Rapport Final IFT3150

Hamdi Ghannem
Sous la supervision de
M.Fabian Bastin

29 avril 2022

1 Introduction

Le but du projet est d'explorer l'utilisation de techniques avancées de programmation en Julia dans la perspective de mettre celles-ci en application dans divers projets de recherche opérationnelle, de recherche menés en parallèle.

L'optimisation linéaire sera d'abord introduit, puis l'algorithme du simplexe qui a été implémenté tout au long du projet sera présenté.

En optimisation mathématique, un problème d'optimisation linéaire demande de minimiser une fonction linéaire sur un polyèdre convexe. Cette fonction qu'on minimise, ainsi que les contraintes du problème sont décrites par des fonctions linéaires.

$$\begin{gathered} \min \ z = c^T x \\ \text{tq} \ \ \mathrm{Ax} \leq b \\ \mathrm{x} \geq 0 \end{gathered}$$

où $x \in R^n$ est l'inconnu, le vecteur des variables réelles à optimiser, et les données qui décrivent le problème sont des vecteurs $c \in R^n$ et $b \in R^m$ et une matrice $A \in R^{m \cdot n}$

Le théorème fondamental de la programmation linéaire nous indique que les extremums de notre fonction linéaire sur la région du polyèdre convexe existe dans les points extrêmes de notre région (soit les coins de notre région)

L'algorithme du simplexe est alors un algorithme de résolution de ces problèmes d'optimisation linéaire qui nous permet de traverser intelligemment d'un point extrême a un autre, en améliorant notre valeur objective à chaque itération de l'algorithme.

2 Problématique du projet

Au début du projet, on a voulu explorer la programmation en GPU, en implémentant l'algorithme du simplexe avec la librairie de CUDA sur Julia. CUDA est une technologie de GPGPU (General-Purpose Computing on GPU), qui utilise un processeur graphique pour exécuter des calculs généraux à la place du CPU, développé par NVIDIA. L'aspect fort de la programmation en GPU est en fait la possibilité d'exploiter des coeurs CUDA (un maximum de 1024 coeurs en Julia) pour paralléliser le calcul et réduire possiblement le temps de calcul des opérations comme les multiplications matricielles, les produits scalaires, les additions et soustractions etc...

Le premier mois était une phase d'exploration, où j'apprenais comment la programmation GPU fonctionne réellement. J'ai regardé plusieurs ressources en ligne comme par exemple "GPU programming in Julia Workshop - JuliaCon 2021" ou la documentation de CUDA.jl en ligne ici.

J'ai alors commencé à tester des implémentations de CuArray (les Arrays en CUDA) pour voir la difference de performance entre le CuArray et un Array CPU classique. Je me suis rendu compte qu'il y a beaucoup de problèmes par rapport aux CuArrays, par exemple la librairie LinearAlgebra.jl de Julia ne fonctionne pas totalement avec les CuArrays, il manque beaucoup de fonctionnalités, dont la définition de la matrice identité en CUDA.

La tâche était alors loin d'être simple, puisque je devais réimplémenter beaucoup de fonctionnalités qui existent exclusivement sur CPU.

Plusieurs fonctionnalités à coder se trouvaient dans le projet initial énoncé en janvier. Entre autre, une des fonctionnalités que nous voulions ajouter était d'interfacer notre solveur du simplexe en CUDA avec la librairie Math Optimisation Interface (MOI) et la librairie JuMP (Julia Mathematica oPtimisation), ce qui permet à l'utilisateur de créer un problème linéaire rigoureusement et de faciliter les benchmarks avec d'autres solveurs. Après un mois d'exploration sur les différentes parties du projet, Monsieur Bastin et moi nous sommes rendus compte de la taille réelle du projet et que l'implémentation des deux tâches étaient impossibles en moins de 4 mois. Alors, nous avons dû abandonnés l'idée d'interfacer notre solveur avec les librairies JuMP et MOI, et avons priorisé l'implémentation de l'algorithme du simplexe avec CUDA, ce qui nous a permis également d'avoir une idée de comment la programmtion en GPU peut être utilisée.

