Systèmes distribués et grilles : TP CUDA

Thibaut EHLINGER, Benjamin HERB *Université de Strasbourg*

3 octobre 2016

Dans ce TP nous devions exécuter des produits matriciels, qui sont des opérations mathématiques très gourmandes, avec des complexité algorithmique de l'ordre de $\mathcal{O}(n^3)$. Afin de mettre pleinement à disposition les capacités de nos machines, nous avons donc exécuté les calculs cette fois sur la carte graphique de nos ordinateurs, grâce à la technologie CUDA (Compute Unified Device Architecture). Comme pour le TP précédent, il s'agit d'un produit de matrices 4096×4096. Il est à noter que sur les cartes graphiques des machines de la salle J4 , des **gtx680, architecture Kepler**, nous n'avions pas le droit de lancer des calculs trop longs. Pour des raisons de sécurité, sans doute, il nous était impossible de réaliser des calculs dépassant les 11-12 secondes. Néanmoins cela ne nous a pas empêché d'obtenir des résultats concluants, étant donné que de toute manière nous cherchions à optimiser le temps de calcul.

Dans ce rapport, nous expliquerons tout d'abord les problématiques de dimensionnement des blocs de *threads* et leurs conséquences. Puis, nous étudierons les différents temps et puissances de calculs obtenus en fonction du dimensionnement des blocs, sur le kernel 2. Enfin, nous traiterons les expérimentations effectuées sur le kernel 4, en nous concentrant sur la mémoire partagée : comment l'utiliser au mieux et les bienfaits de son utilisation.

1 Contexte

1.1 Environnement d'expérimentation

Comme dit précédemment, nous travaillons ici sur une carte graphique **gtx680, architecture Kepler**. Les données ont été prélevées une seule fois, puis notées sur un fichier `Excel` prévu à cet effet. Ensuite, nous avons converti ces donnés au format `csv`, qui ont ensuite été représentées graphiquement avec `Gnuplot`. Parallèlement, nous avons aussi intégré dans ce rapport un graphique qui était déjà généré dans la feuille `Excel`.

1.2 Des dilemmes de pavage

L'intérêt du calcul distribué sur une carte graphique est de mettre à profit la puissance de calcul d'une GPU. En revanche, les GPU sont optimisées pour les calculs dits *SIMD*, pour *single instruction - multiple data*, ce qui veut dire qu'il est très simple pour une carte graphique de réaliser moult fois la même instruction sur un grand jeu de données diversifiées. Le calcul matriciel correspond exactement à ce genre de calculs.

Une GPU s'organise en blocs de *threads*, aussi appelés *warps*. Chaque *thread* est en charge d'une toute partie du calcul. En effet ces *threads* ne sont pas faits pour se comporter comme des *threads* POSIX : ils sont conçus pour réaliser des calculs très très simples. Ici, par exemple, un *thread* est en charge d'une seule et unique case de la matrice résultat à la fois. Ce qui veut dire qu'il gère la somme de $Nb_{colonnes}(\mathcal{A}) \times (Nb_{lignes}\mathcal{B})$ produits de `double`.

Une première contrainte est que le nombre de *threads* par bloc ne peut dépasser 1024. On est aussi limité pour le nombre total de blocs, mais cette limite ne nous a pas concerné pour notre TP, celle-ci allant de 2^{16} à $2^{31} - 1$. Le dilemme mis en exergue dans ce TP concerne le dimensionnement de ces blocs : en effet, on peut subdiviser les matrices à multiplier entre elles en blocs de différentes tailles, ce qui a différentes conséquences. Les questions auxquelles nous essaierons de répondre sont :

— Sur une seule dimension, quelle semble être la largeur optimale ?
— Sur deux dimensions, quel(s) facteur(s) influe le plus sur les différentes puissances/durées de calcul ?

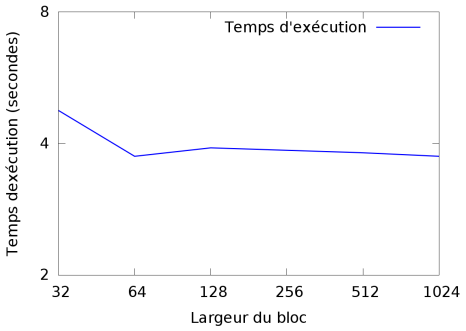
2 Allocation de blocs d'une seule dimension

2.1 Analyse relative

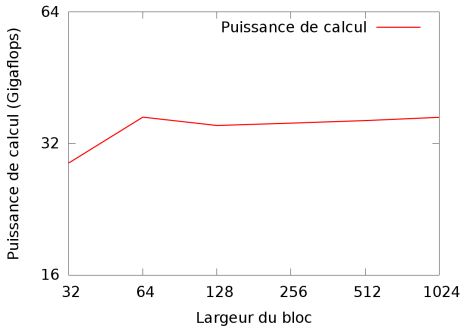
La figure 1 présente, à gauche, le temps d'exécution du produit en fonction de la largeur du bloc, et à droite, la la puissance d'exécution en fonction de la largeur du bloc.

