

UNIVERSITAT POLITÈCNICA DE CATALUNYA

---

# Projecte TGA

---

## INFORME

### TARGETES GRÀFIQUES I ACCELERADORS

#### *Autors*

DAVID LATORRE  
OSCAR DANIEL SANS



# Índex

|  |    |
|--|----|
| - Introducció: Binarització Nick.....                      | 3  |
| - Funcionament de l'algorisme de Nick.....                 | 5  |
| - Aplicació gràfica.....                                   | 8  |
| - Opcions de compilació importants.....                    | 9  |
| - Requisits i instruccions d'execució.....                 | 11 |
| - Implementació CPU.....                                   | 15 |
| - Implementació Cuda. Codi no Kernel.....                  | 18 |
| - Implementació principal Kernel GPU (primer mètode).....  | 21 |
| - Implementació segon mètode Kernel GPU.....               | 21 |
| - Implementació tercer mètode Kernel GPU.....              | 22 |
| - Anàlisi.....   | 23 |
| - Prestacions del sistema utilitzat.....                   | 23 |
| - Comparació CPU vs Kernels GPU.....                       | 25 |
| - Comparació Kernels GPU.....                              | 28 |
| - Comparació 16x16 threads/block, 32x32 threads/block..... | 30 |
| - Comparació kernel mètode 1, kernel mètode 3.....         | 36 |
| - Anàlisi altres crides CUDA.....                          | 39 |

## Introducció: Binarització Nick

Avui dia, en el món de la Visualització per Computador, la binarització d'imatges pren un paper molt important. És especialment necessari el procés de binarització per a molts camps dins d'aquesta disciplina, com per exemple pel tractament de documents i imatges degradades. Però, sobretot, és crucial perquè és una manera de passar-li informació als nostres ordinadors d'una manera que es pugui tractar i manipular.

Gràcies a la binarització som capaços d'identificar objectes en una imatge, som capaços de saber què hi ha en una imatge i, conseqüentment, som capaços d'aplicar tècniques del camp de la intel·ligència artificial per fer meravelles com per exemple un sistema d'identificació facial. Sense l'aplicació del procés de binarització, moltes coses ens resultarien quasi impossibles de fer, a més que serien molt més ineficients. Per exemple, seria molt més difícil realitzar un programa escanejador de textos (és a dir, que reconegui caràcters a partir d'una imatge que enfoca a un text) en temps real.

Ara bé, binaritzar correctament una imatge també pot resultar un procés molt tediós. Necessitem trobar una forma en què puguem separar tots els nivells de gris (o color) d'una imatge en tan sols de dos tipus, blanc o negre, cert o fals, de forma que a la imatge s'identifiqui, o es diferencii exactament el que volem. I, exactament això és el que fa que trobar un mètode de binarització que funcioni correctament per a la majoria d'imatges, per la majoria d'usos, resulti quasi impossible. No és el mateix, per exemple, binaritzar una imatge que representa un document d'informació, on tot i que la il·luminació de la imatge no estigui ben balancejada segurament no resultarà un problema, que binaritzar una imatge macromètrica per diferenciar molècules cancerígenes de les que no ho són. En aquest darrer cas, segurament necessitarem aplicar bastantes aplicacions morfològiques per obtenir resultats que es puguin considerar bons.

És per tot això que la binarització, en la majoria de casos, és un procés que utilitza bastants recursos del sistema. La manera més simple d'aconseguir binaritzar una imatge és mitjançant “thresholding” (llindars). És a dir, aquells

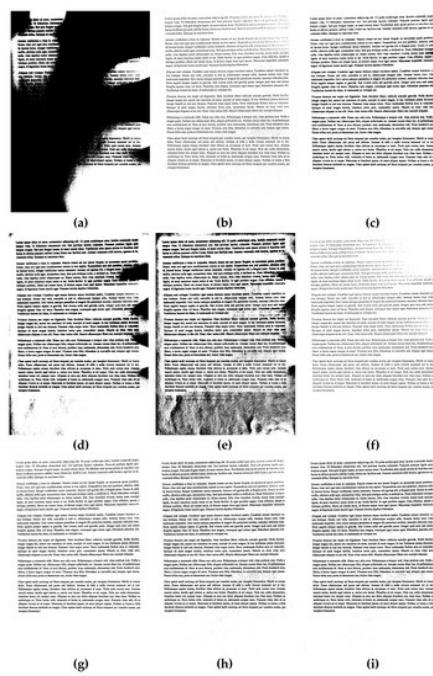
pixels els quals tinguin una instensitat major a aquest thresholding els marquem com a blancs, per diferenciar-los dels que tinguin una intensitat menor al thresholding (que els marcarem com a negres).

