Comparaison de l'exécution des algorithmes naïfs de multiplication de matrice sur CPU et GPU

Picard Michaël

Le but de cette expérience est de montrer les différences de performances entre la multiplication sur CPU et sur GPU de matrices, en utilisant les algorithmes naîfs de calculs. Ceci dans le but de débuter en programmation parrallèle et d'expérimenter techniques et utilisation du language CUDA.

Nous allons tester le temps d'exécution de 4 algorithmes :

- le 1er en CPU naïf
- le 2eme en GPU naïf, avec division en blocks
- le 3eme en GPU avec mémoire shader,i.e la mémoire locale à chaque block du GPU, et avec division en blocks
- le 4ème est une modification du 3eme pour augmenter le nombre de calcul exécuté par thread à 4, en posant une taille de blocks et de variable locales fixes

Tout ces tests seront effectué avec des matrices alloué localement au centre de calcul (CPU ou GPU).

Nous nous attendons à des performances qui décroisse exponentiellement avec la taille de la matrice pour le 1er algorithme (CPU), et des performances à peu près stables pour les 2 algorithmes sur GPU, avec un léger gain de puissance pour le 3eme algorithme, ainsi qu'à une possible amélioration supplémentaire pour le dernier, avec un doute sur son efficacité dû aux transferts mémoires.

I Algorithme

CPU : Algorithme naïf exécuté sur le CPU, de la même manière qu'un humain le ferait.

```
void multiplicationMatriceCPU(const MatriceCPU *m1,const MatriceCPU *m2,
     MatriceCPU *resultat){
    Element tmp;
    for(int i=0;i<m1->dimension;i++){
        for(int j=0;j<m1->dimension;j++){
            tmp=ZERO_ELEMENT;
            for(int k=0;k<m1->dimension;k++)
                tmp=additionElement(tmp,multiplicationElement(
                  m1->matrice[i*m1->dimension+k],
                  m2->matrice[k*m1->dimension +j]));
            resultat->matrice[positionElement(i,j,resultat)]=tmp;
        }
    }
}
```

N.B (pour les algorithmes GPUx):

- divMaxDim renvoie la plus grand diviseur (<=32) de la dimension envoyé
- dimSup renvoie la dimension supérieur multiplie de 64

GPU1 : Algorithme naïf, qui divise la matrice en block carrés de même dimension et qui calcule de manière humaine chaque case de la matrice

```
__global__ static void multiplicationMatriceGPU_Kernel(const MatriceGPU m1,
                        const MatriceGPU m2,MatriceGPU resultat,
                        const int nbThreadPerBlock){

    // On détermine les coordonnées de la case à calculer
    unsigned long ligne= blockIdx.y*nbThreadPerBlock + threadIdx.y,
                  colonne= blockIdx.x*nbThreadPerBlock + threadIdx.x;
    Element sum = ZERO_ELEMENT;

    for(int k=0;k<resultat.dimension;k++)
        sum = additionElement(sum,multiplicationElement(
          m1.matrice[ligne*m1.dimension+k],
          m2.matrice[k*m2.dimension+colonne]));

    // On affecte le résultat à sa case (ordonné par ligne et colonne)
    resultat.matrice[positionElement(ligne,colonne,&resultat)] = sum;
}

void multiplicationMatriceGPU(const MatriceGPU *m1,const MatriceGPU *m2,
     MatriceGPU *resultat){
    //Matrice de dimension dimension*dimension
    const unsigned long dim=resultat->dimension;
    //dimension d'un block = div *div <= 32*32
    int div = divMaxDim(dim);
    //On sépare notre matrice en divG*divG block
    int divG = dim/div;
    dim3 dimBlock(div,div,1),dimGrid(divG,divG,1);
    // Appel du Kernel
    multiplicationMatriceGPU_Kernel<<<dimGrid,dimBlock>>>(*m1,*m2,*resultat,div);
    // On attend que tous les calculs soit terminés
    cudaDeviceSynchronize();
```

