	48	V0, V13, V60, V89, V92, V99
C123	6	48	V0, V13, V62, V89, V92, V99
C124	6	57	V0, V13, V63, V89, V92, V99

Table 6.2: (Suite)

La liste des relations de la base de données			
Relations	Nombre d'attributs	Nombre de tuples	Liste des attributs
C125	6	48	V0, V13, V65, V89, V92, V99
C126	6	48	V0, V13, V66, V89, V92, V99
C127	6	48	V0, V13, V67, V89, V92, V99
C128	6	48	V0, V13, V72, V89, V92, V99
C129	6	48	V0, V13, V73, V89, V92, V99
C130	6	94	V0, V13, V76, V89, V92, V99
C131	6	88	V0, V13, V77, V89, V92, V99
C132	6	84	V0, V13, V78, V89, V92, V99
C133	6	218	V0, V13, V80, V89, V92, V99
C134	6	48	V0, V13, V84, V89, V92, V99
C135	6	92	V0, V13, V85, V89, V92, V99
C136	6	53	V0, V13, V86, V89, V92, V99
C137	6	140	V0, V13, V89, V92, V93, V99
C138	7	1952	V0, V7, V13, V24, V89, V92, V99
C139	7	1708	V0, V8, V13, V24, V89, V92, V99
C140	7	3285	V0, V13, V16, V46, V89, V92, V99
C141	7	3516	V0, V13, V17, V46, V89, V92, V99
C142	7	32072	V0, V13, V24, V46, V81, V95, V99
C143	7	1114	V0, V13, V24, V54, V89, V92, V99
C144	7	1509	V0, V13, V24, V55, V89, V92, V99
C145	7	2259	V0, V13, V24, V89, V91, V92, V99
C146	7	2025	V0, V13, V46, V51, V89, V92, V99
C147	7	2624	V0, V13, V46, V58, V89, V92, V99
C148	7	5385	V0, V13, V46, V75, V89, V92, V99
C149	7	2591	V0, V13, V46, V89, V92, V98, V99
C150	7	130	V0, V13, V49, V57, V89, V92, V99
C151	7	106	V0, V13, V57, V61, V89, V92, V99
C152	8	4440	V0, V12, V13, V46, V57, V89, V92, V99
C153	8	3401	V0, V13, V15, V24, V57, V89, V92, V99
C154	8	2101	V0, V13, V24, V27, V57, V89, V92, V99
C155	8	33437	V0, V13, V24, V36, V46, V68, V95, V99

Table 6.2: (Suite)

La liste des relations de la base de données			
Relations	Nombre d'attributs	Nombre de tuples	Liste des attributs
C156	8	6498	V0, V13, V43, V46, V57, V89, V92, V99
C157	9	164	V0, V2, V3, V24, V81, V95, V96, V97, V100
C158	10	342	V0, V13, V24, V35, V57, V68, V89, V92, V95, V99

Le tableau 6.2 montre les caractéristiques de chaque relation (table) du benchmark N°26 de *Renault*, à savoir, le nombre de tuples (lignes)et le nombre d'attributs (colonnes) ainsi que la liste des attributs.

6.5 Présentation des Résultats

Après l'implémentation et la réalisation du projet, nous procédons à son expérimentation, ceci en exécutant l'application résultante dans les trois modes décrits au préalable. L'exécution de l'application s'est faite avec le benchmark N°26 de *Renault*, et nous avons récolté des résultats qui sont comme suit :

6.5.1 Résultats obtenus en séquentiel (mode CPU)

Dans cette partie nous avons fait deux exécutions dans le mode CPU sur la même machine en changeant les relations (tables) à joindre. Les résultats de ces tests sont montrés dans les tableaux suivants :

1. **Premier test :**
   Dans ce test, nous joignons les relations suivantes dans l'ordre : C1, C2, C45, C92, C22, C61, C58.

Table 6.3: Les résultats du premier test en mode CPU.

