Université Laval

Faculté de sciences et génie

Department d'informatique et de génie logiciel

HydraSolveur

Exploration de la parallélisation des traitements dans un solveur

Éloi Brassard-Gourdeau 111 103 323

Pierre-Marc Levasseur 111 080 897

Evan Rausch-Larouche 111 106 326

Jean-François Tremblay 111 098 555

17 avril 2017

1 Introduction

L'utilisation des processeurs graphiques (GPU) pour paralléliser des traitements de données est de plus en plus populaire de nos jours dans une grande variété d'applications. En effet, les processeurs graphiques modernes sont extrêmement puissants et ils permettent d'accélérer significativement des calculs très long, particulièrement lorsqu'ils sont simples. C'est le genre de calculs que l'on peut retrouver dans, par exemple, le filtrage de contraintes dans un solveur. Bien utilisée, nous croyons que la parallélisation des algorithmes de filtrage avec le processeur graphique pourrait grandement diminuer le temps de résolution de certains problèmes par des solveurs. Nous avons donc décidé de faire un projet exploratoire sur ce sujet: construire un solveur, paralléliser certains traitements plus longs et voir à quel point nous pouvons améliorer les temps de calculs.

Un autre aspect de notre projet est la parallélisation au niveau du processeur central (CPU). Les cœurs individuels des processeurs ne gagnent pratiquement plus en performance ; au lieu d'inclure des cœurs plus rapides, les concepteurs de processeurs réussissent à mettre de plus en plus de cœurs sur une même puce. Même si cela peut naïvement mener à la conclusion que les programmes s'accéléreront en utilisant tous les cœurs du processeur, ce n'est souvent pas le cas. Les solveurs de contraintes tirent rarement avantage des multiples cœurs présents dans un ordinateur (Choco le fait[1]) et ceci en fait un sujet intéressant à explorer.

2 Description du problème

Tel que mentionné dans l'introduction, l'objectif de ce projet est de développer un solveur de base qui utilisera les ressources offertes par les processeurs graphiques modernes pour paralléliser le filtrage des contraintes. Étant donnée l'architecture de ces processeurs graphiques, certains algorithmes de filtrage se prêtent mieux à un traitement hautement parallélisé que d'autres. L'objectif du projet sera donc d'identifier et de développer des algorithmes de filtrage qui tireront avantage de l'architecture spécifique des GPU. Nous pourrons ensuite comparer leurs performances lorsqu'ils sont exécutés sur le processeur central et sur le GPU en terme de retours arrières et de temps d'exécution.

De plus, nous allons aussi tirer avantage du fait que les processeurs ont plusieurs cœurs de calculs en parallélisant l'exploration de l'arbre de recherche sur le CPU. L'objectif de ceci est tout simplement d'exploiter une heuristique aléatoire en lançant plusieurs fois le même problème dans plusieurs fils d'exécution (threads), et en prenant la réponse du premier qui résout le problème.

2.1 Processeurs graphiques et CUDA

CUDA est une manière de programmer les processeurs graphiques de la marque Nvidia pour faire du GPGPU. Le GPGPU est un acronyme pour *general purpose graphics processessing unit*, ce qui signifie qu'il nous est possible d'utiliser le matériel traditionnellement limité au traitement et à l'affichage d'images pour résoudre des problèmes plus généraux.

Ceci peut apporter des gains de performance très importants puisque les processeurs graphiques sont massivement parallèles, pouvant posséder plus de 3000 cœurs. Toutefois, cela vient avec un lourd coût: une architecture SIMD (single instruction, multiple data). Un grand groupe de threads lancés en parallèle doit donc effectuer la même tâche sur des données différentes. Il faut ensuite trouver un moyen de combiner tous ces résultats pour récupérer la solution au problème initial. Ce genre d'architecture est bien adapté à des calculs d'algèbre linéaire (d'ailleurs très dominants en infographie), mais certains problèmes sont difficilement décomposables de cette façon. La résolution de CSP (*constraint solving problem*) par exploration d'arbre est un de ces problèmes; les branchements conditionnels constants rendraient cet algorithme difficile à implémenter directement sur un GPGPU. Toutefois, dans ce projet, nous avons identifié quelques sous-problèmes clés qui eux pourraient bénéficier d'une accélération en utilisant un processeur graphique.

Les processeurs graphiques Nvidia ont une architecture particulière qui se reflète beaucoup dans le modèle de programmation. Une représentation graphique de cette architecture se trouve dans la figure 1. Il faut d'abord programmer ce qu'on appelle un *kernel*, c'est-à-dire la fonction exécutée par chaque thread sur le processeur graphique. On lance ensuite ce *kernel* sur un ou plusieurs blocs de une, deux ou trois dimensions. Chaque thread connait sa position dans son bloc d'exécution via son *threadid* et c'est en utilisant cette information qu'il est possible de personnaliser les traitements. Prenons la tâche d'additionner deux matrices de taille 100×100 pour exemple: premièrement, il faudrait programmer un *kernel* qui récupérerait les éléments en (x, y) (via sa position dans le bloc bi-dimensionnel) dans les deux matrices d'entrée et qui écrirait le résultat de la somme des éléments dans la matrice

[1] http://choco-solver.readthedocs.io/en/latest/3_solving.html#multi-thread-resolution