GPU2 : Amélioration de l'algorithme GPU1 en stockant temporairement des éléments de la matrice dans le but de réduire les appels à la mémoire globale GPU

```
__global__ static void multiplicationMatriceGPU_Kernel(const MatriceGPU m1,
                        const MatriceGPU m2,MatriceGPU resultat,
                        const int nbThreadPerBlock){
    Element sum = ZERO_ELEMENT;
    // On utilise la mémoire locale, en utilisant le fait que la
    // matrice est découpé en block de dimension <= 32*32
    __shared__ Element Mgshader[32][32];
    __shared__ Element Ngshader[32][32];

    // Index des block et thread
    int bx=blockIdx.x,by=blockIdx.x,tx=threadIdx.x,ty=threadIdx.y;
    // On détermine les coordonnées de la case à calculer
    int ligne = by*nbThreadPerBlock+ty, colonne = bx*nbThreadPerBlock+tx;
    unsigned long Width = resultat.dimension;

    for(int s=0;s<(Width/nbThreadPerBlock);s++)
    {
      // On remplit les tableaux temporaires locaux
```

```
        Mgshader[ty][tx]=m1.matrice[ligne*Width+(s*nbThreadPerBlock + tx)];
        Ngshader[ty][tx]=m2.matrice[colonne+Width*(s*nbThreadPerBlock + ty)];
        __syncthreads(); // On attend que les tableaux soient remplis

        for(int k=0;k<nbThreadPerBlock;k++){ // on calcule la valeur temporaire
            sum=additionElement(sum,multiplicationElement(
              Mgshader[ty][k],Ngshader[k][tx]));
        }
        __syncthreads(); // On attend la fin de tous les calculs
    }
    // On affecte le résultat à sa case (ordonné par ligne et colonne)
    resultat.matrice[ligne*Width+colonne] = sum;
}

void multiplicationMatriceGPU(const MatriceGPU *m1,const MatriceGPU *m2,
     MatriceGPU *resultat){
    //Matrice de dimension dimension*dimension
    const unsigned long dim=resultat->dimension;
    //dimension d'un block = div *div <= 32*32
    int div = divMaxDim(dim);
    //On sépare notre matrice en divG*divG block
    int divG = dim/div;
    dim3 dimBlock(div,div,1),dimGrid(divG,divG,1);
    // Appel du Kernel
    multiplicationMatriceGPU_Kernel<<<dimGrid,dimBlock>>>(*m1,*m2,*resultat,div);
    // On attend que tous les calculs soit terminés
    cudaDeviceSynchronize();
```

GPU3 : Amélioration de l'algorithme GPU2 en fixant comme paramètre que chaque coté des matrices en entrées est multiple de 64, que les tableaux locaux sont de taille 64*64 toujours rempli et que les blocks sont tous de taille 64*16

```
__global__ static void multiplicationMatriceGPU_Kernel(const MatriceGPU m1,
                       const MatriceGPU m2,MatriceGPU resultat){
    Element sum[4];
    // Tableau des 4 variables locales correspondant aux 4 cases de la
    // matrices que l'on calcule par thread
    for(int i=0;i<4;i++)
        sum[i]=ZERO_ELEMENT;

    // On crée les tableaux temporaires locaux de dimension
    // 4 fois (64*64) celle d'un block (64*16)
    __shared__ Element Mgshader[64][64];
    __shared__ Element Ngshader[64][64];

    // Index des blocks et threads
    int bx=blockIdx.x,by=blockIdx.x,tx=threadIdx.x,ty=threadIdx.y;
    for(int s=0;s<gridDim.x;s++){ //On remplit les tableaux temporaires locaux
        for(int i=0;i<4;i++){//
            Mgshader[ty+i*16][tx]=m1.matrice[resultat.dimension*(s*64+ty+i*16)+bx*64+tx];
            Mgshader[ty+i*16][tx]=m2.matrice[resultat.dimension*(by*64+ty+i*16)+s*64+tx];
        }
        __syncthreads(); // On attend que les tableaux soient remplis
```

```
        for(int i=0;i<4;i++){ // Pour chaque case,
            for(int k=0;k<64;k++) // on calcule la valeur temporaire
                sum[i]=additionElement(sum[i],multiplicationElement(
                  Mgshader[ty+i*16][k],Ngshader[k][tx+i*16]));
        }
        __syncthreads(); // On attend la fin de tous les calculs
    }
    // On affecte chaque résultat à sa case (ordonné par les blocks et les threads)
    for(int i=0;i<4;i++)
        resultat.matrice[resultat.dimension*(64*by+ty+i*16)+bx*64+tx]=sum[i];
    __syncthreads();
}

void multiplicationMatriceGPU(const MatriceGPU *m1,const MatriceGPU *m2,
     MatriceGPU *resultat){
    //Matrice de dimension dimension*dimension
    const unsigned long dim=resultat->dimension;
    unsigned long dimSup=dimSUP(dim); //dimSup = X*64 >=dim
    // On divise la matrice de dimension dimSup en nbBlock*nbBlock block
    unsigned long nbBlock=dimSup>>6;
    dim3 dimBlock(64,16,1),dimGrid(nbBlock,nbBlock,1);
        // On copie les matrices entrées dans des matrices
        // de dimension supérieures ou égales
        MatriceGPU *m1bis=initialiserMatriceGPU(dimSup);
        MatriceGPU *m2bis=initialiserMatriceGPU(dimSup);
        cpMatriceDimDiff(m1,m1bis);cpMatriceDimDiff(m2,m2bis);
        MatriceGPU *resultatbis=initialiserMatriceGPU(dimSup);
    // Appel du Kernel
    multiplicationMatriceGPU_Kernel<<<dimGrid,dimBlock>>>(*m1bis,*m2bis,*resultatbis);
    cudaDeviceSynchronize(); // On attend que tous les calculs soit terminés
        cpMatriceDimDiff(resultatbis,resultat); // on récupére le bon résultat
        freeMatriceGPU(m1bis);freeMatriceGPU(m2bis);freeMatriceGPU(resultatbis);
}
```