Relations	Attributs en commun	R1	R2	comparés	résultant	Temps (s)
$C157 \bowtie C94$	4	164	365	59860	563	0,004558
$Résultat \bowtie C110$	5	563	48721	27429923	23648	0,082779
$Résultat \bowtie C55$	3	23648	25	591200	23648	0,246185
$Résultat \bowtie C53$	3	23648	125	2956000	47296	0,256926
$Résultat \bowtie C76$	3	47296	199	9411904	187922	0,497404

Table 6.3: (Suite)

Relations	Attributs en commun	R1	R2	comparés	résultant	Temps (s)
$Résultat \bowtie C144$	4	187922	1509	283574298	850192	2,090676
$Résultat \bowtie C95$	5	850192	14238	12105033696	850192	19,406248
$Résultat \bowtie C16$	2	850192	3	2550576	1562528	9,193176
$Résultat \bowtie C43$	2	1562528	42	65626176	9125680	16,858118

Le tableau suivant résume les résultats obtenus en faisant point sur le maximum, le minimum et la moyenne du temps de traitement (jointure) avec le nombre de comparaison effectuées lors de la jointure.

2. **Deuxième test :**
   Dans ce test, nous joignons les relations suivantes dans l'ordre : C158, C153, C151, C150, C139, C145, C103, C116, C122, C126, C129, C114, C117.

Table 6.4: Les résultats du deuxième test en mode CPU.

Relations	Attributs en commun	R1	R2	comparés	résultant	Temps (s)
$C158 \bowtie C153$	7	342	3401	1163142	596	0,009047
$Résultat \bowtie C151$	6	596	106	63176	679	0,004451
$Résultat \bowtie C150$	6	679	130	88270	1103	0,005007
$Résultat \bowtie C139$	6	1103	1708	1883924	2083	0,008878
$Résultat \bowtie C145$	6	2083	2259	4705497	5258	0,022337
$Résultat \bowtie C103$	5	5258	48	252384	5258	0,054598
$Résultat \bowtie C116$	5	5258	48	252384	5258	0,051784
$Résultat \bowtie C122$	5	5258	48	252384	5258	0,056681
$Résultat \bowtie C126$	5	5258	48	252384	5258	0,05559
$Résultat \bowtie C129$	5	5258	48	252384	5258	0,054792
$Résultat \bowtie C114$	5	5258	50	262900	5382	0,049332
$Résultat \bowtie C117$	5	5382	54	290628	6058	0,047271

6.5.1.1 Analyse des résultats obtenus en séquentiel Après avoir récupéré et traité les résultats des exécutions, nous avons pu mettre en place le tableau 6.5 qui représente le temps d'exécution et la vitesse de comparaison qui est en fonction du nombre de tuples (lignes) comparés et le temps de traitement de la comparaison.

Table 6.5 – Résumé des temps d'exécution (jointure) en mode CPU (séquentiel).

	Tuples comparés	Temps de traitement
Minimum	59860	0,004451
Maximum	12105033696	19,406248

Nous tirons des tests effectués les conclusions suivantes :

- La vitesse moyenne de comparaison des exécutions faites en mode CPU est de 254953408 tuples/s.
- Le temps moyen de jointure des exécutions faites en mode CPU est de 2,719 s.

6.5.2 Résultats obtenus en mode GPU (parallèle)

Dans cette partie nous avons aussi fait deux exécutions sur la même machine mais en mode GPU. Les relations (tables) jointes dans ces exécutions changent à chaque fois. Les résultats de ces tests sont montrés dans les tableaux suivants :

1. **Premier test :**
   Dans ce test, nous joignons les relations suivantes dans l'ordre : C1, C2, C45, C92, C22, C61, C58.

Table 6.6: Les résultats du premier test en mode GPU.

Relations	Attributs en commun	R1	R2	comparés	résultant	Temps (s)
$C157 \bowtie C94$	4	164	365	59860	563	0,047984
$Résultat \bowtie C110$	5	563	48721	27429923	23648	0,573523
$Résultat \bowtie C55$	3	23648	25	591200	23648	0,651181
$Résultat \bowtie C53$	3	23648	125	2956000	47296	0,735907
$Résultat \bowtie C76$	3	47296	199	9411904	187922	0,898719
$Résultat \bowtie C144$	4	187922	1509	283574298	850192	1,630918
$Résultat \bowtie C95$	5	850192	14238	12105033696	850192	12,179707
$Résultat \bowtie C16$	2	850192	3	2550576	1562528	5,723484
$Résultat \bowtie C43$	2	1562528	42	65626176	9125680	11,24204

Le tableau suivant résume les résultats obtenus en faisant point sur le maximum, le minimum et la moyenne du temps de traitement (jointure) avec le nombre de comparaisons effectuées lors de la jointure.