Dans l'étude **relative** des résultats, c'est à dire lorsqu'on les compare entre eux, deux choses importantes sont à notes :

— Le seul écart notable est entre 32 et 64 *threads* de largeur pour le bloc. Comme mentionné en cours, les meilleurs résultats sont souvent obtenus entre 32 et 64 threads par bloc, ici la bonne réponse est 64.
— 64 n'est que légèrement meilleur que les valeurs qui lui sont supérieures. Néanmoins 64 est clairement la meilleure valeur. C'est le meilleur compromis entre le nombre de blocs et la taille des blocs. Au-delà, les blocs



(a) *Temps de calcul en fonction de la largeur de bloc.*



(b) *Puissance de calcul en fonction de la largeur de bloc.*

Figure 1 – *Temps et puissance de calcul en fonction de la largeur du bloc 1D.*

sont trop grands pour être le plus performant. En deçà de 64 *threads*, il y a sûrement trop de communication de données par rapport au temps total de calcul.

2.2 Analyse absolue

Dans l'absolu, ce produit matriciel avec un bloc 1D n'est pas très performant. En effet, pour comparaison, le même calcul avec le CPU a pris **3.73 secondes** pour une puissance de calcul de **36.81 gigaflops**. On voit donc que le CPU est plus performant que le GPU en bloc 1D. On comprend donc que la GPU n'est pas suffisamment monopolisée, le calcul pas assez bien distribué pour qu'il puisse être réellement rentable.

3 Allocations de blocs 2 dimensions sans mémoire partagée

3.1 Violation de la coalescence des données

Avant d'analyser les résultats concrètement obtenus, nous rappelons que, lors du TP, nous avons expérimenté un échange simple entre lignes et colonnes dans l'algorithme du produit, qui intervertissait juste le sens de parcours et donc le découpage en blocs lors du calcul de la matrice résultat. Nous avions alors vu que cette simple modification avait un impact considérable sur les résultats. Nous avions en fait observé une violation de la coalescence des données. En d'autres termes, les données parcourues en mémoire n'étaient plus, ou beaucoup

moins contiguës, ce qui avait pour conséquence une immense perte de temps, sans doute provoquée par des va et viens dans le cache ou bien par une plus grande durée moyenne des communications.

Analyse des résultats en 2D sur le kernel 2 en jouant sur X et Y sans mémoire partagée

Dans cette partie, nous allons donc jouer sur un deuxième facteur pour améliorer les performances et les rendre enfin meilleures que sur la CPU. Pour ce faire, nous allons désormais jouer sur **deux** paramètres afin d'étudier les résultats : la *hauteur* et la *largeur* des blocs. Ainsi, les blocs pourront potentiellement contenir des données en mémoire de manière plus contiguë. Cela pourrait potentiellement diminuer le temps de calcul étant donné que les mêmes données seront peut-être chargées en cache pour plusieurs *threads*.

La figure 2 est une représentation des résultats obtenus. Comme il s'agit d'un tableau à trois dimensions, le graphe utilisé ici, qui nous a gracieusement été fourni par notre enseignant, se sert de l'axe des abscisses pour la variable $\mathcal{X}$, la largeur du bloc, et de l'axe des ordonnées pour la variable $\mathcal{Y}$. La troisième variable, la puissance de calcul, est elle représentée par la couleur appliquée au niveau de l'intersection des deux points. On voit que la couleur bleu, par exemple, témoigne d'une puissance inférieure à 20 Gflops, tandis que la couleur violette témoigne d'une puissance de calcul entre 60 et 80 Gflops, la plus haute puissance de calcul enregistrée dans ce TP et avec le kernel 2 étant de **70.94** Gflops, pour $X = 16$ et $Y = 64$.

Là encore, on constate que les meilleurs résultats obtenus sont ceux ayant des X et des Y ayant des valeurs intermédiaires. Il s'agit des valeurs violette sur le graphique. Nous allons étudier plus précisément les 5 meilleurs scores, qui sont assez représentatifs.