II Matériel

Description du GPU (via deviceQuery) :

```
  Quadro 600
  CUDA Capability Major/Minor version number:    2.1
  Total amount of global memory:                 1023 MBytes (1072889856 bytes)
  ( 2) Multiprocessors, ( 48) CUDA Cores/MP:     96 CUDA Cores
  GPU Max Clock rate:                            1280 MHz (1.28 GHz)
  Memory Clock rate:                             800 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 131072 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65535), 3D=(2048, 2048, 2048)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
```

```
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1536
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (65535, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
```

Description du système (via *lstopo*) :

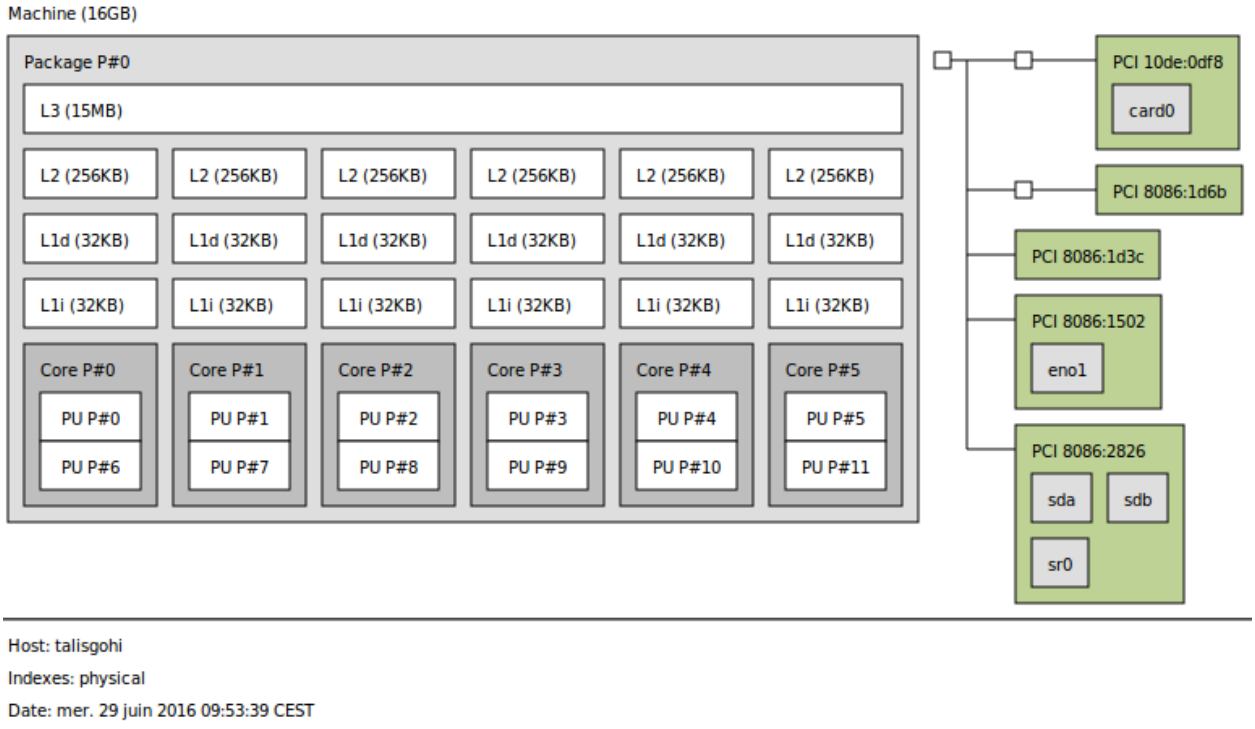


Figure 1: *Description du système*

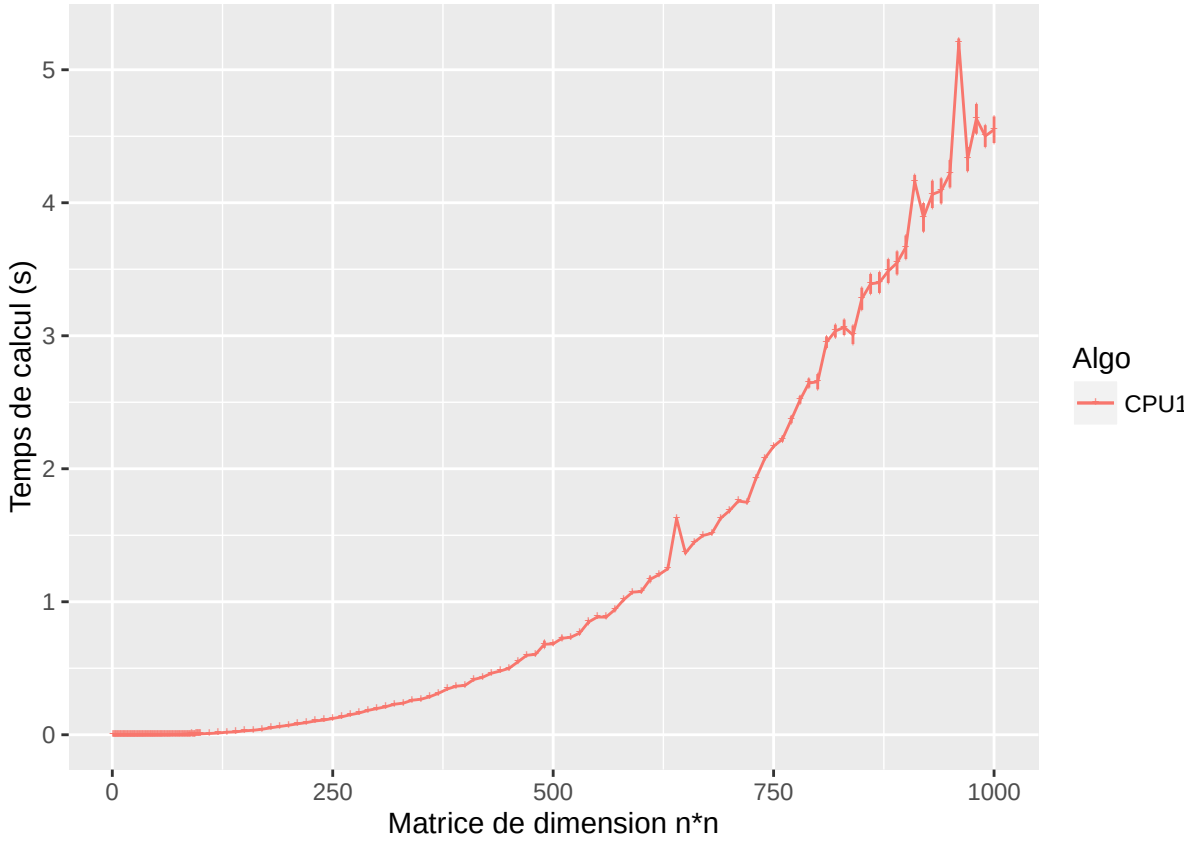
III Resultats d'expériences

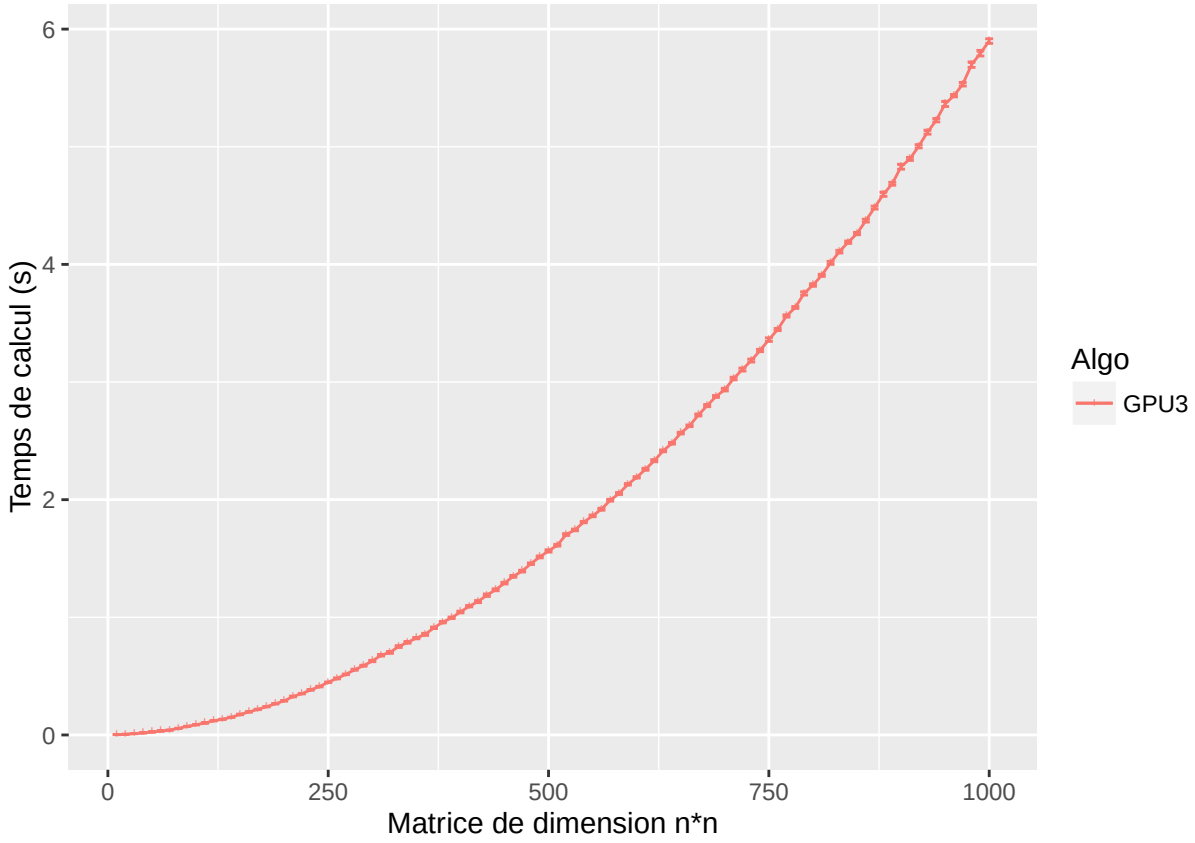
Nous avons choisi de séparer certain affichage :