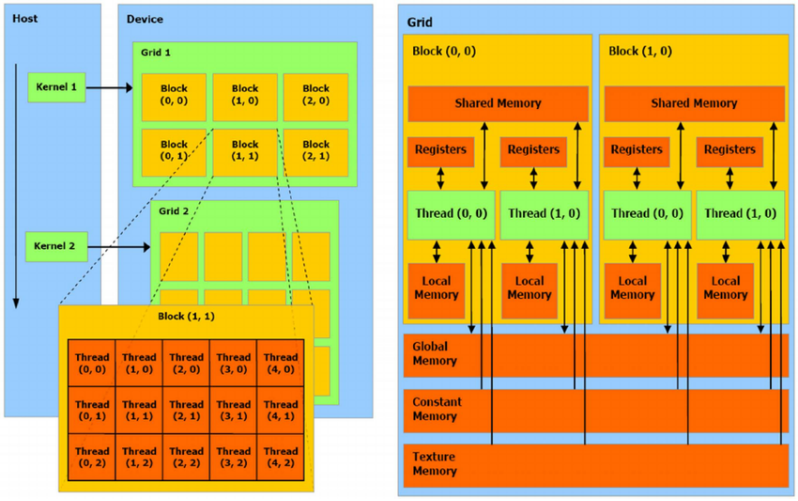


Figure 1: Schéma de l'architecture CUDA. On voit comment l'exécution des threads est répartie sur plusieurs blocs, ainsi que l'architecture en couche de la mémoire; cette dernière rappelle l'architecture multicouche de la mémoire RAM (cache L1, L2, L3 et RAM), avec la même augmentation de la latence à chaque couche, sauf que cette fois le programmeur a un contrôle total sur la location de la mémoire allouée. Source : *CUDA Programming Guide* de Nvidia.

résultat à la même position. Ensuite, le programme lancerait ce *kernel* sur un bloc de 100×100 en lui passant en paramètre l'adresse des deux matrices qu'il faut additionner ainsi qu'un pointeur vers un espace mémoire pour écrire la réponse.

Cet exemple nous mène à parler de l'architecture mémoire de CUDA. En effet, les *kernels* CUDA peuvent seulement lire les adresses mémoire situées sur le GPU (NB: les GPUs modernes possèdent maintenant entre 2 et 16 gigaoctets de RAM). Le programmeur doit donc, en utilisant les fonctions `cudaMalloc` et `cudaMemcpy`, allouer de la mémoire sur le GPU et copier les données qu'il veut traiter de la mémoire utilisée par le CPU vers celle utilisée par le processeur graphique. `cudaMemcpy` permet également de transférer des données entre les deux espaces mémoire. À l'intérieur du GPU, on retrouve plusieurs niveaux de mémoire[2] :

1. *Global memory*: mémoire accessible par tout les *threads* de tout les blocs
2. *Constant memory*: même niveau que la *global memory* mais en lecture seule.
3. *Shared memory*: cette mémoire est accessible par tous les threads d'un bloc donné.
4. *Local memory*: accessible seulement par le thread courant. Peu utilisée; un thread peut souvent tout stocker ses variables locales dans ses registres.
5. *Registers*: les registres du cœur du GPU sur lequel le thread courant s'exécute. Il peut y en avoir jusqu'à 255 par coeurs, ce qui est énorme par rapport au 16 registres par cœur x86 moderne. Chaque variable déclarée sur la pile d'un *kernel* est, en conséquence, presque garantie d'être dans un registre, rendant l'exécution d'un programme très rapide.

L'utilisation judicieuse de ces différents niveaux de mémoire est cruciale pour atteindre un bon niveau de performance.

3 Approche proposée

3.1 Architecture du solveur

Dans cette section, nous allons étudier l'architecture générale du solveur construit pour ce projet. L'objectif n'est pas de fournir une documentation exhaustive des différentes classes, mais bien de faire un tour d'horizon du solveur

[2]La mémoire de texture n'est pas mentionnée ici puisqu'elle n'est pas beaucoup utilisée en GPGPU.

et de mettre en évidence les différentes décisions qui nous ont permis de facilement intégrer les fonctionnalités de CUDA dans un solveur de base[3].

Commençons d'abord par la représentation des variables dans le solveur. Nous avons la classe abstraite *Variable* qui offrent les opérations de base pour obtenir les informations relatives au domaine d'une variable (cardinalité, borne inférieure, borne supérieure). Elle offre aussi les opérations pour filtrer les valeurs de son domaine de différentes manières et pour l'instancier. Plusieurs classes sont dérivées de cette dernière, mais l'implémentation qui nous intéresse pour les algorithmes de filtrage des sections suivantes est la classe *BitsetIntVariable*. Comme son nom l'indique, cette classe représente une variable dont le domaine contient des valeurs entières et dont l'appartenance des valeurs est stockée dans un vecteur de bits. La classe conserve donc la borne inférieure originale qu'elle utilise pour calculer l'index d'une valeur précise dans son vecteur de bits. Ainsi, le filtrage d'une valeur se fait en temps constant; il s'agit simplement de calculer l'index de la valeur en question et de mettre son bit correspondant à 0. Nous avons décidé d'utiliser cette structure de données pour minimiser l'espace mémoire requis pour représenter une variable et son domaine pour faciliter plus tard le transfert d'informations entre le CPU et le GPU.

Ensuite, nous avons la classe abstraite *Constraint* qui représente évidemment une contrainte. Cette classe offre les opérations pour filtrer le domaine ou les bornes des variables qu'elle contient. Elle permet aussi de vérifier si elle agit sur une variable donnée et elle permet à l'utilisateur de savoir si elle est satisfaite depuis la dernière fois que son algorithme de filtrage a été appelé. Nous avons bien sûr plusieurs implémentations de *Constraint* (*AllDifferent*, *SumConstraint*, etc.), mais nous allons seulement étudier certaines d'entre elles dans les prochaines sections lorsque nous regarderons leurs algorithmes de filtrage. Un détail important ici est que les contraintes peuvent être configurées individuellement pour utiliser ou pas leur algorithme de filtrage utilisant le GPU. Il est donc possible d'instancier plusieurs contraintes différentes et de faire en sorte que seulement certaines d'entre elles utilisent le processeur graphique tout dépendant des performances de leur algorithme.