Après plusieurs tests et recherches faites, je me suis rendu compte que si je veux paralléliser mes calculs sur CUDA, il faut que les opérations de pivots soient thread-safe. Or, en explorant les notions des CuArrays, ceci n'est pas le cas. Je me suis rendu compte aussi que pour avoir accès aux éléments de mon CuArray, il faut procéder bloc par bloc de mémoire, au lieu de pouvoir avoir

accès à un seul élément de mon tableau. Ceci veut dire que les for loops où je vais avancer élément par élément ne marchent plus et que je dois trouver une nouvelle manière pour faire mes fonctions. Or, Julia permets de vectoriser une fonction, c'est-à-dire appliquer une fonction élément par élément sur un vecteur, et CUDA va automatiquement paralléliser la fonction sur son CuArray, d'où la raison pour laquelle l'opérateur du broadcasting '.' en Julia apparaît partout dans les codes qui utilisent CUDA.

Par exemple, si on a un vecteur de Int, puis on veut ajouter 1 à chaque élément de notre vecteur, la manière classique de le faire sur CPU est:

```
for i in 1:length(v)
    v[i]+=1
end
```

Alors qu'en CUDA, on utilise l'opérateur '.', par exemple:

```
v.+=1
```

Ceci nous ramène vers la partie test de CUDA, où j'ai essayé de voir s'il y avait une différence de performance entre GPU et CPU.

3 Benchmarks sur CUDA

Pour vraiment bénéficier du gain de performance, la librairie de CUDA recommande d'utiliser des Float32 au lieu de Float64. Alors les benchmarks que j'ai fait ici sont exclusivement en Float32 et on va utiliser une matrice de taille $1000 \cdot 1000$

3.1 La multiplication

Pour le premier benchmark le plus important, c'est la multiplication entre matrices. On va prendre deux matrices générés au hasard.

```
In [20]: @benchmark CUDA.@sync x*x #CUDA

Out[20]: BenchmarkTools.Trial: 10000 samples with 1 evaluation.
          Range (min … max):  198.100 µs …  28.577 ms  ┊ GC (min … max): 0.00% … 0.00%
          Time  (median):     251.900 µs               ┊ GC (median):    0.00%
          Time  (mean ± σ):   301.067 µs ± 620.489 µs  ┊ GC (mean ± σ):  0.46% ± 1.02%

          198 µs        Histogram: log(frequency) by time       1.22 ms <

          Memory estimate: 672 bytes, allocs estimate: 33.
```

```
In [24]: @benchmark y*y #CPU

Out[24]: BenchmarkTools.Trial: 954 samples with 1 evaluation.
          Range (min … max):  3.481 ms … 9.114 ms  ┊ GC (min … max): 0.00% … 38.37%
          Time  (median):     5.289 ms              ┊ GC (median):    0.00%
          Time  (mean ± σ):   5.218 ms ± 1.079 ms  ┊ GC (mean ± σ):  3.99% ±  9.24%

          3.48 ms        Histogram: frequency by time       8.79 ms <

          Memory estimate: 3.81 MiB, allocs estimate: 2.
```

On peut observer une amélioration de plus de 20 fois pour la multiplication en CUDA.

Ceci me donne alors l'idée d'exploiter le caractère matriciel d'une autre implémentation du simplexe, qui est le simplexe revisé. C'est une forme du simplexe, mais tabulaire, où au lieu de faire les calculs sur tout le tableau du simplexe, on utilise plusieurs matrices plus petites et on fait des multiplications de matrices pour avoir accès aux données dont j'ai besoin. Je propose alors à Monsieur Bastin de changer l'implémentation du simplexe standard, vers le simplexe revisé, pour avoir une utilisation plus intelligente et efficace de l'algorithme, surtout avec le contexte des calculs matriciels sur GPU.