Hi ha molts algorismes que fan servir “thresholding”. Dins d'aquests podem diferenciar dos tipus: els globals, i els locals.

Per una part, els algorismes de binarització global són els més ràpids, els més simples, i per tant els que proporcionen resultats més pobres. Per a aquests, un sol valor de threshold és assignat per a tota la imatge. En canvi, per als algorismes de binarització local, per a cada pixel de la imatge és calcula un valor threshold diferent. Això, produceix que la binarització local sigui molt més costosa que la binarització global. No obstant, per contrast, algorismes de binarització local com: “Bernsen”, “Niblack”, “Sauvola”... Són capaços de realitzar binaritzacions correctes en fins i tot imatges on la il·luminació no es uniforme.

Un d'aquests algorismes de binarització local, el qual és bastant bo, i destaca pels seus bons resultats respecte molts altres, és l'algorisme de Nick.

Això ho podem observar, per exemple a la següent imatge:



On: (a) Otsu, (b) Niblack, (c) Sauvola, (d) Bradley (mean), (e) Bernsen, (f) Meanthresh, (g) **NICK**..

## Funcionament de l'algorisme de Nick

Suposem que tenim una imatge a nivell de grisos, és a dir amb un canal d'informació, i de 8 bits, és a dir amb intensitats de 0 a 255. Llavors, l'objectiu és, per a cada pixel calcular un valor threshold ( $t(x,y)$ ), de la manera següent:

$$t(x,y) = m(x,y) + K \cdot \sqrt{(\sum P(x,y)^2 - m(x,y)^2)/NP} \quad (1)$$

En aquesta fórmula,  $m(x,y)$  fa referència al valor intensitat que és la mitjana de totes les intensitats dels píxels de la finestra local (finestra lliscant) que té com a centre el pixel del qual està calculant el threshold,  $NP$  és el nombre total de píxels de la finestra local, i  $P(x,y)$  és el valor intensitat d'un dels píxels de la finestra local.  $K$ , d'altre banda, és un paràmetre que acostuma a variar entre [-0.2,-0.1], donant més bons resultats per uns certs valors respecte d'altres segons el que vulguem binaritzar.

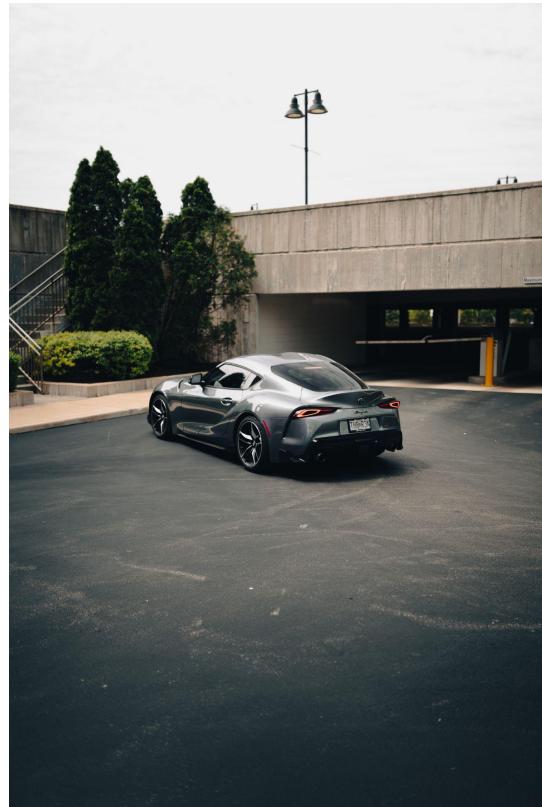
Un cop tinguem el valor threshold per a un pixel, si el valor intensitat del píxel és major al threshold llavors binaritzarem el píxel amb 1, sinó amb 0.

Computar el valor de tots els thresholds, i per tant el temps de computació final per a aquest algorisme és bastant gran, resultant en una complexitat:

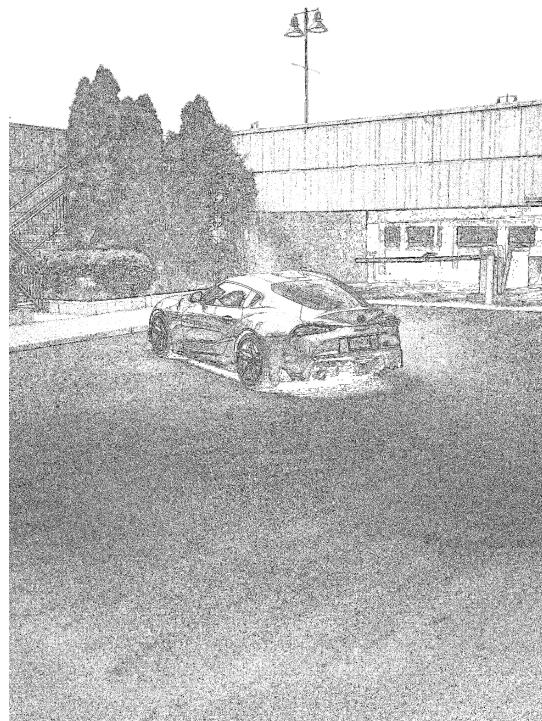
$$O(W^2N^2)$$

On  $W^2$  és el tamany de l'amplada (o altura, és igual ja que la finestra és un quadrat) en píxels de la finestra local, i  $N^2$  és el valor en pixeles de amplada (o alçada) de la imatge, suposant que la imatge és un quadrat (sinó seria width \* height). Llògicament, com més gran sigui el valor de la finestra local, resultats més bons s'aconseguiran.

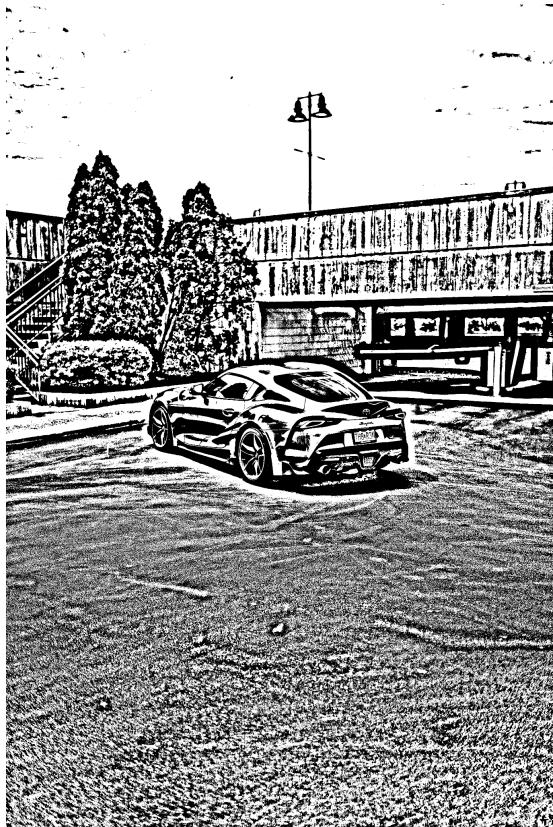
Per exemple, si volem binaritzar la següent imatge:



Si la binaritzem amb  $k = -0.01$ , i una amplada de finestra de 2 pixels amb l'aplicació que hem desenvolupat en aquest projecte, obtindrem el següent resultat:

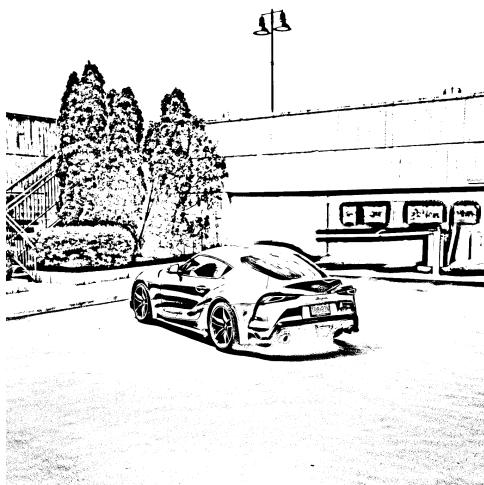


En canvi, si ho fem amb  $k = -0.01$ , però amb una amplada 100 pixels de finestra, obtindrem el següent resultat:



Podem observar, per tant, que amb una finestra major els resultats queden més definits.

Ja que hem vist que passa si variem el tamany de la finestra, vegem que passa si mantenint l'amplada de finestra de 100 pixels, la calculem amb  $k = -0.2$ :



Vegem per tant, que es tracta senzillament d'una binarització diferent, ni millor ni pitjor.

Finalment, per acabar aquesta introducció, necessitem clarificar el per què resulta útil paral·lelitzar l'algorisme de Nick. Bé, això ja ho acabem de veure, degut a la seva gran complexitat això resulta en que quan tenim imatges certament grans (com per exemple 10.000 x 10.000 píxels), si les volem binaritzar amb un grau més o menys bo de correctesa i per tant amb un tamany de finestra més o menys gran, fer-ho amb un algorisme seqüencial que només utilitzi un thread de CPU resultarà en un temps d'execució molt gran (que pot arribar a ser fins i tot superior a 1 minut, com veurem posteriorment), i per tant paral·lelitzar-lo resulta ser molt útil.