Table 6.7 – Analyse des valeurs obtenues du premier test en mode GPU.

2. **Deuxième test :**
   Dans ce test, nous joignons les relations suivantes dans l'ordre : C7, C13, C52, C66, C100, C81, C39.

Table 6.8: Les résultats du deuxième test en mode *GPU*.

Relations	Attributs en commun	R1	R2	comparés	résultant	Temps (s)
$C158 \bowtie C153$	7	342	3401	1163142	596	0,068755
$Résultat \bowtie C151$	6	596	106	63176	679	0,073989
$Résultat \bowtie C150$	6	679	130	88270	1103	0,077106
$Résultat \bowtie C139$	6	1103	1708	1883924	2083	0,08922
$Résultat \bowtie C145$	6	2083	2259	4705497	5258	0,104891
$Résultat \bowtie C103$	5	5258	48	252384	5258	0,10968
$Résultat \bowtie C116$	5	5258	48	252384	5258	0,116743
$Résultat \bowtie C122$	5	5258	48	252384	5258	0,123574
$Résultat \bowtie C126$	5	5258	48	252384	5258	0,130016
$Résultat \bowtie C129$	5	5258	48	252384	5258	0,13685
$Résultat \bowtie C114$	5	5258	50	262900	5382	0,143652
$Résultat \bowtie C117$	5	5382	54	290628	6058	0,148624

6.5.2.1 Analyse des résultats obtenus en mode *GPU* (parallèle) Après avoir récupéré et traité les résultats des exécutions, nous avons pu mettre en place le tableau 6.9 qui représente le temps d'exécution et la valeur de la vitesse de comparaison en fonction du nombre de tuples (lignes) comparés et du nombre d'attributs (colonnes) comparés.

Table 6.9 – Résumé des temps d'exécution (jointure) en mode *GPU* (parallèle).

	Tuples comparés	Temps de traitement
Minimum	59860	0,047984
Maximum	12105033696	12,179707

Nous tirons des tests effectués les conclusions suivantes :

- La vitesse moyenne de comparaison des exécutions faites en mode *GPU* est de 357274522 tuples/s.
- Le temps moyen de jointure des exécutions faits en mode *GPU* est de 1,92 s.

6.5.3 Résultats obtenus en mode *Hybrid*

Les exécutions faites dans cette partie sont aussi effectuées sur la même machine en mode *Hybrid*. Les relations (tables) jointes sont changées à chaque nouvelle exécution.

Les résultats de ces tests sont montrés dans les tableaux suivants :

1. **Premier test :**
   Dans ce test, nous joignons les relations suivantes dans l'ordre : C1, C2, C45, C92, C22, C61, C58.

   Table 6.10: Les résultats du premier test en mode *Hybrid*.

Relations	Attributs en commun	R1	R2	comparés	résultant	Temps (s)
$C157 \bowtie C94$	4	164	365	59860	563	0,000085
$Résultat \bowtie C110$	5	563	48721	27429923	23648	0,031544
$Résultat \bowtie C55$	3	23648	25	591200	23648	0,0008
$Résultat \bowtie C53$	3	23648	125	2956000	47296	0,003276
$Résultat \bowtie C76$	3	47296	199	9411904	187922	0,010794
$Résultat \bowtie C144$	4	187922	1509	283574298	850192	0,329443
$Résultat \bowtie C95$	5	850192	14238	12105033696	850192	13,534649
$Résultat \bowtie C16$	2	850192	3	2550576	1562528	0,005931
$Résultat \bowtie C43$	2	1562528	42	65626176	9125680	0,091325

   Le tableau suivant résume les résultats obtenus en faisant point sur le maximum, le minimum et la moyenne du temps de traitement (jointure) avec le nombre de comparaisons effectuées lors de la jointure.