- L'algorithme CPU est séparé des algorithmes GPU : il s'agit d'un choix nécessaire, car il est à la fois logique de séparé les coeurs de calculs et nécessaire de la faire pour une meilleur lisibilité des résultats.
- Les algorithmes GPU seront séparés entre eux s'il est nécessaire de le faire par soucis de lisibilité

Sur chacun des graphiques présentés ci-dessous, un point correspond à une moyenne de 32 run et les barres d'erreurs sont affichées.

1 Temps d'exécution

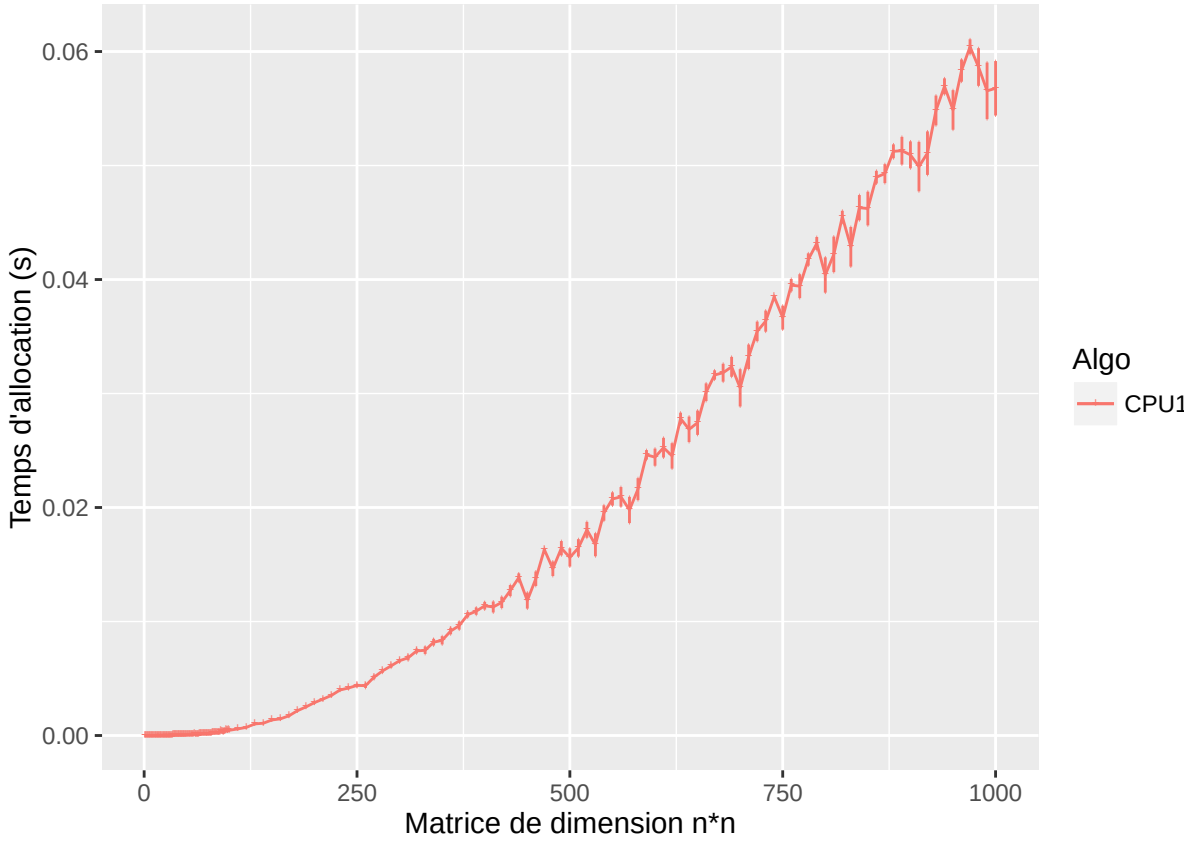




On remarque ici que, comme attendu, l'algorithme CPU en dimensions élevées de matrice (clairement lisible

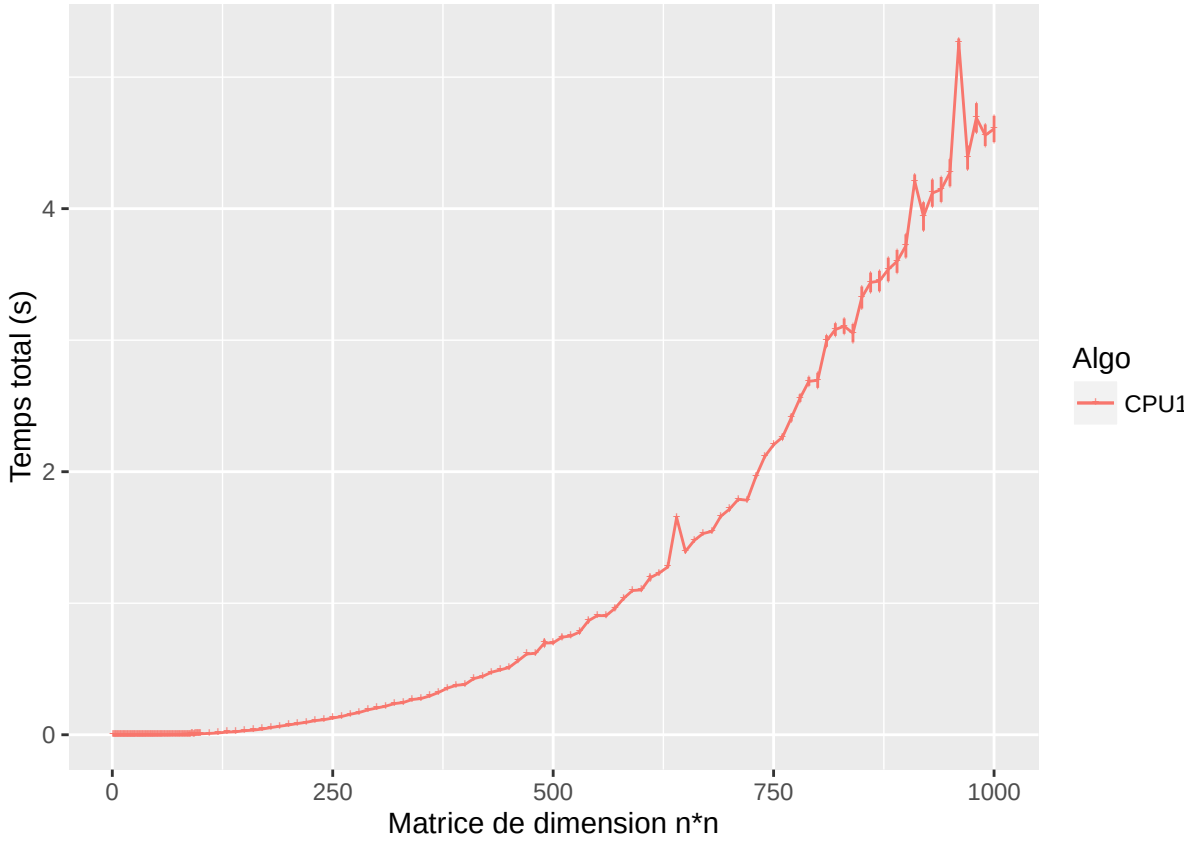
dès la dimension 500*500) est plus lent que les algorithmes GPU1 et GPU2, le GPU2 étant plus rapide que le GPU1. Fait surprenant, l'algorithme GPU3 que l'on espérait plus efficace que les autres est très rapidement le plus lent (très clairement visible dès la dimension 500*500).