3.2 Trouver la variable sortante dans le simplexe

L'algorithme est simple, si on prends 2 vecteurs A et B, on voudrait trouver l'argmin de la division de A par B élément par élément, en ignorant les valeurs négatives. J'ai fait deux implémentations du code en CUDA. Voici les benchmarks:

```
In [30]: @benchmark CUDA.@sync findExitingVar(x,y) #CUDA
```

```
┌ Warning: Performing scalar indexing on task Task (runnable) @0x0000000082ed8da0.
│ Invocation of getindex resulted in scalar indexing of a GPU array.
│ This is typically caused by calling an iterating implementation of a method.
│ Such implementations *do not* execute on the GPU, but very slowly on the CPU,
│ and therefore are only permitted from the REPL for prototyping purposes.
│ If you did intend to index this array, annotate the caller with @allowscalar.
└ @ GPUArrays C:\Users\hamdi\.julia\packages\GPUArrays\umZob\src\host\indexing.jl:56
```

```
Out[30]: BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):  193.000 μs …  42.952 ms  ┊ GC (min … max): 0.00% … 30.54%
 Time  (median):     221.000 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   265.399 μs ± 805.529 μs  ┊ GC (mean ± σ):  1.40% ±  0.51%

  193 μs        Histogram: log(frequency) by time        962 μs <

 Memory estimate: 17.23 KiB, allocs estimate: 308.
```

```
In [32]: @benchmark CUDA.@sync findExitingVar2(x,y) #CUDA
```

```
Out[32]: BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):  58.500 μs …  53.258 ms  ┊ GC (min … max): 0.00% … 32.46%
 Time  (median):     71.800 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   93.354 μs ± 538.915 μs  ┊ GC (mean ± σ):  1.85% ±  0.32%

  58.5 μs         Histogram: frequency by time          256 μs <

 Memory estimate: 6.39 KiB, allocs estimate: 116.
```

```
In [34]: @benchmark findExitingVar(X,Y) #CPU
```

```
Out[34]: BenchmarkTools.Trial: 10000 samples with 8 evaluations.
 Range (min … max):  3.475 μs … 578.712 μs  ┊ GC (min … max):  0.00% … 98.69%
 Time  (median):     5.312 μs               ┊ GC (median):     0.00%
 Time  (mean ± σ):   7.308 μs ±  20.430 μs  ┊ GC (mean ± σ):  18.15% ±  6.55%

  3.48 μs      Histogram: log(frequency) by time      20.6 μs <

 Memory estimate: 35.62 KiB, allocs estimate: 3.
```

```
In [39]: @benchmark findExitingVar2(X,Y) #CPU
```

```
Out[39]: BenchmarkTools.Trial: 10000 samples with 10 evaluations.
 Range (min … max):  1.670 μs … 313.360 μs  ┊ GC (min … max):  0.00% … 98.43%
 Time  (median):     1.850 μs               ┊ GC (median):     0.00%
 Time  (mean ± σ):   2.435 μs ±   9.556 μs  ┊ GC (mean ± σ):  13.90% ±  3.54%

  1.67 μs      Histogram: log(frequency) by time      5.68 μs <

 Memory estimate: 7.95 KiB, allocs estimate: 2.
```

```
In [26]: function findExitingVar(Acol,b)
             x=filter(x->x[1]>0,Pair.(Acol./b,1:size(b,1)))
             last(x[argmin(first.(x))])
         end

Out[26]: findExitingVar (generic function with 1 method)

In [38]: function findExitingVar2(Acol,b)
             pos(x) = x ≤ 0 ? Inf : x
             return argmin(pos.(b./Acol))
         end
```

Ce qu'on peut remarquer avec ce benchmark, c'est que :

3.2.1 CuArray Indexing

La première implémentation de l'algorithme pour trouver la variable sortante utilise une for loop implicite, où je filtre mon CuArray, et puisque le temps d'accès au GPU élément par élément est beaucoup trop lent, il y a une grosse différence entre la deuxième implémentation qui utilise uniquement de la vectorisation, et la premiere (150 microsecondes)