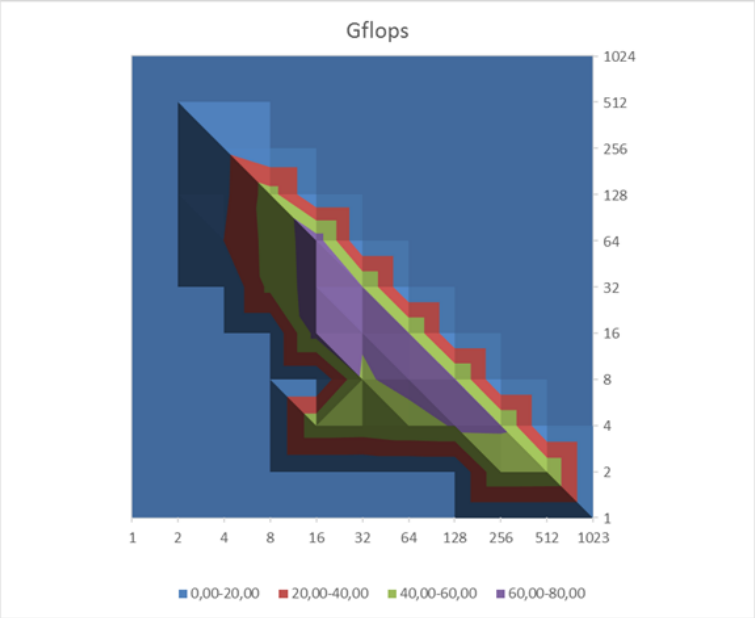


Figure 2 – *Puissance de calcul en fonction de la hauteur du bloc $\mathcal{Y}$ (en ordonnées) et de la largeur des blocs $\mathcal{X}$ (en abscisse).*

— $X = 16\,;\, Y = 64$ $(16 \times 64 = 1024) \rightarrow 70.94$ Gflops
— $X = 16\,;\, Y = 32$ $(16 \times 32 = 512) \rightarrow 70.00$ Gflops
— $X = 16\,;\, Y = 16$ $(16 \times 16 = 256) \rightarrow 68.16$ Gflops
— $X = 64\,;\, Y = 8$ $(64 \times 8 = 256) \rightarrow 61.20$ Gflops
— $X = 128\,;\, Y = 8$ $(128 \times 8 = 1024) \rightarrow 61.16$ Gflops

Bien qu'on ne puisse pas donner de règle absolue, étant donné que certains résultats ne diffèrent que très faiblement, on constate tout de même une certaine tendance. Si `X` est égal à 16, alors `Y` est inclus entre 16 et 32, tandis que si `Y` est égal à 8, X est égal à 64 ou 128. Ainsi leur produit est toujours inclus entre 256 et 1024.

Au contraire, on constate que de manière générale, les valeurs 1,2,4,256,512 et 1024 ne donnent pas les meilleurs résultat, quelque soit leur pendant sur l'autre axe, et quelque soit le fait qu'il s'agisse de `X` ou de `Y`.

On devine ainsi que le pavage optimal tend vers une forme carrée, avec un carré ni trop grand ni trop petit. Il est tout à fait compréhensible que le carré ne dût pas avoir une surface supérieure à 1024, puisque cela n'est pas supporté par CUDA. On comprend aussi que le carré ne doive pas être trop petit, car dans ce cas là, on aurait trop de communications et de chargements comparé

au nombre de calculs effectué par chaque bloc. En fait, on voit très bien sur la figure 2 que les valeurs du milieu sont les plus élevées.

4 Analyse des résultats en 2D sur le kernel 4 avec mémoire partagée

Pour l'instant nous ne nous étions pas préoccupés des allocations mémoires et des registres utilisés. Dans cette section, c'est exactement ce qui va nous intéresser. Nous allons tenter d'optimiser l'allocation des différentes mémoires de la GPU pour ce calcul matriciel.

4.1 Considérations théoriques

Allocation mémoire pour les sous-matrices de $\mathcal{C}$

Pour optimiser *matériellement* le produit matriciel, nous nous sommes servis de la **mémoire partagée** (plus couramment appelée sous son nom anglais *shared memory*). La mémoire partagée d'une GPU ayant une taille plutôt restreinte, il faut l'utiliser intelligemment. C'est pourquoi nous avons procédé de la manière suivante.