Nous avons aussi une classe *Model* qui contient les variables et les contraintes d'un problème quelconque. La classe *Solver* utilise donc cette dernière pour résoudre un problème de satisfaction et renvoyer une solution. *Solver* contient l'algorithme de fouille en profondeur avec filtrage. On peut configurer une instance de cette classe pour qu'elle atteigne la cohérence de domaine ou d'intervalle selon le choix de l'utilisateur. *Solver* implémente aussi un système de redémarrage basé sur une suite géométrique. Ainsi, la fouille peut recommencer si le solveur atteint une certaine limite de retours arrière. Ensuite, pour gérer la propagation des contraintes, nous avons la classe *Propagator* qui s'occupe de filtrer les contraintes et d'atteindre la cohérence locale ou un état incohérent (lorsqu'une variable a un domaine vide). La classe *Solver* utilise aussi la classe *VariableSelector* qui s'occupe d'instancier une variable à chaque cycle de l'algorithme de recherche. Cette classe contient donc les différentes heuristiques de sélection de variable. Nous allons principalement utiliser une heuristique aléatoire dans les sections suivantes.

Finalement, la classe *Solver* génère une instance de la classe *Solution* qui contient la solution et différentes statistiques sur l'exécution du solveur (temps d'exécution, nombre de retours arrière, etc.).

3.2 Parallélisation de l'algorithme de fouille en profondeur

Notre projet vise à améliorer le temps de résolution de problèmes de satisfaction de contraintes en général. Nous remarquons qu'un solveur typique n'utilise qu'un seul coeur du processeur pour obtenir une solution pour un problème donné rendant les autres complètement inutiles. Nous proposons alors la parallélisation du solveur sur les différents coeurs du CPU pour améliorer le temps de résolution. Nous considérons que l'optimisation de la latence pour l'obtention d'une solution est plus importante que l'optimisation du débit. C'est entre autres la différence majeure entre la parallélisation sur le CPU et sur le GPU.

Nous avons créé une nouvelle classe *MultiAgentSolver* ayant les mêmes attributs et méthodes que la classe *Solver*, mais à laquelle on indique le nombre de threads à lancer en parallèle. Cette classe génère donc, tout comme la classe *Solver*, une instance de la classe *Solution*. Cette instance se trouve cependant parmi les autres solutions retournées des différents threads dans un vecteur attribué. Ce vecteur est créé dans le constructeur de la classe et sa dimension (le nombre de solveurs qu'il contient) dépend de l'argument donné lors de la création d'une instance de cette classe. Chaque nouveau solveur est créé à partir d'une nouvelle copie du modèle en argument allouée dynamiquement à partir du constructeur copie de la classe *Model*.

Nous utilisons l'interface de programmation *OpenMP* pour lancer les threads à l'appel de la méthode *findsolution()*. Dès qu'une solution est trouvée, la variable *othersolverhasfinished* est mise à jour et les autres threads en exécution se terminent en retournant une solution possiblement invalide. Le solveur itère finalement sur le vecteur contenant les solutions retournées pour trouver une instance valide qu'il renvoie à l'appelant.

[3]L'ensemble du code et des exemples d'utilisation sont disponibles publiquement sur Github à l'adresse suivante : https://github.com/PMarcL/HydraSolver

Les solveurs lancés en parallèle effectuent la fouille en profondeur selon l'heuristique utilisée par le modèle associé à ces solveurs. Dépendamment des domaines des variables et des contraintes du modèle, certaines heuristiques ayant des aspects aléatoires performeront mieux que d'autres. Par exemple, pour les modèles ayant plusieurs variables avec des domaines différents et plusieurs contraintes différentes, utiliser une heuristique de choix structurel ou bien de choix de variable la plus contrainte en combinaison avec une heuristique stochastique serait plus avantageux que d'en utiliser une complètement aléatoire. Pour des modèles comme le carré magique, où toutes les variables ont le même domaine et sont sujettes aux mêmes contraintes, une heuristique de choix de variable aléatoire est adéquate. C'est ce type d'heuristique que nous utilisons pour notre solveur. Statistiquement, plus le nombre de solveurs en exécution est élevé, plus les chances d'instancier les variables optimales seront élevées et donc plus le temps de résolution sera rapide. Néanmoins, une limite majeure par rapport à la parallélisation des solveurs sur le CPU est le nombre de cœurs qu'il contient. Ceci implique que la différence de temps de résolution moyenne entre un très grand nombre de solveurs en parallèle et un unique solveur demeurera comparable à la différence moyenne entre un nombre de solveur de l'ordre du nombre de cœurs de processeurs et un unique solveur. En fait, utiliser un nombre de solveurs en parallèle trop élevé pourrait affecter négativement le temps de résolution dû à la concurrence de ceux-ci sur les CPU et la mémoire.

3.3 Parallélisation des algorithmes de filtrage avec le processeur graphique

3.3.1 Contrainte arithmétique binaire

Dans notre solveur, nous avons une classe *BinaryArithmeticConstraint* permettant d'imposer une contrainte de la forme v_1 *op* v_2 *relop* c, où v_1 et v_2 sont des variables entières, *op* est un opérateur arithmétique dans $\{+, -, \times, \div\}$, *relop* est un opérateur relationnel dans $\{=, \neq, \leq, <, \geq, >\}$ et c est une constante entière. Cette contrainte implémente deux algorithmes permettant de filtrer les valeurs des deux variables pour atteindre la cohérence d'intervalle et la cohérence de domaine. Chaque algorithme est implémenté sur le CPU et sur le GPU. Dans cette section, nous allons donc étudier ces implémentations.