2. **Deuxième test :**
   Dans ce test, nous joignons les relations suivantes dans l'ordre : C158, C153, C151, C150, C139, C145, C103, C116, C122, C126, C129, C114, C117.

   Table 6.11: Les résultats du deuxième test en mode *Hybrid*.

Relations	Attributs en commun	R1	R2	comparés	résultant	Temps (s)
$C158 \bowtie C153$	7	342	3401	1163142	596	0,001745
$Résultat \bowtie C151$	6	596	106	63176	679	0,000151
$Résultat \bowtie C150$	6	679	130	88270	1103	0,000211
$Résultat \bowtie C139$	6	1103	1708	1883924	2083	0,003046
$Résultat \bowtie C145$	6	2083	2259	4705497	5258	0,007036
$Résultat \bowtie C103$	5	5258	48	252384	5258	0,000509
$Résultat \bowtie C116$	5	5258	48	252384	5258	0,000506
$Résultat \bowtie C122$	5	5258	48	252384	5258	0,000497
$Résultat \bowtie C126$	5	5258	48	252384	5258	0,000516

Table 6.11: (Suite)

Relations	Attributs en commun	R1	R2	comparés	résultant	Temps (s)
$Résultat \bowtie C129$	5	5258	48	252384	5258	0,000498
$Résultat \bowtie C114$	5	5258	50	262900	5382	0,000524
$Résultat \bowtie C117$	5	5382	54	290628	6058	0,000582

6.5.3.1 Analyse des résultats obtenus en mode *Hybrid* Après avoir récupéré et traité les résultats des exécutions, nous avons pu mettre en place le tableau 6.12 qui représente le temps d'exécution et la valeur de la vitesse de comparaison en fonction du nombre de tuples (lignes) comparés et du nombre d'attributs (colonnes) comparés.

Table 6.12 – Résumé des temps d'exécution (jointure) en mode *Hybrid*.

	Tuples comparés	Temps de traitement
Minimum	59860	0,000085
Maximum	12105033696	13,534649

Après l'exécution des tests précédents, nous tirons les conclusions suivantes :

- La vitesse moyenne de comparaison des exécutions faites en mode *Hybrid* est de 891846062 tuples/s.
- Le temps moyen de jointure des exécutions faites en mode *Hybrid* est de 1,557 s.

6.6 Conclusion

Dans ce chapitre et après avoir vu les chapitres précédents, nous présentons les outils de développement utilisés au cours de la réalisation du projet et les caractéristiques de la machine utilisée pour le développement et l'expérimentation du projet. Nous avons présenté aussi les modes d'utilisation de l'application, à savoir : *CPU*, *GPU* et *Hybrid*, la base de données utilisé pour l'expérimentation de la jointure avec ces trois modes et enfin les résultats de l'expérimentation. L'expérimentation a démontré l'efficacité de l'algorithme $JoinById$ ainsi que son parallélisme avec $CUDA$ sur le GPU, et le mode $Hybrid$.

Conclusion générale

Dans ce travail modéste, nous avons contribué par une proposition d'une solution à une problématique des bases de données. Dans certains cas, cette problématique est souvent présente.

Nous avons exposé dans le premier chapitre, les différentes architectures des ordinateurs dans leurs globalités, à savoir : séquentielle, parallèle et distrbuée. Nous avons aussi vu les architectures selon le classement fait par *Flynn*.

Dans le deuxième chapitre, nous avons présenté les processeurs graphiques (GPU), la téchnologie $CUDA$ de l'entreprise *Nvidia*®, l'architecture $CUDA$ ainsi que son principe de fonctionnement.

Nous avons présenté dans le troisième chapitre, les algorithmes de jointure, nous avons vu les trois algorithmes les plus connus dans le domaine. Nous avons décrit leurs principes et présenté quelques avantages de ces algorithmes ainsi que certains de leurs inconvénients.

Dans le quatrième chapitre, nous avons présenté deux travaux faits dans le domaine des jointures de tables sur les bases de données relationnelles. Ces deux travaux ont été faits de différentes approches. Le premier a été réalisé en paralléle sur les processeurs CPU multi-coeurs et le second a été réalisé avec une approche distrbuée.

