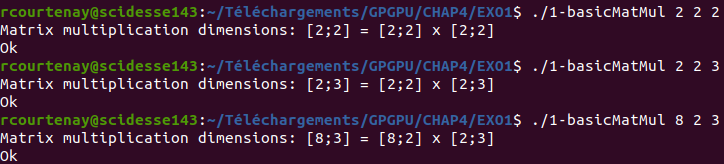
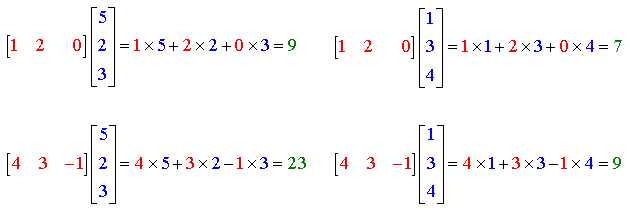
# Exercice 1

## cd /home/42/trecappe/Téléchargements/gpgpu/Chapter 4 - Exercice 1-20221019





## Before coding : What are the relations between the three matrix dimensions to have a well defined multiplication ?

Le nombre de colonnes doit être égale au nombre de lignes de la seconde matrice à multiplier.

## How many floating operations are being performed in your matrix multiply kernel ?

Si deux matrices : A (n\*m) et B (m\*k)

Nombre d’opération flottante : 2\*n\*m\*k

## How many global memory reads are being performed by your kernel ?

Accès en lecture (Al) : (à chaque fois qu’il y a un calcul on utilise 2 valeurs) 2\*n\*m\*k

## How many global memory writes are being performed by your kernel ?

Accès en écriture (Ar) : (taille matrice résultat) m\*k

## Compute the arithmetic intensity of your kernel. The arithmetic intensity is a FLOP/Byte number standing for the number of floating point operations performed per byte of global memory accessed.

Intensité arithmétique : rapport entre le nombre d’opération en virgule flottante et la quantité de mémoire qui est nécessaire à faire ces opérations → Nbr op/(Al+Ar)[\*4 (car en float et faut mettre en bytes (si double précision faut mettre 8))]

Peut simplifier le calcul : (2\*m)/(4(2\*m+1))

Limite quand m tend vers l’infini : 2m/8m : ¼

→ cet algo est pas très efficace sur le GPGPU (une opération arithmétique : besoin de 4 bytes de données (4 fois plus de données que de calcul))

bande passante de la mémoire est bcp plus faible que la puissance de calcul que le GPGPU

le GPGPU pas trop performant car limité par bande passante de la mémoire

**Matmult version Thomas 👍**

#include <stdio.h>

#include <cuda.h>

#include <time.h>

#include "matmul\_utils.hpp"

// Cuda kernel

\_\_global\_\_ void dgemm(float \*A, float \*B, float \*C,

int numARows, int numAColumns, int numBRows, int numBColumns) {

// @TODO@ : Complete here the kernel code

int x =blockIdx.x\*blockDim.x + threadIdx.x; // colonne horizontal pixel coordinate

int y =blockIdx.y\*blockDim.y + threadIdx.y; // ligne vertical pixel coordinate

if (x < numBColumns && y < numARows) {

int C\_Offset = y\*numBColumns+x;

C[C\_Offset] = 0;

for(int k = 0; k < numAColumns; k++) {

int indice\_A = y\*numAColumns+k;

int indice\_B = k\*numBColumns+x;

C[C\_Offset] += A[indice\_A] \* B[indice\_B];

}

}

}

int main(int argc, char \*\*argv)

{

if(argc!=4) {printf("Usage : %s [nb of rows for A] [nb of cols for A] [nb of cols for B]\n", argv[0]);exit(2);}

//initilize a pseudo-random number generator

srand(time(0));

int numARows, numAColumns,numBRows, numBColumns,numCRows, numCColumns;

// Read given dimensions

numARows = atoi(argv[1]); //C = A\*B

numAColumns = atoi(argv[2]);

numBColumns = atoi(argv[3]);

// Compute the remaining dimensions for given ones

numBRows = numAColumns; //@TODO@ ok

numCRows = numARows; //@TODO@ ok

numCColumns = numBColumns; //@TODO@ ok

printf("Matrix multiplication dimensions: [%d;%d] = [%d;%d] x [%d;%d]\n",

numCRows, numCColumns, numARows, numAColumns, numBRows, numBColumns);

// host pointers

float \*host\_a, \*host\_b, \*host\_c;