L'algorithme permettant de filtrer les bornes des variables est très simple. Il vérifie simplement si chacune des variables respecte l'opération avec l'une des bornes de l'autre. La version implémentée côté CPU s'exécute en temps linéaire par rapport au domaine des deux variables. La version utilisant le processeur graphique est plus complexe à analyser. Tel que mentionné précédemment, pour utiliser efficacement le GPU, il faut représenter les données sous forme de matrices et de vecteurs et faire des opérations sur ceux-ci. L'algorithme 1 présente un exemple d'un programme exécuté sur le GPU pour une somme avec l'opérateur plus grand ou égal. Cette algorithme utilise donc le fait qu'ayant la borne inférieure d'un domaine énuméré et un décalage, il est possible de calculer une valeur du domaine. Si l'algorithme 1 est exécuté n fois en parallèle et si la cardinalité originale du domaine d'une variable est égale à n, il est donc possible d'utiliser l'identifiant d'un thread, via la variable $threadIdx.x$, pour calculer la valeur sur laquelle il va travailler et pour savoir à quel index du vecteur *result* il doit mettre le résultat. Il est donc possible de filtrer n valeurs d'une variable en parallèle avec un seul appel à l'algorithme 1. Maintenant, cet

Algorithme 1 Filtrage de borne GPU pour la contrainte arithmétique binaire

1: **procedure** FILTERBOUNDSSUMGEQGPU($v_1originalLowerBound, v_2lowerBound, v_2upperBound, c, result$)
2: **Input** : $v_1originalLowerBound$: borne inférieure originale de la première variable de l'opération.
3: $v_2lowerBound, v_2upperBound$: bornes inférieure et supérieure courante de la deuxième variable.
4: c : constante de l'opération.
5: $result$: pointeur sur le vecteur qui contiendra le résultat.
6:
7: $value \leftarrow v_1originalLowerBound + threadIdx.x$ ▷ Calcul de la valeur sur laquelle le thread va opérer.
8: $result[threadIdx.x] \leftarrow value + v_2lowerBound \geq c \,||\, value + v_2upperBound \geq c$

algorithme travail sur toutes les valeurs du domaine original d'une variable et le vecteur *result* contient un booléen par valeur pour indiquer si elle est présente dans le domaine filtré. Deux questions émergent de cette observation: ne serait-il pas plus efficace de travailler sur les valeurs présentent dans le domaines d'une variable et qu'arrive-t-il si le vecteur de bits original d'un *BitsetInVariable* contenait déjà des valeurs filtrées? La réponse à la première question est simple; il n'est pas plus coûteux de travailler sur plus de valeurs puisque les kernels CUDA sont exécutés en parallèle et il serait beaucoup plus compliqué d'essayer de gérer quelles valeurs sont traitées par le GPU plutôt que de simplement travailler sur l'ensemble des valeurs d'un domaine. Ensuite, pour gérer les valeurs précédemment filtrées dans le vecteur de bits, nous avons dû faire une méthode qui fait une conjonction entre le vecteur *result* et le vecteur de bits d'une variable. Cette méthode doit s'exécuter sur le CPU et s'exécute en temps linéaire par

rapport au domaine de la variable. Nous comparerons les performances de cette approche par rapport à l'algorithme exécuté sur le CPU dans la section 5.

Étudions maintenant l'algorithme permettant d'atteindre la cohérence de domaine pour cette contrainte. Il vérifie si pour chaque valeur du domaine d'une des deux variables, il existe un support dans les valeurs du domaine de l'autre variable. Donc, pour deux variables ayant chacune un domaine de n valeurs, l'algorithme s'exécute en pire cas en $O(n^2)$. L'algorithme utilisant le GPU que nous avons conçu pour régler ce problème s'exécute en $\Theta(n)$ en tout temps. L'idée générale de cet algorithme est de construire une matrice de $n \times m$, où n est la cardinalité de la première variable et m est la cardinalité de la seconde, et de mettre le résultat de la conjonction entre les vecteurs de bits des deux variable à des index précis et l'évaluation de l'opération pour les valeurs correspondantes dans chacune des cellules de la matrice (1 pour indiquer que l'opération est satisfiable, 0 sinon). Ensuite, il suffit de faire la somme de chacune des lignes et mettre le résultat dans un vecteur et faire de même pour les colonnes pour obtenir les résultats du filtrage pour les deux variables. Finalement, il reste simplement à mettre à jour les vecteur de bits des variables côté CPU. L'algorithme 2 montre un exemple du programme exécuté sur le processeur graphique pour mettre les résultats des opérations dans chacune des cellules de la matrice. Ici, nous tirons grandement avantage de l'architecture des GPUs pour faire beaucoup de calculs. Premièrement, nous utilisons encore une fois l'identifiant d'un thread et les bornes inférieures des variables pour calculer les valeurs sur lesquels l'instance va travailler. Remarquez ici que contrairement à l'algorithme 1, ce kernel est exécuté sur un bloc de $n \times m$ threads. Ainsi, chaque instance du programme a un identifiant en x ($threadIdx.x$) et en y ($threadIdx.y$) et il nous est possible de récupérer les dimensions du bloc en utilisant les attributs x et y de $blockDim$. Nous utilisons donc ces informations pour calculer l'index de la cellule sur laquelle le kernel va travailler et nous mettons le résultat de l'évaluation dans celle-ci.

Algorithme 2 Filtrage de domaine GPU pour la contrainte arithmétique binaire

procedure FILTERDOMAINSUMGEQGPU($v_1lb, v_2lb, v_1bitset, v_2bitset, c, matrix$)
  Input : v_1lb, v_2lb : bornes inférieures originales des deux variables.
    $v_1bitset, v_2bitset$: vecteurs de bits des deux variables.
    c : constante de l'opération.
    $matrix$: pointeur sur la matrice qui contiendra le résultat.

  $row \leftarrow blockIdx.y \times blockDim.y + threadIdx.y$
  $col \leftarrow blockIdx.x \times blockDim.x + threadIdx.x$
  $v_1Value \leftarrow v_1lb + threadIdx.y$
  $v_2Value \leftarrow v_2lb + threadIdx.x$
  $matrixIndex \leftarrow row \times blockDim.x + col$
  $matrix[matrixIndex] \leftarrow v_1bitset[row]$ && $v_2bitset[col]$ && $v_1Value + v_2Value \geq c$

L'algorithme 3 utilise la matrice générée précédemment par l'algorithme 2 et calcule la somme de chacune des lignes pour mettre le résultat final dans le vecteur $result$. Ici, nous exécutons ce programme sur un bloc de $1 \times n$ thread et chaque instance a la responsabilité de calculer la somme d'une ligne au complet. Un algorithme très similaire à ce dernier fait des opérations semblables pour calculer la somme des colonnes de la matrice.