Nous savons que chaque *thread* est responsable, pour un bloc donné, d'un seul et unique élément de la matrice résultat $\mathcal{C}$. Or, pour calculer cet élément, il faut faire une addition incrémentale de produits d'éléments des matrices $\mathcal{A}$ et $\mathcal{B}$. Le registre est la mémoire la plus rapide pour les calculs sur GPU. Par conséquent, il suffit d'un seul registre,un `double`, pour stocker progressivement le résultat de cette somme incrémentale de produits. Comme il y a assez de registres pour stocker `X` $\times$ `Y` doubles sur la carte graphique. En effet, il y a au maximum besoin de `SIZE` $\times$ `SIZE` registres en même temps, dans le cas peu optimal où l'on utilise un seul bloc pour calculer tout le produit. Dans notre cas de figure, étant donné que de toute manière nous ne pouvions pas effectuer de calcul de plus de 11 secondes environ, nous ne pourrions jamais manipuler de matrice vraiment très grande, ce qui nous autorisait par conséquent à utiliser des registres pour stocker les résultat intermédiaire pour chaque *thread*. Dans un contexte où les blocs sont trop grands, nous eussions dû utiliser mémoire partagée pour stocker les résultats intermédiaires. C'est d'ailleurs la solution qui nous était présentée à l'origine dans l'enoncé.

En revanche, il fallait réaliser les produits à partir de sous-parties des matrices $\mathcal{A}$ et $\mathcal{B}$.

Allocation de mémoire partagée $\mathcal{A}$ et $\mathcal{B}$

Ces éléments étaient trop nombreux pour être stockés dans des registres pour chaque *thread*, nous devions donc les charger dans la **mémoire partagée**. Ainsi, pour calculer les éléments d'une sous-partie de la matrice résultat de $\mathcal{C}$, il fallait charger en mémoire partagée des lignes entières et des colonnes entières de $\mathcal{A}$ et $\mathcal{B}$. Ainsi, si notre sous-bloc de $\mathcal{C}$ faisait `BLOC_SIZE_X` colonnes pour `BLOC_SIZE_Y` lignes, il fallait charger en mémoire partagée `BLOC_SIZE_Y` $\times$ `SIZE` + `BLOC_SIZE_X` $\times$ `SIZE` éléments en mémoire partagée ! Soit (`BLOC_SIZE_Y` + `BLOC_SIZE_X`) $\times$ `SIZE` éléments.

Pour accélérer au maximum le produit matriciel, nous avons donc, pour un sous-bloc de $\mathcal{C}$ donnée, à chaque fois chargé les lignes et les colonnes correspondantes en mémoire partagée, qui est la deuxième mémoire la plus rapide d'accès pour les *threads* en dehors des registres. Comme $\mathcal{A}$ et $\mathcal{B}$ n'étaient ici accédées qu'en lecture, il n'y avait pas de problème d'accès concurrents à ce niveau.

Problèmes de synchronisation

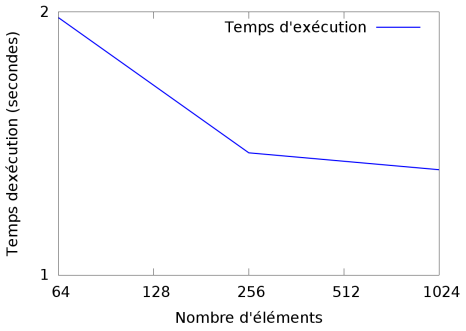
En revanche, pour $\mathcal{C}$, il était indispensable d'attendre que tous les *threads* aient fini leurs calculs pour un même élément du sous-bloc avant de calculer le sous-bloc suivant. Il fallait donc mettre une barrière, un appel de la fonction CUDA `sync_threads()`. Pour être plus précis, nous avons placé un appel à cette barrière une fois après les chargement dans la mémoire partagée, et une fois après l'enregistrement du registre dans la matrice $\mathcal{C}$. Ainsi on est certain que chaque *thread* a les bonnes données avant d'entamer ses additions, et que tous les résultats ont été enregistrés avant de passer au bloc suivant.