Dans le cinquième chapitre, nous avons présenté notre conception du projet. La conception de la structure de données utilisé, les approches utilisées ainsi que l'algorithme $JoinByID$ proposé comme solution pour le probléme étudié.

Le Sixième et dérnier chapitre dans ce mémoire, présente l'implémentation de notre projet. Nous avons présenté les différents outils matériels et logiciels utilisés afin de développer et d'implémenter l'algorithme proposé dans les différents modes décrits dans ce chapitre. Nous avons présenté à la fin de ce chapitre les données utilisées ainsi que les résultats de l'experimentation de ces données dans l'application implémentée.

Nous projetons comme perspective pour ce travail,l'implémentation d'un algorithme de jointure proche de celui proposé ($JoinByID$). Cet algorithme sera basé sur l'algorithme de jointure par hachage (*Hash-Join*) qui sera implémenté dans les mêmes modes utilisés dans le cadre de ce travail, à savoir : séquentiel sur CPU, parallèle sur GPU et Hybride sur le CPU et le GPU.

Bibliographie

[1] Some Computer Organizations and Their Effectiveness, M. Flynn, IEEE Trans. Comput., Vol. C-21, p. 948, 1972.

[2] Anand Lal Shimpi et Wilson, Derek, « Nvidia's GeForce 8800 (G80) : GPUs Re-architected for DirectX 10 » [archive], AnandTech, 8 novembre 2006

[3] Parallel Computing for Real-time Signal Processing and Control, Mohammad Osman Tokhi, 2003.

[4] Design and Evaluation of Main Memory Hash Join Algorithms for Multi-core CPUs, Spyros Blanas, Yinan Li et Jignesh M. Patel, University of Wisconsin–Madison.

[5] Join Algorithms using Map/Reduce, Jairam Chandar, University of Edinburgh, 2010.

[6] http ://www.cril.univ-artois.fr/CPAI08/results/bench.php ?idev=15&idbench=56362

[7] http ://www.nvidia.fr/object/cuda-parallel-computing-fr.html

[8] Bases de données avancées : Évaluation et optimisation des requêtes, Dan VODISLAV, Université de Cergy-Pontoise.

Table des figures

Liste des tableaux

Résumé

L'optimisation du temps de traitement des opérations sur les bases de données ont toujours été un sujet très étudié aussi bien par les développeurs que par les chercheurs. La jointure étant l'une de ces opérations, elle est parmi les opérations les plus couteuses ainsi que parmi les opérations les plus utilisées. L'apparition des systèmes parallèles a pu guidé les études ainsi que les recherches vers des techniques de résolutions de plus en plus optimales, en divisant en ensemble de tâches destinées à être réparties sur différents processeurs et/ou cœurs. Quel algorithme utiliser? Comment paralléliser la jointure? Sur quelle plateforme devrait-on effectuer l'opération? Quel matériel utiliser? C'est à ces questions que nous avons essayé de répondre dans ce travail en proposant un algorithme de jointure et une méthode d'implémentation de cet algorithme en parallèle sous la plateforme CUDA, avec les processeurs graphiques GPU, ainsi qu, une hybridation entre les GPU et les processeurs CPU multi-cœurs.

Mots clés : *algorithme de jointure, parallèle, CUDA, processeur graphique (GPU), processeur CPU multi-cœurs.*

Abstract

The optimization of the processing time of the operations on the databases have always been a subject studied very much, as by the developers than by the researchers. Being one of these operations, the join is among the most costly and most commonly used operations. The appearance of parallel systems has been able to guide studies and research towards increasingly optimal resolution techniques, by dividing this operation to a set of tasks, intended to be distributed over different processors and / or cores. Which algorithm to use? How to parallelize the join? On which platform should the operation be carried out? Which material to use? It's to these questions that we tried to answer in this work by proposing a joining algorithm and an implementation method of this algorithm in parallel under the CUDA platform, with the GPU graphic processors, as well as with the GPU and a multi-core's CPU processors.

Keywords : *join algorithm, parallel, CUDA, graphics processing unit (GPU), central processing unit (CPU) multi-cores.*