Algorithm 3 Génération du vecteur de bits résultant sur le GPU

procedure SUMMATRIXROWS($matrix, rowSize, result$)
  Input : $matrix$: matrice générée par l'algorithme 2.
    $rowSize$: longueur d'une ligne de la matrice.
    $result$: pointeur sur le vecteur qui contiendra le résultat.

  $row \leftarrow threadIdx.x$
  $result[row] \leftarrow 0$
  for $i \in 0..rowSize - 1$ **do**
    $result[row] \leftarrow restult[row] + matrix[row \times rowSize + i]$

Suite à l'exécution de ces algorithmes, il reste simplement à mettre à jour les vecteurs de bits des deux variables côté CPU à partir des vecteurs correspondant aux sommes des lignes et colonnes de la matrice. Cette dernière opération s'exécute en temps linéaire par rapport aux domaines des deux variables. Analysons sommairement la

complexité de l'algorithme de filtrage dans son entièreté. Nous avons premièrement l'exécution de l'algorithme 2 qui se fait complètement en parallèle et en temps constant pour chaque instance. Ensuite, la somme des lignes et des colonnes par l'algorithme 3 et un algorithme similaire à ce dernier qui s'exécute respectivement en temps linéaire par rapport à n et à m. Finalement, l'opération pour mettre à jour les vecteurs de bits côté CPU s'exécute en temps linéaire par rapport à n et à m. Si nous considérons donc que $n = m$, nous avons que la complexité de cet algorithme de filtrage est $1 + 4n \in \Theta(n)$.

3.3.2 Contrainte de sommation

La contrainte de sommation permet de préciser qu'on veut que la somme d'un nombre arbitraire de variables soit égale à un nombre. Ce que nous cherchons à paralléliser en utilisant le GPU ici est le filtrage d'intervalle d'une telle contrainte. Nous avons nos variables X_i, $i = 1..n$ et notre contrainte est $\sum_{i=1}^{n} X_i = s$. Pour vérifier si $a \in dom(X_j)$ a un support d'intervalle pour notre contrainte somme, nous devons vérifier que

$$lowerBoundSum + a \geq s \text{ et } upperBoundSum + a \leq s. \tag{1}$$

sachant que $lowerBoundSum = \sum_{i=1, i\neq j}^{n} lowerBound(X_i)$ et que $upperBoundSum = \sum_{i=1, i\neq j}^{n} upperBound(X_i)$.

Le but ici est de paralléliser l'application de la cohérence d'intervalle à une seule variable X_j. Chaque thread exécuté sur le GPU vérifiera donc, avec la formule 1, si la valeur $a \in dom(X_j)$ qui lui est assignée possède un support d'intervalle par rapport aux autres variables dans la portée de notre contrainte somme.

Nous avons donc créé un kernel CUDA qui prend en paramètres : s, $lowerBoundSum$, $upperBoundSum$, $originalLowerBound$ (la borne inférieure originale de notre variable, pour être en mesure de calculer la valeur associée à un thread), et un pointeur vers le vecteur de bits dans la mémoire du GPU. La valeur associée à un thread sera

$$1024 \times blockIdx.x + threadIdx.x + originalLowerBound. \tag{2}$$

Ceci s'explique par le fait que nous voulons qu'un thread soit associé à chaque entrée du vecteur de bits; si celui-ci contient plus de 1024 valeurs, on atteint la taille maximale des blocs CUDA, et on doit lancer

$$k = \lceil (\text{nombre d'éléments dans le vecteur de bits, filtrés ou pas})/1024 \rceil$$

blocs qui filtreront 1024 valeurs chacun. Aussi, le premier élément de notre vecteur de bits correspond à la borne inférieure originale, d'où l'addition de $originalLowerBound$ à la fin. Sachant tout cela, chaque thread n'a qu'à calculer la valeur qu'il filtre avec la formule 2, utiliser cette valeur pour évaluer l'expression booléenne de la formule 1 et placer, pour terminer, le résultat de l'évaluation dans le vecteur de bits à l'élément $1024 \times blockId.x + threadId.x$. Après l'exécution des threads, il ne reste qu'à recopier le vecteur de bits de la mémoire GPU vers la mémoire CPU.

Nous pouvons, avec notre structure de données $bitsetIntVariable$, obtenir la borne supérieure et inférieure de nos variables en temps constant. Nous pouvons donc calculer $lowerBoundSum$ ainsi que $upperBoundSum$ avec $2(n-1)$ opérations. La copie du vecteur de bits du CPU vers le GPU se fait en $\Theta(m)$, où m est le nombre d'éléments initialement présents dans le domaine de la variable. L'exécution du kernel se fait en temps constant et il est lancé m fois. On peut exécuter p threads en parallèle, où p est le nombre de coeur CUDA dans notre GPU. La copie des données vers le CPU se fait en $\Theta(m)$. L'algorithme est donc en $\Theta(2(n-1) + m + m/p + m) = \Theta(n + m)$.

L'algorithme équivalent sur le CPU, avec lequel nous comparerons notre algorithme précédent dans la section 5, fonctionne de manière très similaire; au lieu de lancer un thread pour chaque valeur que nous voulons filtrer, l'algorithme itére séquentiellement sur chaque valeur présente dans le domaine de la variable filtrée.

On remarque une chose en commune avec le filtrage de la contrainte arithmétique binaire (§3.3.1) : l'algorithme travaille sur une valeur même si elle a déjà été filtrée et les raisons sont exactement les mêmes. Quoi qu'il en soit, en pratique, si un domaine a été en grande partie filtré et que sa cardinalité est petite, on ne veut pas utiliser le GPU car il va perdre son temps à travailler sur des valeurs déjà filtrées. Notre approche reste potentiellement intéressante pour filtrer de très grands domaines lorsque l'algorithme de recherche est des les premiers noeuds de l'arbre de recherche (i.e. le domaine des variables n'a pas été grandement filtré encore).