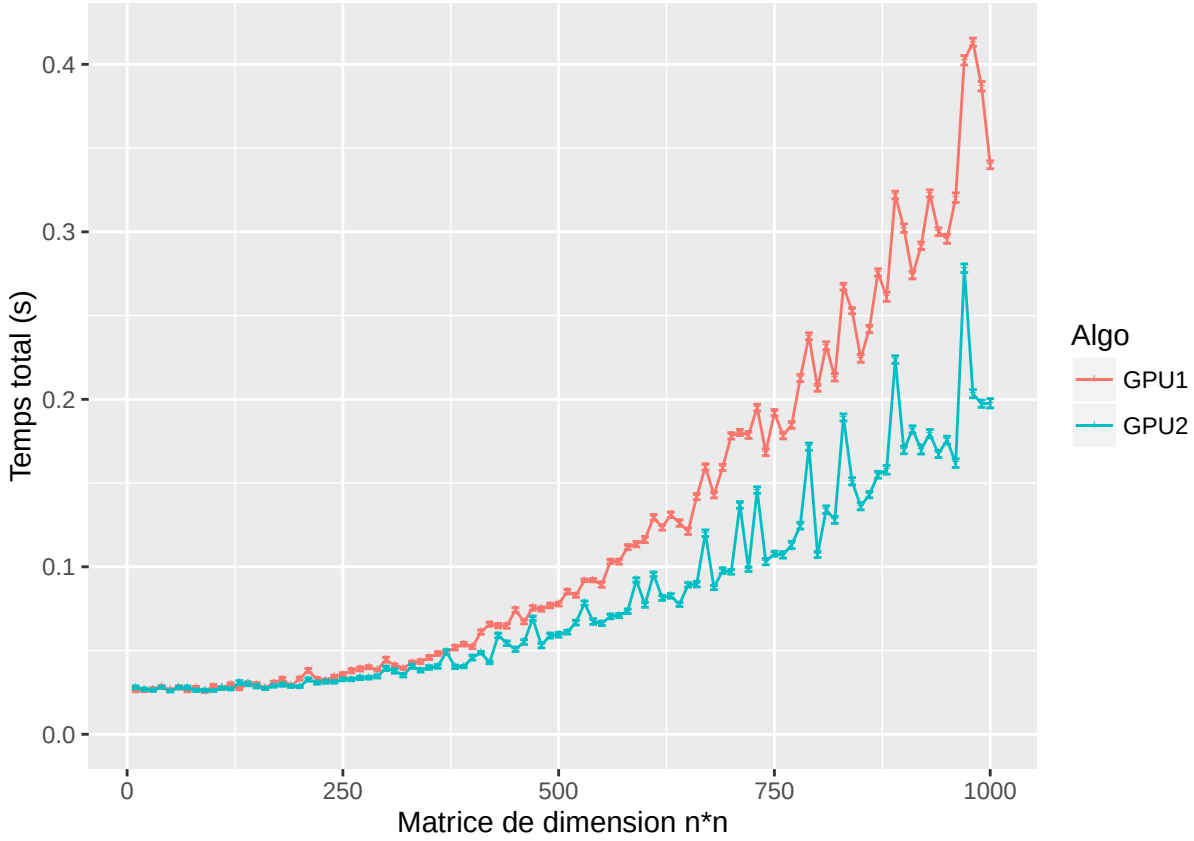
2 Temps d'allocation

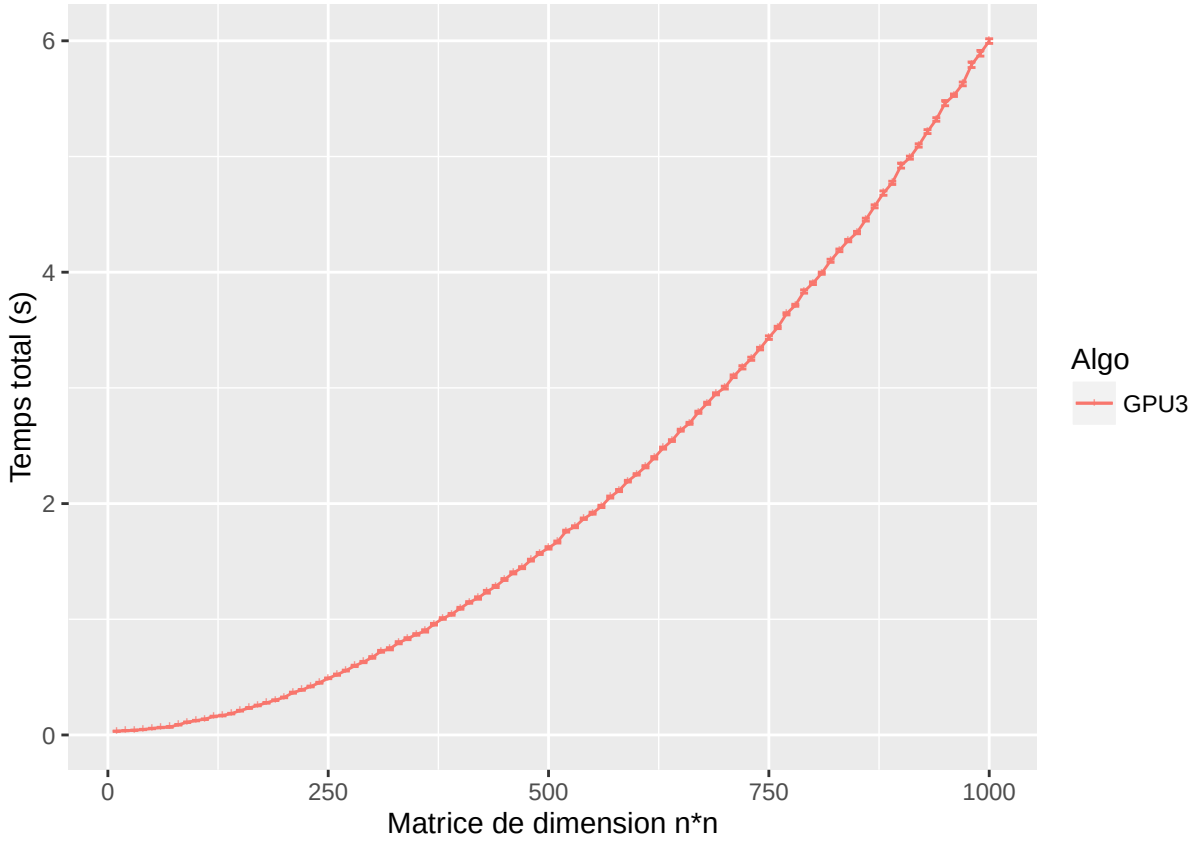


Comme attendu sur la théorie de l'allocation mémoire du CPU et du GPU, le temps d'allocation des matrices sur le GPU est clairement supérieur à celui du CPU. Un autre fait surprenant est que le temps d'allocation des matrices pour l'algorithme GPU3 est supérieur au 2 autres algorithmes GPU, alors qu'il s'agit d'une allocation strictement identique.

3 Temps total







Ici, nous avons rassemblé les temps d'allocations et de calculs de chaque algorithme : nous avons donc les

mêmes observation que pour les temps d'exécutions, car les temps d'allocations sont négligeables à partir d'un certain rang pour les algorithmes CPU et GPU3, et que les temps pour GPU1 et GPU2 n'ont pas d'incidence avec la comparaison entre eux et avec les autres algorithmes.

4 Observation

Globalement les résultats concordent avec nos attentes ; on peut dire qu'à partir du de la dimension 500*500, les algorithmes sont classé dans l'ordre attendu (CPU>GPU1>GPU2, en temps d'exécution) avec une vitesse d'allocation plus lente dans notre plage de test pour le GPU que pour le CPU. De plus, la différence de transfert de données pour les calculs entre les algorithmes GPU1 et GPU2 est bien visible est permet au GPU2 une meilleur efficacité temporelle.