// Device pointers

float \*dev\_a, \*dev\_b, \*dev\_c;

// Allocations on host

host\_a = (float \*)calloc(numARows\*numAColumns, sizeof(float));

host\_b = (float \*)calloc(numBRows\*numBColumns, sizeof(float));

host\_c = (float \*)calloc(numCRows\*numCColumns, sizeof(float));

// Initialize vectors

init(host\_a,host\_b,numARows, numAColumns, numBRows, numBColumns);

// Allocations on device

// @TODO@ : complete device allocations

cudaMalloc((void \*\*) &dev\_a, numARows\*numAColumns\*sizeof(float));

cudaMalloc((void \*\*) &dev\_b, numBRows\*numBColumns\*sizeof(float));

cudaMalloc((void \*\*) &dev\_c, numARows\*numBColumns\*sizeof(float));

// Copy from host to device

// @TODO@ : complete copy from host to device

cudaMemcpy(dev\_a, host\_a, numARows\*numAColumns\*sizeof(float), cudaMemcpyHostToDevice);

cudaMemcpy(dev\_b, host\_b, numBRows\*numBColumns\*sizeof(float), cudaMemcpyHostToDevice);

// Invoke kernel

// @TODO@ : complete compute grid and block dim

dim3 DimBlock(16, 16, 1);

dim3 DimGrid((numARows-1)/DimBlock.x + 1, (numBColumns-1)/DimBlock.x+1, 1);

// Initialize C device data

cudaMemset(dev\_c, 0, numARows \* numBColumns \* sizeof(float));

// Call the kernel

// @TODO@ : complete to call the kernel

dgemm<<<DimGrid,DimBlock>>>(dev\_a, dev\_b, dev\_c, numARows, numAColumns, numBRows, numBColumns);

// Copy result from device to host

// @TODO@ : complete copy from device to host

cudaMemcpy(host\_c, dev\_c, numARows\*numBColumns\*sizeof(float), cudaMemcpyDeviceToHost);

// Check result

check(host\_a,host\_b,host\_c,numARows, numAColumns, numBRows, numBColumns);

// Free device memory

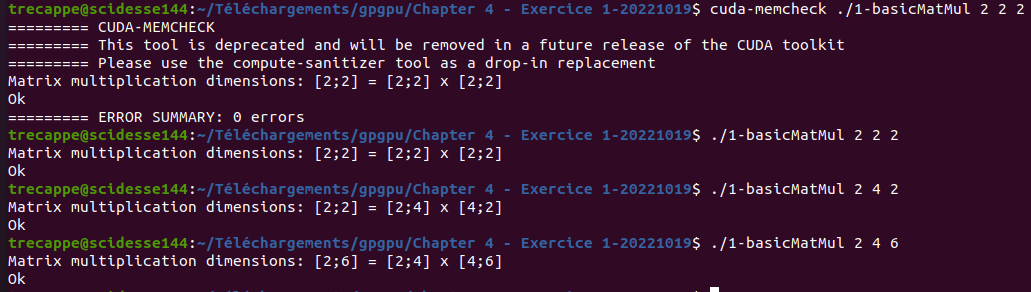
// @TODO@ : complete to deallocate memory

cudaFree(dev\_a); cudaFree(dev\_b); cudaFree(dev\_c);

return 0;

}

Fonctionne avec différente taille de matrice :



MTN : essayer de produire un autre algo/autre implémentation qui va améliorer cette intensité arithmétique (faut la faire augmenter) → résultat plus vite calculé

# Exercice 2

## How many floating operations are being performed in your matrix multiply kernel ?

Si deux matrices : A (n\*m) et B (m\*k)

Nombre d’opération flottante: 2\*width^3

**Ici nous sommes avec deux matrice de tailles width \* width, dont les opérations vont s’effectuer “tuile par tuile”, avec les tuiles de taille TILESIZE \* TILESIZE.**

**Donc un nombre d’opération égale à : 2\*width^3**

## How many global memory reads are being performed by your kernel ?

threads synchroniser

Accès en lecture (Al) : 2\*width\*width\*width

**Nombre d’accès mémoire entre deux matrices de taille width\*width :**

**2\*(width)^3**

## How many global memory writes are being performed by your kernel ?

Accès en écriture (Ar) : (taille matrice résultat) width\*width

**Nombre d’accès en écriture (Ar) : width \* width**

## Compute the arithmetic intensity of your kernel. The arithmetic intensity is a FLOP/Byte number standing for the number of floating point operations performed per byte of global memory access.