4.2 Analyse des résultats avec mémoire partagée

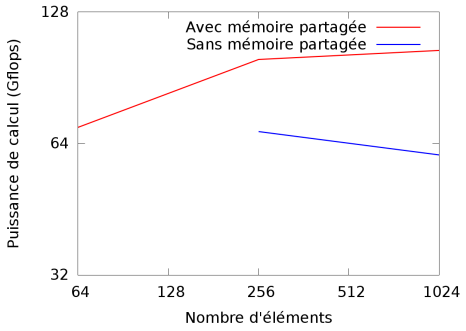
Pour se simplifier la tâche, nous ne comparons dans cette section que les performances de produits sur les grilles carrées de 64, 256 et 1024 éléments (respectivement 8, 16 et 32 cases de côté). Les performances sont données figure 3.

On constate que les résultats sont concluants. Pour rappel, avec un bloc 1D et avec un bloc 2D, le seul moyen d'améliorer les performances était de trouver le meilleur compromis au niveau de la taille des blocs. En 2D, le meilleur résultat se produisait avec un bloc de 16×64, tandis qu'un 1D il avait lieu avec un bloc de 1×6 . Dans tous les cas, des blocs trop grands donnaient des résultats moins bons.

Or, ici, bien qu'en échelle log-log la courbe de performances ne soit pas parfaitement linéaire, et donc que le performances ne s'améliorent pas exactement



(a) *Temps de calcul en fonction de la taille du bloc (en nombre d'éléments).*



(b) *Puissance de calcul en fonction de la taille du bloc (en nombre d'éléments) avec et sanns mémoire partagée.*

Figure 3 – *Temps et puissance de calcul en fonction de la taille du bloc de threads (`Size` × `Size`).*

proportionnellement au nombre d'éléments dans les blocs, on constate que les performances s'améliorent toujours et ce jusqu'à un bloc de 1024 éléments au moins. Deux conclusions très positives en découlent.

Premièrement, on obtenait une puissance de calcul de 68,16 Gflops pour des blocs 16 × 16 sans la mémoire partagée, contre **99,65** Gflops avec la mémoire partagée. Et bien que nous n'ayons pas pu mesurer les performances en 8 × 8 sans mémoire partagée, on voit tout de même que en 8 × 16 ou en sans mémoire partagée on est à 55,94 Gflops contre **69,7** Gflops en 8 × 8 avec mémoire partagée. Enfin on est à 60,27 Gflops en 32 × 32 sans mémoire partagée contre **104,51** Gflops avec mémoire partagée. On voit donc que dans l'absolu, les performances du calcul avec la mémoire partagée sont bien meilleures que sans la mémoire partagée.

De plus, on voit dans la courbe de performance que, alors que les performances baissent si le carré est trop grand sans mémoire partagée, elles **augmentent constamment** avec la mémoire partagée. Ainsi, il semble qu'on se libère d'une contrainte qui bloquait l'amélioration des performances, grâce à la mémoire partagée.

5 Conclusion

En conclusion, nous avons vu dans ce TP CUDA les différentes manières de programmer un produit matriciel de grande ampleur. Dans le pire des cas, avec des bloc 1D, nous atteignions des performances moins bonnes qu'un calcul séquentiel sur le CPU. Probablement parce que les communications et les allocations mémoire étaient trop gourmands comparé au temps effectif de calcul. Il est intéressant de noter qu'une augmentation des blocs peut entraîner une régression des performances.

Néanmoins, dès qu'on utilise des blocs 2D, et ce sans faire attention au détail de l'allocation mémoire, les performances sont bien meilleures sur le GPU que sur le CPU. En revanche, il faut faire très attention au dimensionnement des blocs, en essayant de trouver un bon compromis entre le ratio largeur/hauteur et la taille totale des blocs. Ce compromis doit être déterminé empiriquement, mais nous avons vu en cours et vérifié expérimentalement qu'il fallait chercher dans les valuers tournant autour de 16, 32 ou 64 *threads* de largeur pour un cours. Nous avons aussi mis en exergue l'importance de la coalescence des données, qui, si elle n'est pas respectée, peur grandement détériorer les performances globales.

Enfin, les performances sont très fortement améliorées si l'on gère attentivement l'allocation des données dans les registres et en mémoire partagée. une allocation astucieuse, mais qui nécessite en revanche de faire attention à la synchronisation des *threads*, nous permet d'améliorer les performances en fonction de la taille du carré, sans seuil. Performances qui deviennent logiquement bien meilleures que lorsqu'on ne gère pas du tout l'allocation des données en mémoire. En annexes, on trouvera le code produit qui illustre notre manière de résoudre les différents problèmes posés.

