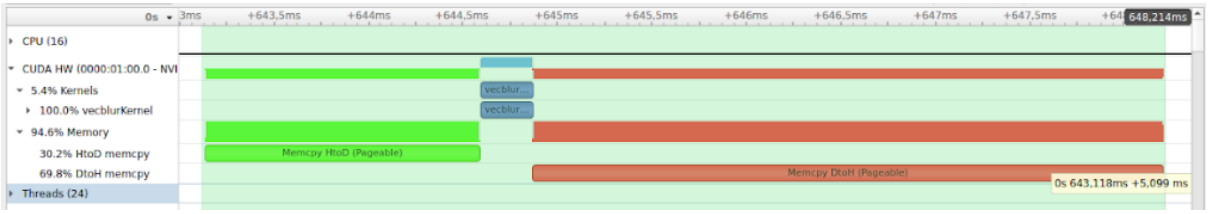
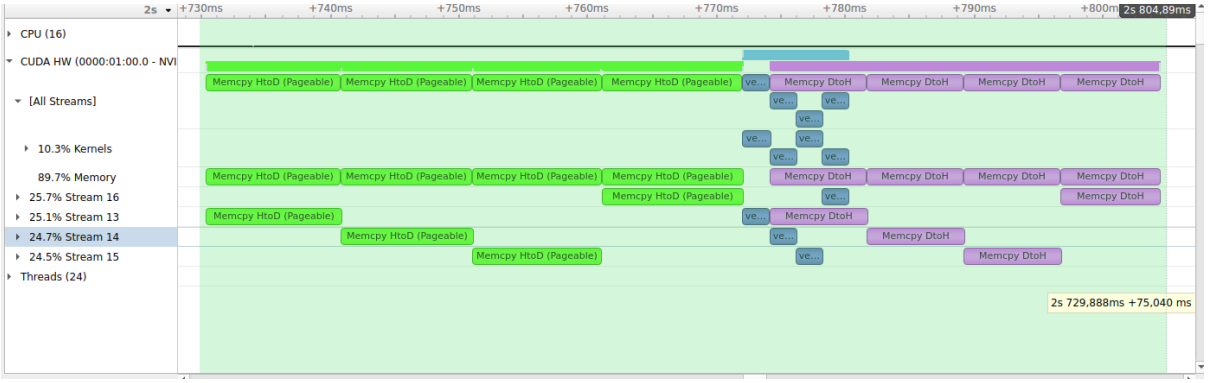
Chapitre 12 :

Profile the new version and report the total execution time from start of the first copy to end of last copy. Compare with the initial version.

Initial :



Nouvelle :



Chapitre 13 :

1. Explain all your choices of optimisation in version 2.

Version 1 :

```
rcourtenay@scidesse143:~/Téléchargements/GPGPU/CHAP13/EX2$ nvc++ -acc=gpu -Minfo
=accel 2-basicMatMulTiled.c matmul_utils.cxx
2-basicMatMulTiled.c:
main:
     31, Generating copy(a[:numAColumns*numARows],c[:numCRows*numCColumns],b[:nu
mBColumns*numBRows]) [if not already present]
     33, Generating NVIDIA GPU code
         35, #pragma acc loop gang /* blockIdx.x */
         37, #pragma acc loop seq
         40, #pragma acc loop vector(128) /* threadIdx.x */
             Generating reduction(+:emp_c)
     37, Loop is parallelizable
     40, Loop is parallelizable
matmul_utils.cxx:
```

Version 2 :

```
     31, Generating copy(a[:numAColumns*numARows],c[:numCRows*numCColumns],b[:nu
mBColumns*numBRows]) [if not already present]
     33, Generating NVIDIA GPU code
         35, #pragma acc loop gang collapse(2) /* blockIdx.x */
         36,   /* blockIdx.x collapsed */
         39, #pragma acc loop vector(128) /* threadIdx.x */
             Generating reduction(+:emp_c)
     39, Loop is parallelizable
```