Intensité arithmétique : rapport entre le nombre d’opération en virgule flottante et la quantité de mémoire qui est nécessaire à faire ces opérations

→ Nbr op/(Al+Ar)[\*4 (car en float et faut mettre en bytes (si double précision faut mettre 8))]

une opération arithmétique a besoin de 0.0625 bytes de données

## Compare with the arithmetic intensity of the basicMatMul kernel. Explain why, at same matrices dimensions, the tiled version is better than the basic one. Get the kernels execution time using NVIDIA profiler nsys.

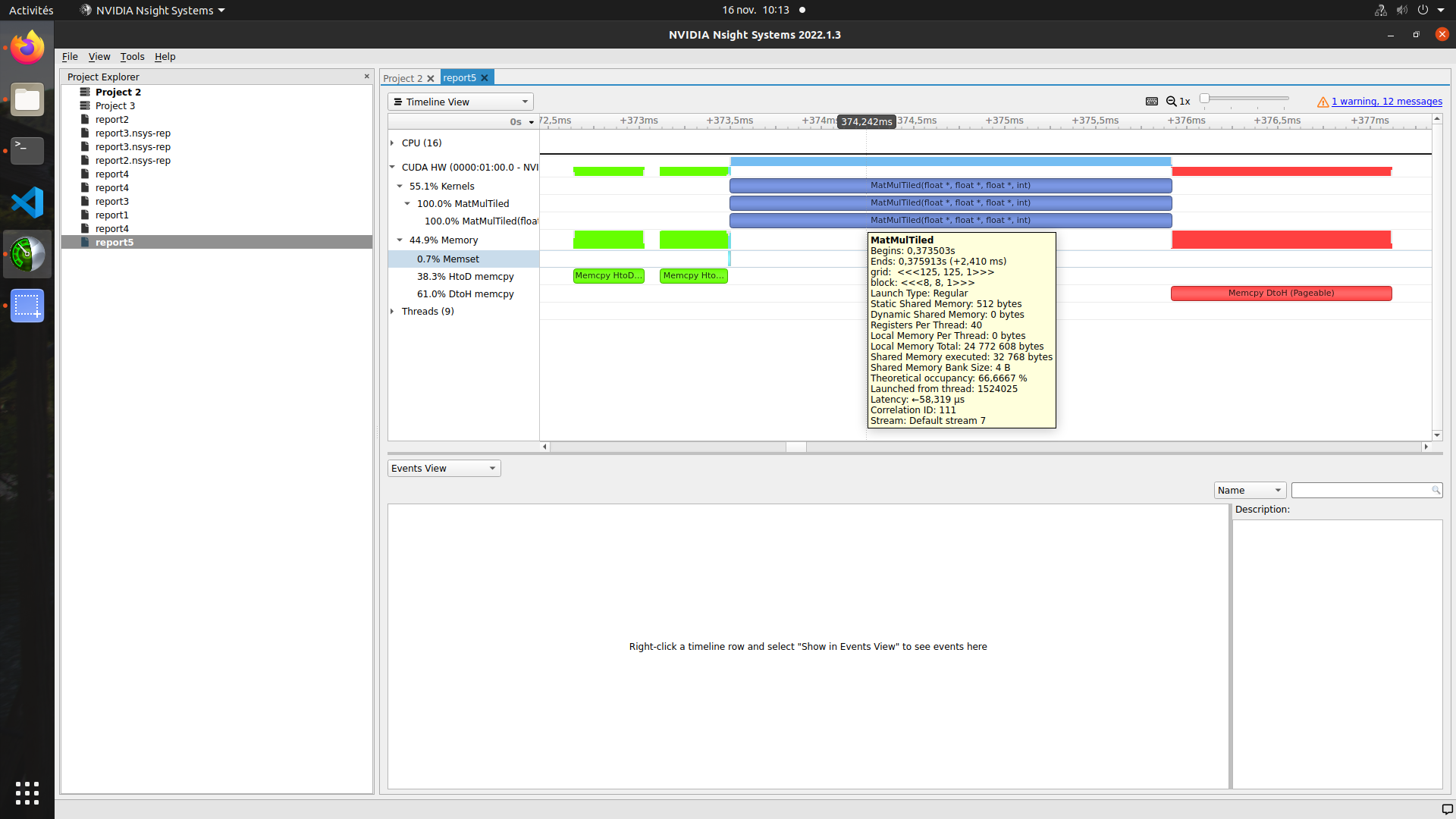
## 

## Use the occupancy calculator to compute the occupancy for TILE\_SIZE equals to 8, 16 and 32. Which size gives the best computational time ?