És més, pel fet que és un algorisme de binarització local, per calcular el threshold de cada pixel utilitza una finestra lliscant, i paral·lelitzar això en una GPU resulta ser molt eficaç i no gaire difícil.

## **Aplicació gràfica**

Per al desenvolupament d'aquest projecte hem decidit fer un projecte amb interfície gràfica, que funcioni sobre el sistema operatiu Windows 10 (o superior...). Tot i que pel fet de implementar la interfície gràfica ho hem fet amb llibreries i dependències de Qt, compilat de forma correcta també pot funcionar amb Linux.

Hem volgut dissenyar per tant una petita espècie d'editor d'imatges, on puguem binaritzar la imatge amb el mètode que hem explicat a la introducció, poguem visualitzar tant la imatge original com els resultats, així com guardar la imatge resultat, i també observar informació sobre tots els diversos recursos consumits pel sistema.

La estructura d'aquest programa, per tant, consisteix en:

- Arxiu **main.cpp**. El qual s'encarrega d'iniciar una instància de la aplicació gràfica.
- (mainwindow.h i mainwindow.cpp), (MyGraphicsView.h i MyGraphicsView.cpp) i (ui\_mainwindow.h i mainwindow.ui): Els quals són arxius i classes que s'encarreguen de la interfície gràfica del programa. Per una part, a l'arxiu mainwindow.ui tenim definida l'estructura principal visual de l'aplicació. A MyGraphicsView hi ha programat el funcionament del visualitzador d'imatges, i a mainwindow tenim programat tot el funcionament de tots els components de l'interfície gràfica, així com l'administració de signals i slots dels components.
- (processor.h i processor.cpp): Tal com diu el nom és la classe que és el processador del programa. És per tant, la classe encarregada de la administració i computació de la binarització de les imatges.
- NickCUDA.cu: És tracta de l'arxiu que conté tot el codi escrit en CUDA. Cada cop que es vulgui executar una binarització amb CUDA, processor cridarà a la funció que toqui definida dins de l'arxiu NickCUDA.cu, i aquesta funció, entre d'altres coses, cridarà al corresponent kernel que s'ha d'executar, també definit a l'arxiu NickCUDA.cu.

## Opcions de compilació importants

Tal com es pot observar per la estructura del nostre programa, necessitem fer per tant dos compilacions per separat. D'una banda, necessitarem compilar tots aquells arxius escrits purament en C++, linkejant-los amb les corresponents dependències de Qt (llibreries i dlls de Qt), i d'altra banda compilar NickCuda.cu amb nvcc, linkejant-lo amb les corresponents dependències de CUDA (llibreries i dlls de Qt).

La forma per la qual hem aconseguit poder comunicar aquestes dues parts compilades de forma separada, és gràcies a dos coses principals:

- D'una banda, l'ús de la gramàtica:

```
extern "C"
```

La qual ens permet definir funcions “externes”, és a dir funcions que no es troben dins de l’espai dels arxius que s'estan compilant, sinó en fitxers objectes exteriors. D'aquesta forma, per tant, podem compilar l'arxiu CUDA per separat, declarant amb “extern “C”” les funcions que seran cridades desde l'exterior (és a dir, per l'arxiu processor.chapp), i a l'arxiu processor.cpp declarar tots aquells externs que cridarem de la manera següent:

```
- extern "C" string NICKGPUMethod(const float* grayscaledImage, int tamanyFinestra,  

float k, int width, int height, QTextBrowser * outputDisplay, string  

fileOUTGPUMETHOD1NICK);  
  

- extern "C" string NICKGPUMethod2(const float* grayscaledImage, int tamanyFinestra,  

float k, int width, int height, QTextBrowser * outputDisplay, string  

fileOUTGPUMETHOD2NICK);  
  

- extern "C" string NICKGPUMethod3(const float* grayscaledImage, int tamanyFinestra,  

float k, int width, int height, QTextBrowser * outputDisplay, string  

fileOUTGPUMETHOD3NICK);
```

- I d'altra banda, tot això també ho hem aconseguit fer gràcies a la creació d'una solució de Visual Studio 2022, que ens ha permès establir tots els linkatges correctament.

Gràcies a Visual Studio 2022, també hem pogut establir fàcilment certes opcions de compilació tant per al compilador C++ de Windows, com per al compilador NVCC. Entre aquestes opcions, considerem important destacar:

- **/Ox**, tant per a msvc c++ com per a nvcc: Tot i que no ho sembli, establir la opció d'optimització màxima fa que els temps que es triga a guardar o carregar una imatge siguin molt més inferiors, entre d'altres millors temporals.