6 Annexe : code

6.1 Kernel 2 : Blocs 2D, sans gestion de mémoire

```
#define BLOCK_SIZE_X_K2      32
#define BLOCK_SIZE_Y_K2 BLOCK_SIZE_X_K2
```

```
__global__ void MatrixProductKernel_v0(void)
{
  // Calcul des index
  int col = (blockIdx.y * blockDim.y) + threadIdx.y;
  int lig = (blockIdx.x * blockDim.x) + threadIdx.x;

  // Si les index sont inferieurs a la taille des matrices
  if(col < SIZE && lig < SIZE){
    double res = 0.0;
    int i;

    // Calcul du produit d'une ligne de A et d'une colonne de B
    for(i=0;i<SIZE;i++)
    {
      res += GPU_A[i][lig]*GPU_B[col][i];
    }

    // Ecriture du resultat dans C
    GPU_C[col][lig] = res;
  }
}
[...]
  case GK2 :
  Db.x = BLOCK_SIZE_X_K2;
  Db.y = BLOCK_SIZE_Y_K2;
  Db.z = 1;
  // La condition ternaire ajoute un bloc si la taille de la
    matrice n'est pas une puissance de 2
  Dg.x = SIZE/BLOCK_SIZE_X_K2 + (SIZE%BLOCK_SIZE_X_K2  ?  1  :
    0);
  Dg.y = SIZE/BLOCK_SIZE_Y_K2 + (SIZE%BLOCK_SIZE_Y_K2  ?  1  :
    0);
  Dg.z = 1;
  MatrixProductKernel_v0<<<Dg,Db>>>();
  break;
```

6.2 Kernel 4 : Blocs 2D, avec gestion de mémoire

```
#define BLOCK_SIZE_X_K4      32
#define BLOCK_SIZE_Y_K4 BLOCK_SIZE_X_K4
```

```
__global__ void MatrixProductKernel_v1(void){
  int i,j;
  // Calcul des index
  int col = (blockIdx.x * blockDim.x) + threadIdx.x;
  int lig = (blockIdx.y * blockDim.y) + threadIdx.y;
  double res = 0.0;

  // Acces a la memoire partagee
  __shared__ double shared_A[BLOCK_SIZE_X_K4][BLOCK_SIZE_Y_K4];
  __shared__ double shared_B[BLOCK_SIZE_X_K4][BLOCK_SIZE_Y_K4];

  // Pour chaque bloc de la matrice
  for(i = 0; i <  (SIZE / BLOCK_SIZE_X_K4)+(SIZE%BLOCK_SIZE_X_K4
    ?1:0); i++){
    // Calcul de l'offset
    int offset = i * BLOCK_SIZE_X_K4;

    // Si le thread n'est pas hors limite de A
    if((offset+threadIdx.x) < SIZE && lig < SIZE){
      shared_A[threadIdx.y][threadIdx.x] = GPU_A[lig][offset +
    threadIdx.x];
    }
    // Si le thread n'est pas hors limite de B
    if((offset+threadIdx.y) < SIZE && col < SIZE){
      shared_B[threadIdx.y][threadIdx.x] = GPU_B[offset +
    threadIdx.y][col];
    }
    __syncthreads();
    // On attend que tous les threads ont finis d'ecrire


    // Si le bloc de threads ne se trouve pas en bordure
    if(offset < SIZE-1){
      for(j=0;j<BLOCK_SIZE_X_K4;j++){
        res += shared_A[threadIdx.y][j]*shared_B[j][threadIdx.x];
      }
    // Sinon seul les threads eligibles peuvent calculer
    }else{
      for(j=0;j< (SIZE%BLOCK_SIZE_X_K4);j++){
        res += shared_A[threadIdx.y][j]*shared_B[j][threadIdx.x];
      }
```

```
    }

    if(col < SIZE && lig < SIZE){
      GPU_C[lig][col] = res;
    }
    __syncthreads();
    // On attend que tous les threads ont finis
  }
}
[...]
  case GK4 :
  Db.x = BLOCK_SIZE_X_K4;
  Db.y = BLOCK_SIZE_Y_K4;
  Db.z = 1;
  // La condition ternaire ajoute un bloc si la taille de la
    matrice n'est pas une puissance de 2
  Dg.x = SIZE/BLOCK_SIZE_X_K4 + (SIZE%BLOCK_SIZE_X_K4  ?  1  :
    0);
  Dg.y = SIZE/BLOCK_SIZE_Y_K4 + (SIZE%BLOCK_SIZE_Y_K4  ?  1  :
    0);
  Dg.z = 1;
  MatrixProductKernel_v1<<<Dg,Db>>>();
  break;
```