3.2.2 Mixed-GPU-CPU

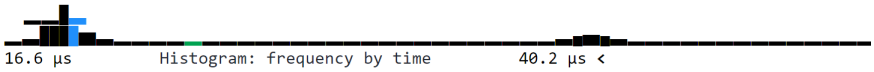
On remarque aussi, à notre grande surprise, que cette fois ci, le CPU est vraiment beaucoup plus rapide que le GPU. On peut conclure alors que le GPU n'est pas toujours plus rapide que le CPU. Au contraire, ceci nous donne une toute nouvelle idée pour l'implémentation de l'algorithme du simplexe révisé, où on peut utiliser un mix entre des calculs faits sur GPU (par exemple la multiplication) et des calculs fait sur CPU (comme trouver la variable sortante dans l'algorithme du simplexe). Il reste que si on prends deux vecteurs de grande taille=1000000, le calcul avec CUDA est beaucoup plus rapide (145 microsecondes en CUDA contre 3 millisecondes en CPU)

3.3 Conversion GPU vers CPU

Le dernier benchmark nous donne une idée pour utiliser un mix entre GPU et CPU, mais est-ce que la conversion entre un Array sur CPU vers un CuArray est coûteuse? Voici ce que nous indique le benchmark

```
In [43]: @benchmark CUDA.@sync CuArray(X)
```

```
Out[43]: BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):  16.600 μs … 127.400 μs  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     18.500 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   21.162 μs ±   6.213 μs  ┊ GC (mean ± σ):  0.00% ± 0.00%
```

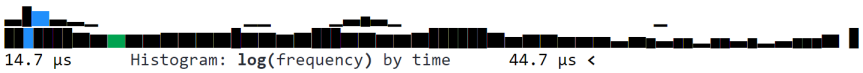


```
 16.6 μs          Histogram: frequency by time         40.2 μs <

 Memory estimate: 224 bytes, allocs estimate: 7.
```

```
In [44]: @benchmark Array(x)
```

```
Out[44]: BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):  14.700 μs …  2.764 ms  ┊ GC (min … max): 0.00% … 97.73%
 Time  (median):     15.800 μs              ┊ GC (median):    0.00%
 Time  (mean ± σ):   19.287 μs ± 28.443 μs  ┊ GC (mean ± σ):  1.40% ±  0.98%
```



```
 14.7 μs       Histogram: log(frequency) by time      44.7 μs <

 Memory estimate: 7.95 KiB, allocs estimate: 2.
```

On peut voir alors que des fois, ça vaut la peine de convertir notre CuArray vers un Array sur CPU pour faire un calcul qui nous prends 1.5 microsecondes au lieu de 70 microsecondes.

4 Problèmes rencontrés lors de l'implémentation du projet

4.1 Position réelle de la colonne entrante dans la base

Dans le simplexe standard, on a un vecteur des coûts réduits, pour trouver la position de la colonne entrante dans la base, il faut juste trouver la position (argmin) du coût réduit le plus négatif. Ainsi on aurait la position exacte de la colonne entrante, puisque c'est un seul vecteur.

Pour le simplexe révisé, On a deux vecteurs des coûts réduits, on a le premier vecteur CbinvA que l'on calcule avec $C_b^T B_{inv} A$ (avec C_b le vecteur des coûts réduits initiaux des colonnes qui correspondent à la base actuelle, B_{inv} l'inverse de la matrice de base initiale trouvé après l'initialisation du problème, qui est automatiquement updatée à chaque fois qu'on pivote notre matrice en faisant rentrer une nouvelle variable de base à la place d'une ancienne, et A la matrice correspondante aux colonnes hors base initiales). Ce vecteur corresponds aux coûts réduits des colonnes hors base initiales mais updatée à chaque pivot. Le deuxieme vecteur Cbinv que l'on calcule avec $C_b^T B_{inv}$ (mêmes définitions juste avant). Ce vecteur corresponds aux coûts réduits des colonnes de la base initiale updatée a chaque pivot. Ceci veut dire que pour trouver la position de la colonne entrante dans la base, il faut calculer le minimum des deux vecteurs, puis voir le minimum des deux.

Le problème avec cette approche c'est qu'on va se retrouver avec une position qui est relative aux colonnes qui sont dans la base ou les colonnes qui sont hors de la base, mais pas une position absolue. Par exemple si je prends le premier cas où j'utilise l'implementation du simplexe standard et que j'ai un vecteur de coûts réduits égal à [-1.5,-1,2,-3,-1.2] (avec les colonnes de base initiale bidx=[1,5]). Je peux facilement dire que la position de la colonne entrante est égal à 4 puisque -3 est le coût réduit le plus négatif.