4 Protocole d'expérimentation

Pour tester les algorithmes présentés à la section précédente, nous allons utiliser les exemples d'utilisation que nous avons développé pendant la construction du solveur. De plus, puisque les algorithmes de filtrages risquent de performer différemment pour des très grands domaines, nous allons les tester individuellement sans utiliser un problème particulier.

Ainsi, la parallélisation de l'algorithme de fouille en profondeur sera testée avec le problème des n-dames et le problème du carré magique sans utiliser les algorithmes de filtrage exécutés sur le processeur graphique. Puisque nous utilisons une heuristique aléatoire pour s'assurer que les décisions varient entre les différents fils d'exécution, nous allons résoudre ces problèmes plusieurs fois avec et sans la parallélisation pour accumuler des statistiques sur le temps de résolution et le nombre de retours arrière dans les deux cas. De plus, puisqu'il nous est possible de facilement changer le nombre de fils d'exécution utilisé par notre solveur multi-agents, nous allons accumuler des statistiques pour différentes quantités de fil d'exécution pour résoudre un même problème.

Pour les algorithmes de filtrage, nous allons aussi comparer le temps de résolution moyen des problèmes mentionnés précédemment avec et sans l'utilisation du processeur graphique. Nous allons aussi étudier le temps d'exécution de leur algorithme de filtrage pour des domaines plus imposant que ceux utilisés dans le problème des n-dames et du carré magique. L'objectif de ceci est de voir l'évolution du temps d'exécution pour des domaines grandissant et voir s'il y a des points critiques où l'utilisation du processeur graphique devient préférable par rapport à simplement utiliser le processeur central ou vice versa. Les données enregistrées seront donc le temps d'exécution des appels aux algorithmes de filtrage et la cardinalité du domaine des variables à chaque appel.

5 Résultats et discussion

5.1 Performances du solveur avec plusieurs fils d'exécution

Nombre de threads	Temps de résolution (secondes)	Retours arrière	Redémarrages
1	3.71	5900.88	14.47
5	1.34	2027.72	12.77
10	1.14	1387.28	12.2

Table 1: Moyennes statistiques du solveur multi-agents pour le modèle du carré magique de dimension 6 (processeur i7 3.4ghz - 4coeurs)

Nombre de threads	Temps de résolution (secondes)	Retours arrière	Redémarrages
1	0.26	9.51	4.75
5	0.16	0.71	2.00
10	0.26	0.68	1.64

Table 2: Moyennes statistiques du solveur multi-agents pour le modèle des 25 reines (processeur i7 3.4ghz - 4coeurs)

Regardons d'abord les moyennes de temps de résolution du problème du carré magique de dimension six. Nous observons une amélioration de 63% par rapport au temps de résolution d'un solveur sur un thread lorsque nous utilisons un *MultiAgentSolver* de cinq threads et une amélioration de 15% par rapport à ce dernier lorsqu'un *MultiAgentSolver* de dix threads est utilisé. Ces résultats concordent avec nos hypothèses de probabilité de choix de variables plus optimales. En effet, nous avons beaucoup plus de chances de trouver un bon chemin rapidement si l'on démarre plusieurs solveurs en même temps. Pour ce qui est des moyennes du nombre de de retours arrière, elle est représentative des temps de résolutions. Nous observons que pour un *MultiAgentSolver* de cinq threads, le nombre moyen de retours arrière effectués pour la résolution du problème équivaut à 34% du nombre moyen pour un unique solveur, soit une amélioration de 66%. Nous pouvons déduire que c'est en grande partie cette amélioration qui permet le gain de performance de 63% mentionné plus tôt. Nous pouvons aussi conclure que le démarrage de plusieurs threads ne ralentit pas vraiment l'exécution, puisque l'amélioration en temps et en nombre de retours arrière est similaire.

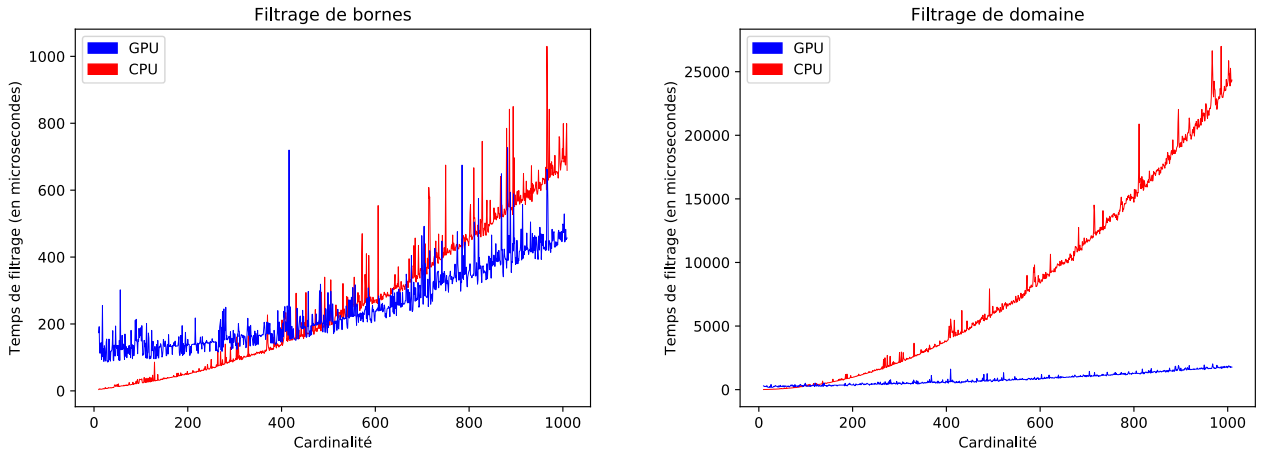
Nous ne pouvons cependant pas tirer les mêmes conclusions en regardant le problème des N-reines. Pour cinq threads, nous avons une nette amélioration du temps d'exécution, mais celui-ci remonte lorsque le nombre de threads est de dix. La ratio de diminution de retours arrière est aussi beaucoup plus grand dans les deux cas. Ceci s'explique par le fait que dans ce problème, nous avons énormément de contraintes binaires, ce qui cause beaucoup d'accès mémoire et qui peut générer un ralentissement si plusieurs threads s'exécutent en même temps. En regardant les redémarrages et les retours arrière, il n'y a pas de doute que les solutions trouvées sont plus rapides en augmentant le nombre de threads, dans les sens que l'on observe moins d'états différents, mais cela ne se traduit pas en gain de temps d'exécution. De plus, malgré le fait que les moyennes pour un thread et pour dix threads soient semblables,

leurs variances sont très différentes, avec 0.075 s^2 dans le premier cas et 0.002 s^2 dans le second. On peut aussi observer cette différence de variance très marquée dans le nombre de retour arrière (379 et 1.4[4]).