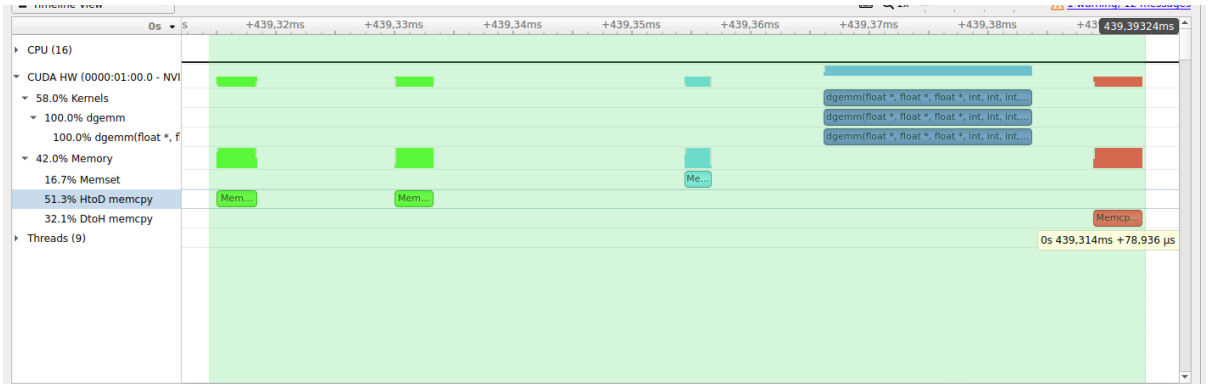
Voici mon code pour la 2ème version :

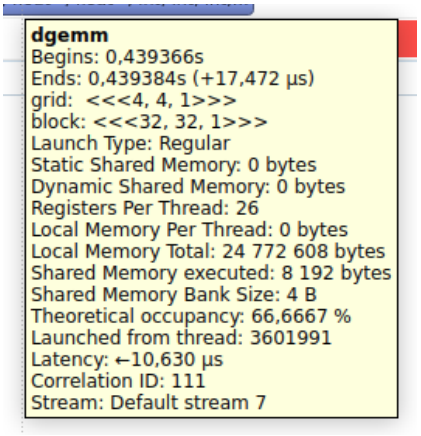
```
// Call function
#pragma acc data copy (a[0:numARows*numAColumns], b[0:numBRows*numBColumns], c[0:numCRows*numCColumns])
{
#pragma acc parallel
{
#pragma acc loop collapse(2) gang vector
      for (int x=0; x<numBColumns; x++){
      for (int y=0; y<numARows; y++){
             float emp_c = 0;
#pragma acc loop collapse(1) reduction(+:emp_c)
             for (int k=0; k<numAColumns; k++){
             emp_c += a[k+numAColumns*y] * b[x+numBColumns*k];
             }
      c[x+numBColumns*y] = emp_c;
      }
      }
}
}
```

Tout comme ma première version j'ai copié les tableaux a, b et c du host au device avant d'exécuter le code en parallèle. Ainsi, on peut traiter les tableaux par les threads en parallèle.
Ensuite, j'indique qu'il faut exécuter le code en parallèle.
La première ligne de code *acc loop collapse* indique qu'il y a 2 boucles imbriquées qui sont exécutées en parallèle. La boucle la plus externe doit être distribuée sur plusieurs gangs (divise l'espace de travail) et la boucle interne doit être parallélisée sur plusieurs vectors. En effet, à l'intérieur des gangs, on exécute plusieurs opérations simultanément. Ainsi, on optimise l'utilisation des ressources.
La deuxième directive indique que la boucle doit être exécutée en parallèle avec une réduction sur la variable emp_c (toutes les valeurs calculées pour emp_c doivent être sommées en utilisant l'opérateur d'addition).

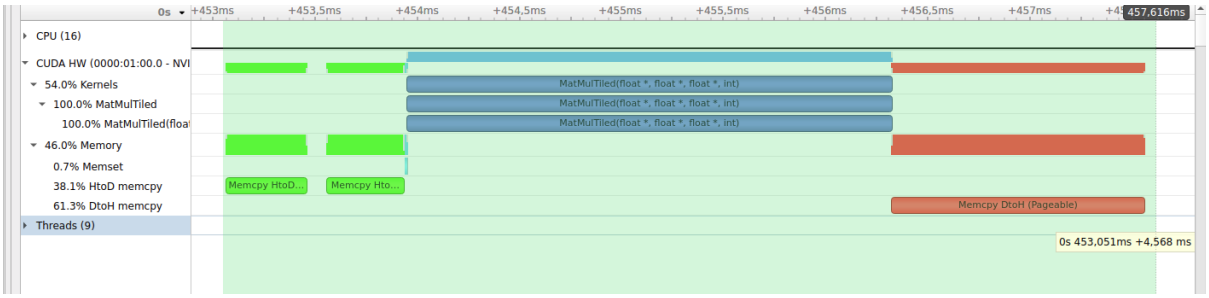
2. Compare all your versions of matrix multiplication you wrote so far : CUDA versions from chapter 4 and OpenACC from this exercise. What is the best version? EXPLAIN