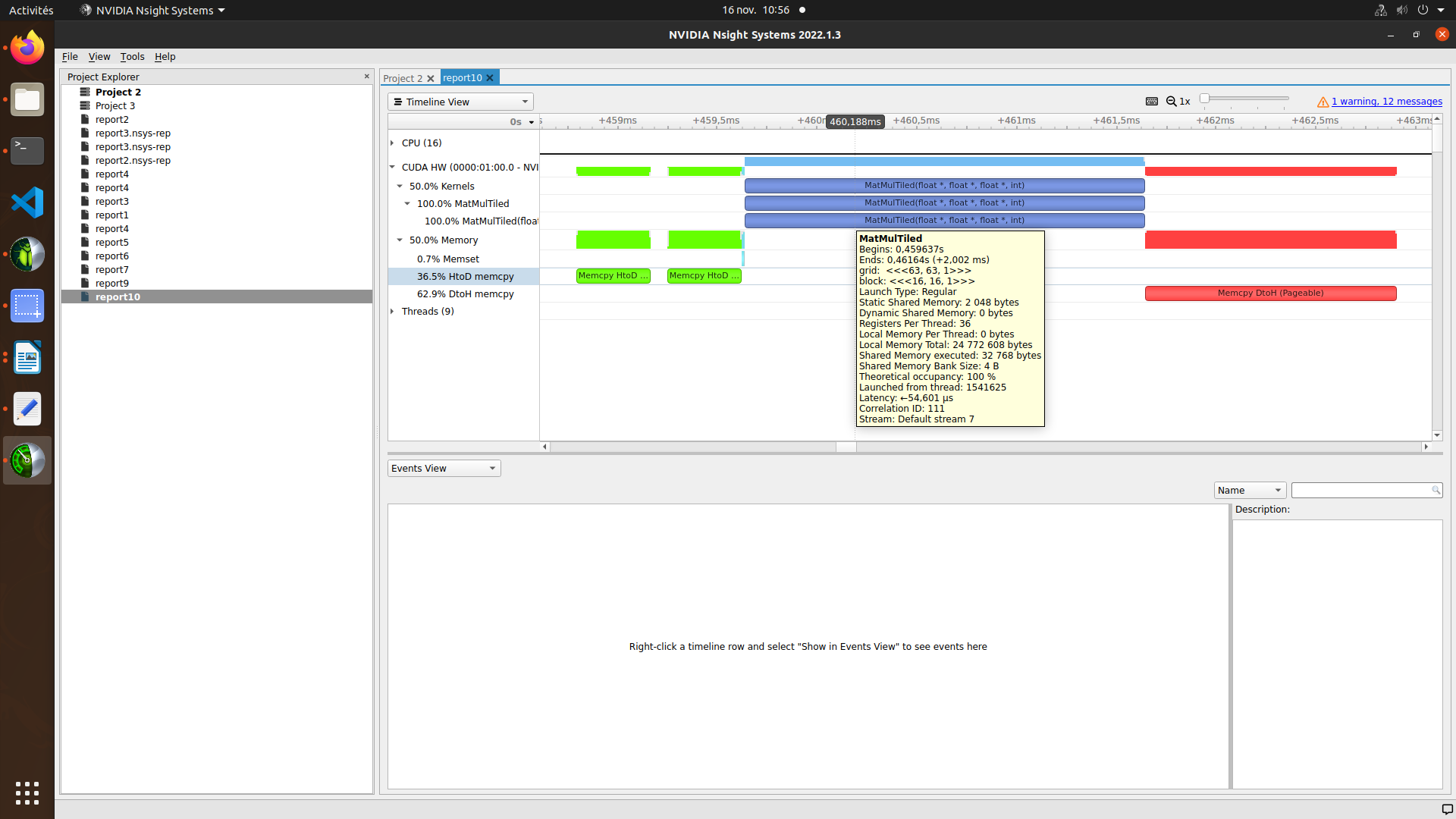
peut pas savoir

fais en cours 👍

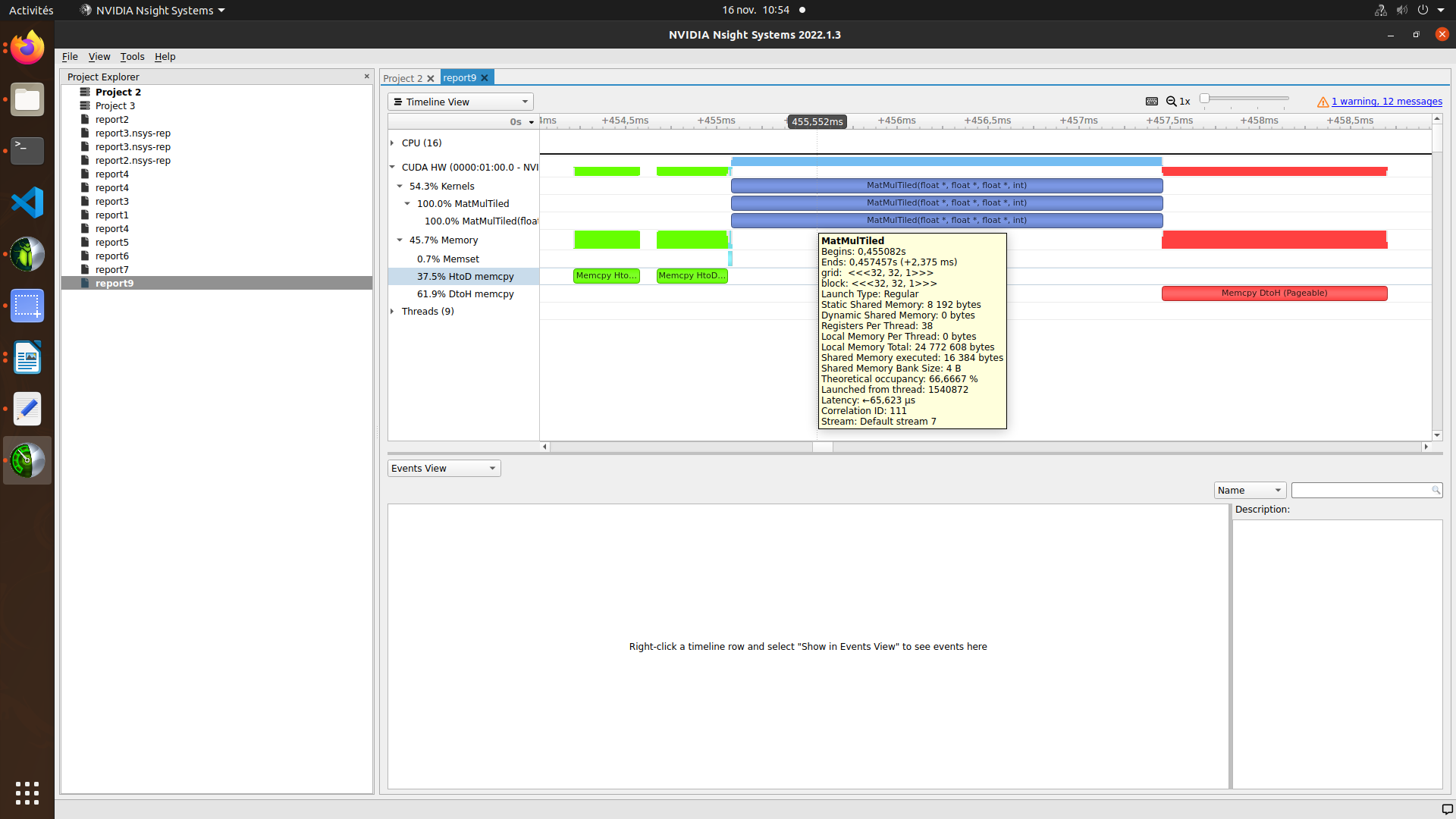
### TILESIZE = 8 && DIMBLOCK 8X8



### TILESIZE = 16 && DIMBLOCK 16X16



### TILESIZE = 32 && DIMBLOCK 32X32



→ prend plus de temps que pour 16 tuiles (contredit les propos en cours qui dit que l’intensité arithmétique augmente autant que le nombre de tuiles/threads)

## Suppose you have matrices with dimensions bigger than the max thread dimensions. Sketch an algorithm that would perform a matrix multiplication algorithm that would perform the multiplication in this case.

Mon algorithme règle déjà ce problème.

Dans mon algorithme, la taille des threads est de 8\*8\*1 donc 64.

## Suppose you have matrices that would not fit in global memory. Sketch an algorithm that would perform a matrix multiplication algorithm that would perform the multiplication out of place.