En bref, dans tout les cas, lorsque l'on utilise plusieurs threads, les solutions trouvées nécessitent moins d'étapes, et les différences sont assez significatives. De plus, cette mécanique nous protège plus efficacement des chemins malchanceux qui peuvent parfois ralentir le solveur. Par contre, certains types de contraintes nécessitant beaucoup d'accès mémoire peuvent causer un ralentissement, et il faut faire attention de choisir un nombre de threads adapté à la modélisation du problème à résoudre. Une règle du pouce ici serait de ne pas utiliser un nombre de thread plus élevé que le nombre de cœurs de l'ordinateur.

5.2 Performances des algorithmes de filtrage utilisant le processeur graphique

5.2.1 Contrainte arithmétique binaire



(a) (b)

Figure 2: Comparaison des performances de l'algorithme de filtrage de la contrainte arithmétique binaire exécuté sur le processeur central (Intel i7, 4 coeurs 3.4GHz) et sur le processeur graphique (NVidia GTX 960, 1024 cœurs CUDA).

Commençons par étudier le temps d'exécution de l'algorithme pour des domaines de cardinalité variant entre 10 et 1000[5]. La figure 2 présente deux graphiques comparant les performances de l'algorithme de filtrage de borne (2a) et de filtrage de domaine (2b). Dans les deux cas, l'algorithme exécuté sur le processeur graphique est initialement beaucoup plus lent que la version exécutée sur le processeur central. Cette différence est causée par le temps de latence pour transférer des variables entre la mémoire du CPU et celle du GPU. Même en minimisant la quantité d'information transférée entre les deux espaces mémoire, le délai est assez important pour que l'utilisation du processeur graphique pour des petits domaines dégrade les performances au point qu'il est grandement préférable de simplement utiliser l'algorithme exécuté sur le processeur central. Ceci dit, dans le cas du filtrage de domaine, l'algorithme exécuté sur le GPU devient rapidement beaucoup plus performant. En effet, nous avons précédemment mentionné que cette version s'exécute en temps linéaire par rapport au domaine des variables tandis que la version exécutée sur le CPU s'exécute en temps quadratique. Nous observons donc ici que l'algorithme devient plus performant lorsque les variables ont un domaine dont la cardinalité est plus grande qu'environ 100. Pour l'algorithme de filtrage d'intervalle, la version exécutée sur le processeur graphique devient un peu plus performante pour des domaines de cardinalité d'environ 500. Le gain de performance ici n'est pas vraiment significatif et nécessite des cardinalités assez grande que nous croyons que dans des situations réalistes, l'utilisation du processeur graphique n'est pas appropriée pour résoudre le problème.

Une chose importante à souligner ici est que lorsque les algorithmes exécutés sur le CPU sont utilisés par le solveur, leurs performances s'améliorent durant l'exploration de l'arbre de recherche. Puisque ceux-ci s'adaptent au contenu des domaines des variables à chaque appel, plus l'algorithme de fouille en profondeur s'approche d'une

[4]Cette énorme différence est dû à quelques valeurs aberrantes lorsque l'on exécute sur seulement un thread.

[5]La générations des données pour ce test a été fait en utilisant le projet *BanchmarkBinaryArithmeticConstraint* disponible dans le dépôt Git du projet.

Configuration	10-Dames	15-Dames	20-Dames
Cohérence de domaine, filtrage sur le CPU	0.013	0.075	0.299
Cohérence de domaine, filtrage sur le GPU	1.443	6.59	18.27
Cohérence d'intervalle, filtrage sur le CPU	0.007	0.035	0.1
Cohérence d'intervalle, filtrage sur le GPU	2.038	7.51	19.431

Table 3: Temps de résolution moyen (en secondes) du problème des n-dames pour différentes configurations du solveur.

feuille, plus le filtrage s'exécutera rapidement. Par contre, si le solveur utilise les algorithmes exécutées sur le processeur graphique, le temps de filtrage restera toujours le même. Ces algorithmes filtrent toujours l'entièreté des domaines de leurs variables sans tenir compte du contenu réel de ceux-ci. Donc, même si la majorité des variables sont instanciées, le filtrage s'exécutera comme si les domaines étaient pleins. Ce que nous cherchons à souligner ici est que même s'il est possible d'avoir des gains de performances pour des domaines imposant en utilisant le processeur graphique, il est possible que les performances réel du solveur soient beaucoup moins bonnes par rapport à l'utilisation des algorithmes de filtrage sur le CPU. Donc, pour utiliser efficacement le processeur graphique, il faudrait développer une heuristique qui change l'algorithme de filtrage dynamiquement durant la recherche. Ainsi, il serait possible de tirer avantage des performances du GPU quand les domaines sont très grands et utiliser les algorithmes exécutés sur le CPU lorsque les domaines sont plus petit qu'un certain seuil.

Regardons les performances du solveur lors de la résolution du problème des n-dames avec et sans l'utilisation des algorithmes exécutés sur le processeur graphique. Ce problème a été modélisé en utilisant seulement des instances de la contrainte arithmétique binaire. Ainsi, le filtrage est soit complètement effectué sur le processeur central ou complètement sur le processeur graphique. Les statistiques ici ont été générées pour 100 résolutions du problème et pour 10, 15 et 20 dames pour chaque configuration. Le tableau 3 présente donc les temps de résolution moyen pour chaque configuration[6]. On observe clairement les conséquences des observations que nous avons faites précédemment. L'utilisation du processeur graphique ralentit significativement la résolution du problème pour ces instances. Pour avoir un gain de performance, nous croyons qu'il faudrait avoir des domaines significativement plus grands et utiliser l'heuristique discutée précédemment.