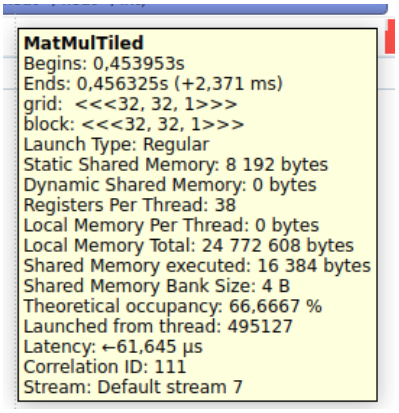
Version original :



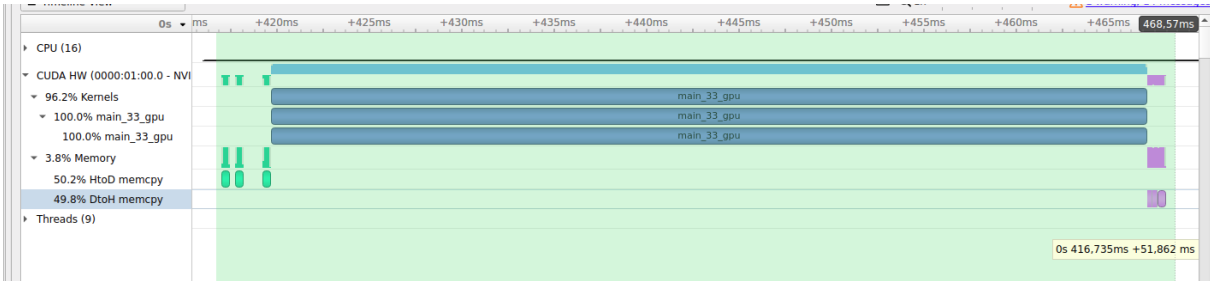


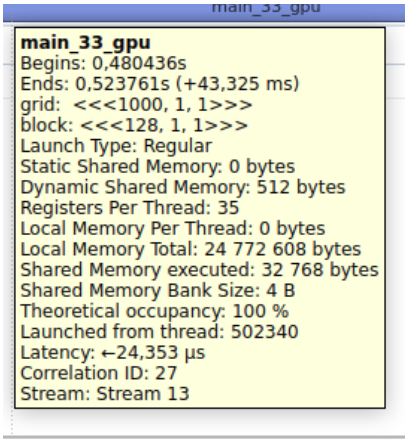
Version avec tuile :



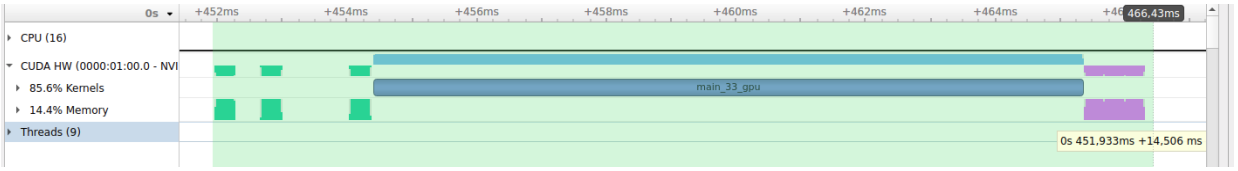


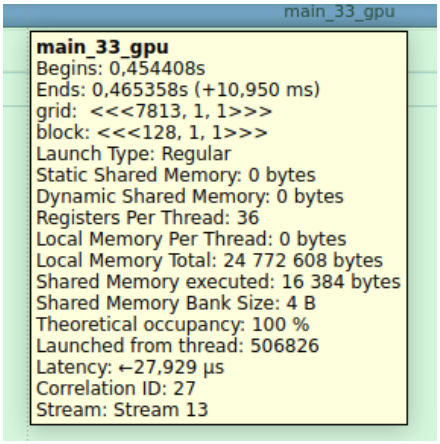
Version 1 :





Version 2 :





Versions	Temps
Première	17.472ms
Avec tuiles	2.371ms
OpenACC	43.325ms
OpenACC avec tuiles	10.950ms

La version avec les tuiles est la plus rapide suivie de la version avec OpenACC. L'utilisation de tuiles améliore les performances de la multiplication de matrices en mémoire cache, en réduisant les accès à la mémoire principale et en optimisant l'utilisation de la mémoire cache. Alors que dans mon code, j'optimise parallèlement la multiplication de matrices sans intégrer les tuiles.