- **compute\_80,sm\_80** Code Generation: Necessari per establir l'arquitectura de compilació. En el nostre cas és almenys “80”, ja que utilitzem una GPU de la generació Ampere (més endavant es detallarà més informació sobre aquesta GPU a l'apartat d'anàlisi).
- **-use\_fast\_math:** Es tracta potser de la opció de compilació més important de nvcc. Gràcies a aquesta opció sacrificuem una mica de presició a les variables float (la qual ni notarem) per un gran guany en el temps d'execució dels kernels.
- C++ 20.

## Requisits i instruccions d'execució

Degut a que el procés de compilació i linkatge per a aquest programa és un procés complicat, a més que és necessiten un munt extra de llibreries i dependències, en cas que es vulgui executar el programa directament, sense modificar res de codi, es proporciona a l'entregable directament una carpeta **binari**, la qual conté ja compilat el programa per a una màquina 64 bits Windows 10 o superior, la qual ja conté dins del directori les dependències de Qt incloses. Els requisits per executar el binari, doncs, són les següents:

- Windows 10 o superior.
- Sistema amb una GPU d'Nvidia amb capacitat de computació CUDA de la generació **Maxwell o superior** (ja que el binari ha sigut compilat amb la opció **compute\_52,sm\_52**). Si es vol utilitzar una GPU de generació inferior, per tant, caldrà recompilar el codi correctament.
- Nvidia CUDA Toolkit ([link](#)) instal·lat, i establert el PATH del sistema correctament (variable **CUDA\_PATH** així com d'altres establertes automàticament en el procés d'instal·lació del Toolkit).

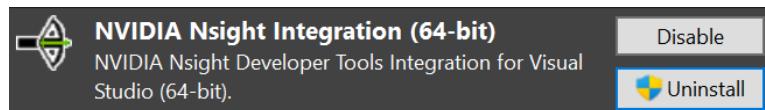
Si es compleixen totes aquestes condicions, es podrà executar correctament el binari. Per executar-lo, senzillament cal executar el **.exe** dins de la carpeta **binari**.

Si el que es vol és modificar el codi font, o canviar opcions de compilació, llavors caldrà recompilar el codi. Això es pot fer mitjançant la solució de Visual Studio 2022 inclosa en la carpeta **solution** dins de l'entregable del projecte (l'arxiu solució és el que té la extensió **.sln**, aquest s'ha d'executar amb Visual Studio). Els requisits, per tant, per l'edició i compilació de la solució en aquest projecte són:

- Tots els anteriors.
- Visual Studio 2022 o superior.
- **Qt 5.15** ([link](#)).
- Extensió de Visual Studio: Qt Visual Studio Tools:

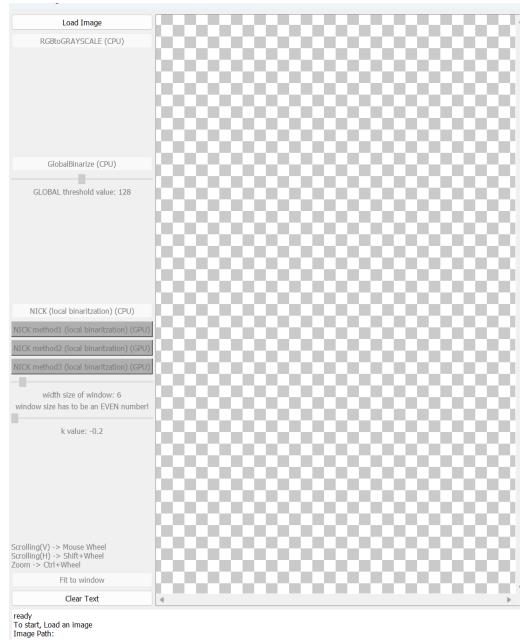


- Opcionalment, per compilar d'una manera fàcil el codi CUDA, les noves eines de profiling de Nvidia, Nsight Compute ([link](#)) i Nsight Systems ([link](#)), així com l'extensió Nsight per a Visual Studio:



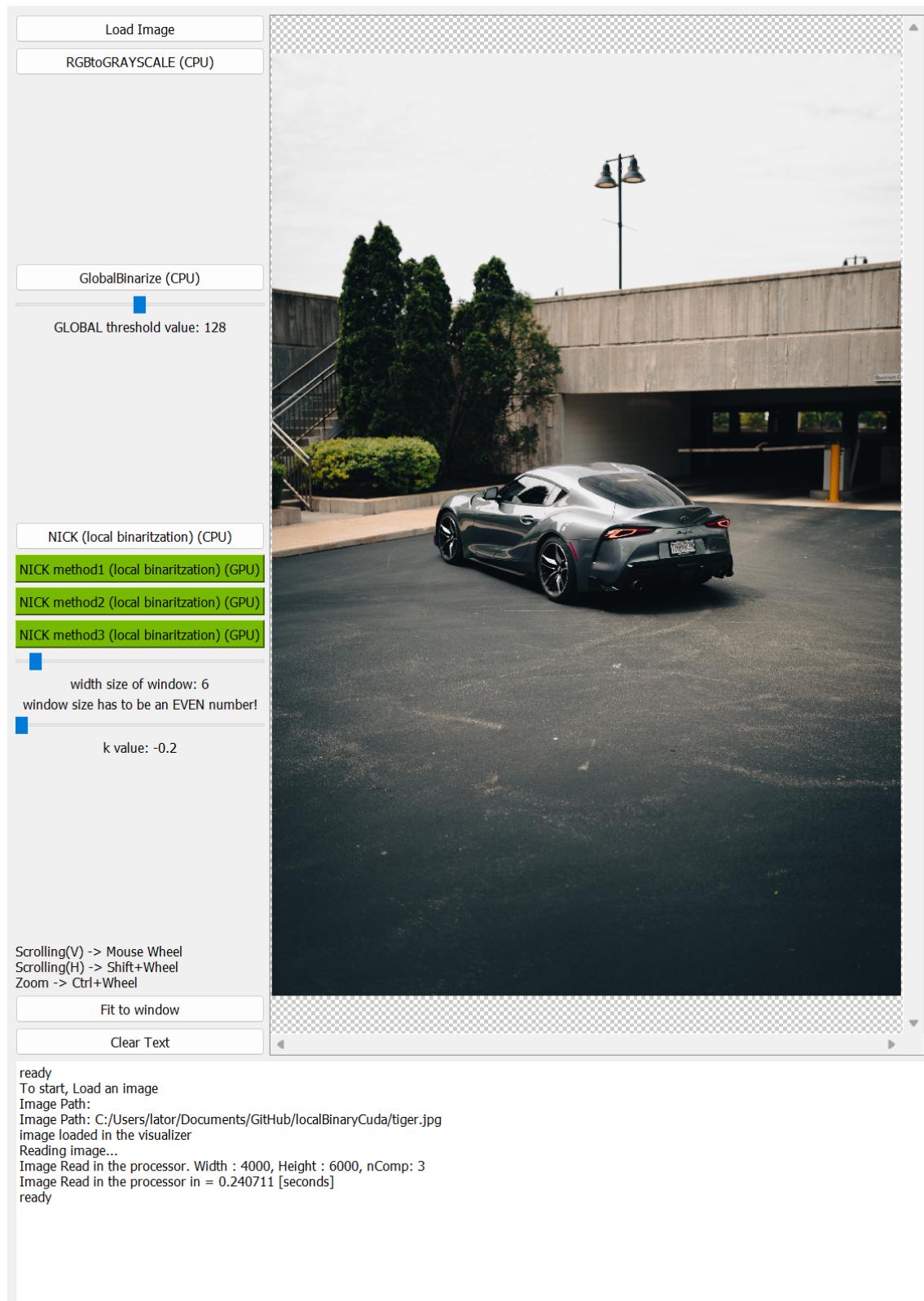
- Compiler msvc C++ 20 o superior.
- Qt al Path de Windows.

Explicat això, ara anem a veure en que consisteix la interfície gràfica de l'aplicació:



Com podem observar, distingim tres parts principals. D'una banda tenim la part superior dreta, on hi podem visualitzar la imatge que obrim, i com queda un cop la binaritzem amb l'opció corresponent.

A continuació, a la part esquerre superior, tenim les diverses opcions que ens dona el programa. Primerament, per a triar una, caldrà obrir una imatge:



Un cop hem obert una imatge, podem triar a fer tres coses principals. Les dos primeres són bàsiques, i no cal explicació sobre els seus algorismes degut a la seva simplicitat. Es tracta del botó RGBtoGRAYSCALE, el qual ens convertirà la imatge que hem carregat a escala de grisos, és a dir a un canal de color, i del botó GlobalBinarize, el qual ens binaritzarà la imatge globalment segons el threshold especificat al slider inferior del botó.

La implementació d'aquestes dues opcions és simple, ràpida, i no és l'objectiu principal d'aquest projecte. És per això que no hi farem cap implementació CUDA per a aquestes opcions (no tindria sentit). A més, que l'unic propòsit d'aquestes opcions és de comparació visual respecte a la binarització de Nick.

A continuació, ens trobem amb els botons encarregats de binaritzar la imatge mitjançant el mètode Nick. Tenim una implementació CPU, i tres implementacions CUDA (marcades en verd). tres mètodes que seran explicats posteriorment. Els valors possibles a triar de l'algorisme de Nick son l'amplada de la finestra i el k value, els quals es poden triar amb els dos sliders posteriors.