5.2.2 Contrainte de sommation

Nous avons tout d'abord comparé la performance du filtrage d'intervalle sur le GPU et le CPU ; pour cela, nous avons pris une contrainte jouet :

$$X_1 + X_2 + X_3 = \lfloor n/2 \rfloor$$

$$dom(X_i) = \{0..n\}, i = 1, 2, 3$$

et nous y avons appliqué l'algorithme de filtrage sur le CPU et ensuite sur le GPU. Nous avons fait varier n entre 100 et 200 000 par bonds de 5, pour voir la progression du temps d'exécution du filtrage au fur et à mesure qu'on augmente la taille du domaine des variables. Les résultats de cette expérience sont dans la figure 3. Nous observons donc qu'avec une cardinalité entre 0 et 50 000, l'algorithme exécuté sur le CPU domine clairement celui exécuté sur le GPU. Par contre, avec une cardinalité plus grande que 100 000, le GPU est nettement plus performant que le CPU pour filtrer la contrainte. Ces résultats, bien qu'intéressants, sont un peu décevants. Peu de problème vont, en pratique, nécessiter le filtrage d'aussi grands domaines. Le cas réel avec lequel nous avons expérimenté est le carré magique, et la plus grande instance que nous pouvons résoudre en temps raisonnable est d'ordre 12, ce qui génère des domaines de variables avec une cardinalité initiale de 144 pour les contraintes de somme. En conséquence, nul besoin de souligner que l'utilisation du GPU pour résoudre le carré magique retournait une solution beaucoup plus lentement que si nous avions simplement utilisé le CPU (les données ne sont pas présentées). Beaucoup des points de discussion sont communs avec le filtrage de la contrainte arithmétique binaire : par exemple, le CPU possède le même avantage lorsqu'on progresse dans l'arbre de recherche et que les domaines des variables commencent à être majoritairement filtrées. Telle que mentionnée précédemment, une heuristique choisissant judicieusement l'algorithme de filtrage à utiliser serait de mise ici.

[6]Le nombre de retour arrière et de redémarrage moyen étant presque identique dans tous les cas, nous avons décidé de seulement étudier le temps de résolution.

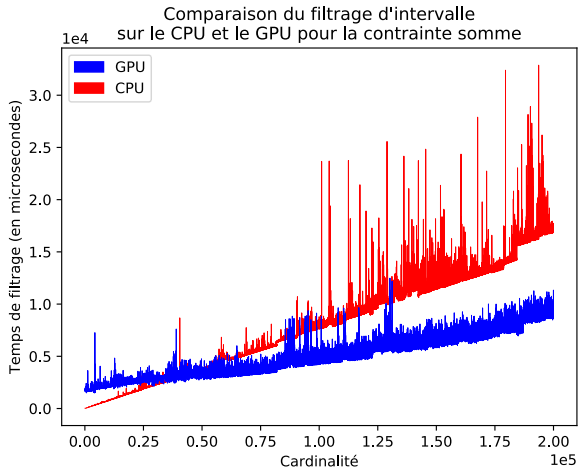


Figure 3: Résultat pour le filtrage de la contrainte somme, CPU : Intel Core i7, 4 coeurs à 3.5GHz. GPU : Nvidia Geforce 860M, 640 coeurs CUDA.

6 Conclusion

Bien que nous n'ayons pas réussi à améliorer significativement les performances de notre solveur avec le processeur graphique, nous tirons des conclusions intéressantes de notre expérience. Premièrement, la construction d'un solveur de base est assez facile et se fait relativement rapidement. Le défi est plutôt dans l'implémentation des algorithmes de filtrage et dans la gestion efficace des domaines des variables. Notre solveur n'est pas très efficace à plusieurs égards, mais il nous en appris beaucoup sur les enjeux derrière la construction d'un tel logiciel. Nous sommes tout de même fier de la flexibilité de celui-ci puisqu'il nous a permis de facilement tester plusieurs cas de figure en changeant que quelques options dans le modèle (filtrer sur GPU ou CPU, cohérence de domaine ou d'intervalle, etc.).

Les processeurs graphiques modernes sont effectivement très rapides, mais si on les utilise pour traiter des données trop petites, le temps de latence pour transférer des données dans leur mémoire est trop important pour que leurs performances se fassent sentir. Dans le cas des solveurs de contraintes, il faudrait avoir des variables assez massives pour que leur utilisation en vaille la peine et il faudrait très certainement développer une heuristique pour changer dynamiquement les algorithmes de filtrage quand les domaines se vident. Ceci dit, si le modèle contient effectivement des domaines énormes, l'utilisation du processeur graphique pourrait grandement améliorer les performances du solveur. Une idée intéressante dans une telle situation serait d'utiliser le processeur graphique pour atteindre la cohérence de singleton dans les premiers nœuds de l'arbre de recherche et utiliser le processeur central pour le reste de la fouille en profondeur. Il reste donc beaucoup d'idées à explorer quant à l'utilisation de cette technologie dans les solveurs de contraintes.

Quant à la parallélisation au niveau du CPU, les résultats furent impressionnants. Le simple fait de lancer plusieurs solveurs prenant des décisions de branchement aléatoires et de retourner le résultat du premier atteignant une solution a mené à une accélération significative par rapport à notre solveur séquentiel. La même idée a été poussée encore plus loin dans Choco : son solveur parallèle lance plusieurs solveurs séquentiels utilisant des heuristiques différentes. Ces différents solveurs peuvent ensuite se partager des informations, par exemple les *nogoods* appris, pour accélérer d'avantage leur recherche parallèle. L'implémentation de différentes heuristiques de branchement et un système de partage d'information entre les solveurs seraient des améliorations potentielles à apporter à notre solveur parallèle.