Or dans le simplexe révisé, j'ai le vecteur CbinvA=[-1,2,-3] et le vecteur Cbinv=[-1.5,-1.2], je prends alors le minimum entre (-3,-1.5) et j'ai -3 qui est à la position 3 du CbinvA. Je ne sais pas exactement à quelle colonne ceci corresponds. Alors je me retrouve coincée avec une position relative sans savoir où se trouve réellement cette colonne.

La solution que j'ai trouvé, c'est de faire l'initialisation du simplexe révisé avec le simplexe standard, ou je détecte si j'ai des variables isolées et faisables que j'ajoute directement dans ma base. Si mon problème n'est pas canonique alors, je dois ajouter $n-m$ variables artificielles (avec n le nombre de contraintes, m le nombre de variables isolées et faisables que j'ai détecté). Ceci est appelé la phase une du simplexe. Je pivote le nouveau problème généré jusqu'à que je trouve une base réalisable. C'est alors la que j'enlève mes variables artificielles,

et c'est la où je peux créer un BitVector en Julia, qui me permets de savoir où se trouve mes colonnes de base initiales et mes colonnes hors base initiales.

En utilisant cette ligne en Julia:

```
s.bidx[leaving]=findall(s.access.xor(t))[entering]
```

Avec access, le BitVector qui contient les informations necessaires, t est la condition pour savoir si le minimum se trouve dans CbinvA ou Cbinv, et la fonction findall qui permets de retrouver la vraie position de la colonne entrante pour la rajouter dans bidx

4.2 Variables artificielles encore dans la base après la phase une

Un autre problème que j'ai rencontré, c'est trouver des variables artificielles dans bidx, alors que j'ai enlevé toutes les variables artificielles. Ceci est un signe que la base choisie est dégénérée. Une base dégénéré veut dire qu'il y a des variables de base qui ont une valeur égale à 0. J'aurais alors un bug dans la définition de mon BitVector, puisqu'il y aurait une erreur d'out of bounds.

L'algorithme pour résoudre ce problème est comme suit: Si le problème artificiel a été résolu et la valeur optimale est nulle, Je vais identifier une variable artificielle qui est une variable de base, S'il n'y en a aucune, je sors de la phase une.

Sinon, je vais pivoter la variable artificielle avec une variable hors-base de coût réduit nul (peu importe laquelle, on peut juste prendre celle avec le plus petit indice). Je répète jusqu'à ce que l'on ait plus de variables artificielles dans la base.

Si le problème artificiel a été résolu et que la valeur optimale est non nulle, alors le problème n'est pas réalisable.

5 Conclusion et vision du futur

Ce projet a été un véhicule pour apprendre la programmation sur GPU avec CUDA sur Julia et j'ai adoré travailler la dessus! J'ai été surpris par des imprévus bouleversants qui m'ont certes ralenti et j'ai appris à persévérer malgré ces obstacles, et j'en ai appris énormément sur la programmation en CUDA et les détails derrière tout ça.

Concernant le futur de ce projet, l'implémentation du code du simplexe révisé en CUDA peut être ajoutée dans la librairie de Jasmin.jl, fait par Jean Laprés-Chartrand.

Unes des améliorations au code serait d'implémenter des CuSparseArrays ou les matrices creuses en CUDA pour exploiter le coté creux de l'algorithme du simplexe.

Une autre amélioration serait d'implémenter le presolve et la phase une en GPU au lieu du CPU, et même implémenter le simplexe dual, et le simplexe primal dual en GPU pour voir la différence avec le simplexe révisé.

Un code qui génère un problème d'optimisation linéaire assez grand serait un bon projet futur, pour qu'on puisse aisément tester le code implémenté dans ce projet, et même dans d'autres projets d'optimisation linéaire.

Je tiens à remercier M.Fabian Bastin pour m'avoir fait confiance et supervisé ce projet de recherche.