Finalment, a la part inferior de la interfície visual tenim una petita finestra informativa, la qual ens anirà informant detalladament sobre tots els temps d'execució de les parts de cada algorisme (ja sigui de CPU o GPU), així com d'altre informació important.

Per exemple, si executem una binarització local Nick CPU amb una amplada de finestra de 10 pixels i k -0.01, amb les característiques del sistema que es detallen a l'apartat anàlisi, obtindrem uns resultats com els següents:

```
NICK CPU LOCAL BINARITZATION...
windowSize = 10 x 10
k = -0.01
ALLOCATING SPACE FOR NEW IMAGE...
ALLOCATED SPACE IN = 2.1e-05 [seconds]
CONVERTING TO GRayscale...
CONVERTED TO GRayscale IN = 0.03366 [seconds]
BINARITZATING...
BINARITZATED IN = 1.629317 [seconds]
WRITING IMAGE...
IMAGE WRITTEN IN = 0.866775 [seconds]
Nick cpu locally Binarized image saved in:
C:/Users/lator/Documents/GitHub/localBinaryCuda/tigerNICKCPUBINARITZATED
.png
```

```
Opening Nick cpu Binarized image in the visualizer
Nick CPU binarized image opened in the visualizer
ready
```

En canvi, si executem una binarització local GPU primer mètode amb una amplada de finestra de 10 pixels i k -0.01, obtindrem uns resultats com els següents:

```
NICK GPU METHOD1 LOCAL BINARITZATION (only global memory, transactions
between CPU-GPU with pinned memory)...
windowSize = 10 x 10
k = -0.01
ALLOCATING SPACE FOR GRAYSCALED IMAGE ON CPU...
ALLOCATED SPACE FOR GRAYSCALED IMAGE ON CPU IN = 1.4e-05 [seconds]
CONVERTING TO GRayscale (CPU)...
CONVERTED TO GRayscale (CPU) IN = 0.049319 [seconds]
NOW DOING GPU STUFF...
GPU computation done:
TIME SPENT ALLOCATING AND COPYING INTO GPU = 0.028906 [seconds]
TIME SPENT IN THE GPU KERNEL = 0.026076 [seconds]
TIME SPENT COPYING DATA FROM GPU TO CPU = 0.031994 [seconds]
CONVERTING IMAGE FLOAT POINTER TO CHAR POINTER TO WRITE THE IMAGE
(CPU)...
CONVERTED FLOAT POINTER TO CHAR POINTER IN(CPU) = 0.052737 [seconds]
WRITING IMAGE...
IMAGE WRITTEN IN = 0.882621 [seconds]
Nick gpu method1 image saved in:
C:/Users/lator/Documents/GitHub/localBinaryCuda/tigerNICKGPUMETHOD1BINAR
ITZATED.png
Opening Nick GPU method1 Binarized image in the visualizer
Nick GPU method1 binarized image opened in the visualizer
ready
```

Destacar també que, tal com s'indica en aquesta finestra, per a cada binarització que fem, el resultat es guardarà automàticament al directori on es troba la imatge original.

## Implementació CPU

Primer de tot, hem implementat el codi de l'algorisme Nick dissenyat per a correr directament en una CPU, d'aquesta manera hem assimilat les bases de la binarització amb aquest algorisme i també hem format les bases de l'estructura del nostre programa.

Els punts principals que formen la implementació a la CPU son la càrrega de la imatge original, el pas de RGB a escala de grisos i finalment la binarització per a procesar la imatge de sortida final, tots aquests amb aquest ordre.

La càrrega de la imatge original es fa a través de la llibreria stb, que té una funció que retorna un vector amb els valors RGB de la imatge, sent la posició 0 el valor Red del pixel 1, la posició 1 el valor Green del pixel 1, etc.

El pas de RGB a escala de grisos es realitza a través d'una fórmula que entrega bons resultats per a realitzar aquesta tasca, on multipliquem cada valor RGB de cada pixel per una constant. D'aquesta manera obtenim el valor en la escala de gris que busquem.

$$imgGray = 0.2989 * R + 0.5870 * G + 0.1140 * B$$

Una vegada tenim la imatge en escala de grisos emmagatzemada en un vector, ja podem transformarla en blanc o negre, utilitzant l'algoritme de Nick. La nostra implementació de l'algoritme és, per a cada pixel, calcular la seva finestra. Llavors per a cada finestra utilitzem la fórmula per calcular el valor del píxel final, és a dir, calculem el seu valor threshold, i una vegada comparat, especificarem un 0 (negre) o un 255 (blanc) al punter que representa la imatge final. Cal destacar que per eficiència espacial el punter que representa la imatge final seria millor que fos un punter per exemple de tipus “bool” en comptes de “char”, no obstant necessitem declarar-lo com a “char” ja que la funció que guarda la imatge, `stbi_write_png`, necessita com a paràmetre un punter “char”.