Mais il y a quelques demi-surprises : le temps d'allocation, censé être identique pour chaque GPU est globalement le même, malgré une augmentation plus rapide de ce temps en fonction de la taille des matrices pour l'allocation nécessaire au GPU3. De plus, notre doute était fondé : les copies nécessaires dans des matrices plus grandes pour le GPU3 ajoutent des temps de calculs plus long même que la version CPU, que ce soit avec ou sans l'allocation de base, ce qui rend cet algorithme totalement incorrect et non valable. (face au algorithme GPU1 et GPU2). Enfin on remarque une lègere baisse de temps nécessaire pour l'allocation sur CPU entre les dimensions 32*32 et 256*256.

IV Conclusion

Grâce à cet expérience, nous avons pu saisir quelques prémices d'utilisation et de questionnement sur la programmation parrallèle sur GPU via le language CUDA, en nous basant sur la multiplication matricielle. Nous avons ainsi pu apréhender la méthode de pensées nécessaire à la parrallèlisation des calculs, ainsi que nous pencher sur les questions de transferts mémoires pouvant avoir un impact sur la vitesse de calculs.

Pour pousser plus loin sur la multiplication matricielle, il serait utile de faire varier l'emplacement mémoire pré-calcul pour introduire la variable dû à la localité ou non des données (ici, toutes les matrices étaient stockées sur la mémoire correspondant à leur unités de calcul [CPU ou GPU] respectives), ou de comparer en ne prenant que des tailles de matrice multiples de 2 (en améliorant peut être l'algorithme GPU3 qui dans le cas particulier où l'on a des matrices de dimension (X*64)*(X*64) ne ferait pas de copies). Il pourrait aussi être intéressant d'adapter ces algorithmes pour des matrices n'étant pas forcément carrées, et d'observer les différences de performances induites.