Observem una mica per sobre el codi de l'algorisme que binaritza la imatge un cop ja hem transformat la imatge a escala de grisos:

```

for (int p = 0; p < width * height; p++)
{
    int row = p / width;
    int col = p % width;

    //bordes de la finestra lliscant
    int beginrow = max(0, row - tamanyMEITATFinestra);
    int begincolumn = max(0, col - tamanyMEITATFinestra);
    int endrow = min(height - 1, row + tamanyMEITATFinestra);
    int endcolumn = min(width - 1, col + tamanyMEITATFinestra);

    //calcular el pixel actual
    int numeropixelsfinestra = (endrow - beginrow + 1) * (endcolumn - begincolumn + 1);

    if (row < height && col < width)
    {
        unsigned char temp;
        int Total_sum = 0;
        int Total_sum_pow2 = 0;
        for (int i = beginrow; i <= endrow; i = i + 1)
            for (int j = begincolumn; j <= endcolumn; j = j + 1)
            {
                temp = imageOUT[i * width + j];
                //printf("%f \n", temp);
                Total_sum = Total_sum + temp;
                Total_sum_pow2 = Total_sum_pow2 + (temp * temp);
            }

        //printf("%f \n", Total_sum);
        float mean = Total_sum / (numeropixelsfinestra*1.0f);
        float Threshold = mean + k * sqrtf((Total_sum_pow2 - mean * mean) / (numeropixelsfinestra*1.0f));

        if (Threshold < imageOUT[row * width + col])
        {
            imageFinalOUT[row * width + col] = 255;
            //printf("Yes \n");
        }
        else
        {
            imageFinalOUT[row * width + col] = 0;
            //printf("No \n");
        }
    }
}

```

Observem que tenim un bucle principal, el qual itera per cada pixel de la imatge, i per a cada pixel calcula el seu valor binaritzat. La manera per la qual fa això es, primerament calculant els extrems de la finestra local del pixel (es podria donar el cas en que el pixel estigués a prop d'algun extrem de la imatge i per tant la finestra local fos més petita), i un cop fet això, s'itera per cada pixel de la finestra local per calcular els paràmetres necessaris de la fórmula de threshold (mitjana d'intensitats dels pixels, suma del quadrat de les intensitats dels pixels...). Un cop ja tenim tota la informació necessària, calculem el threshold i establim si binaritzar com a blanc o negre el pixel.

## Implementació Cuda. Codi no Kernel

Per a la implementació CUDA de l'algorisme de Nick hem realitzat tres kernels diferents, començant des d'un simple, i anant millorarnt-lo de forma que creiem que serà més eficient (després veurem si realment ho és o no a l'apartat d'anàlisi) fins arribar al tercer.

Però, abans d'explicar la implementació dels kernels, necessitem explicar la implementació de les funcions, també escriptes a l'arxiu CUDA, que criden a aquests kernels. Hi tenim tres funcions, una per cada mètode de binarització que utilitza la GPU, és a dir, una per cada kernel diferent.

Cada cop que es premi un dels botons a la interície gràfica de binarització GPU, s'executarà la funció corresponent de la classe “processor”, la qual passarà la imatge a escala de grisos, i convertirà aquesta en un punter de char a float, per tal de passar-li aquest punter (junt amb d'altres paràmetres necessàris, com el tamany de la finestra, k...) a la funció corresponent de l'arxiu NickCUDA.cu.

La qüestió per la qual passem la imatge en un punter de floats, és perquè, tot i que els valors que estiguem tractant només vagin en el rang de 0-255, una targeta gràfica està específicament construïda per fer operacions en coma flotant, d'aquesta forma el temps d'execució es pot reduir considerablement.

La funció corresponent de l'arxiu NickCUDA.cu s'encarregarà de fer les operacions necessàries, i, finalment, d'escriure la imatge al disc. No obstant, tot i que cada una d'aquestes funcions cridi a un kernel diferent, la seva composició és la mateixa. Expliquem-la una mica:

```
float millisecondsMemoryEvent = 0;
float millisecondsKernelEvent = 0;
float millisecondsMemoryBackEvent = 0;
dim3 dimGrid, dimBlock;

dimBlock.x = BLOCKSIZE;
dimBlock.y = BLOCKSIZE;
dimBlock.z = 1;
// + BLOCKSIZE necessari pels pixels que queden
dimGrid.x = (width + BLOCKSIZE - 1) / BLOCKSIZE;
dimGrid.y = (height + BLOCKSIZE - 1) / BLOCKSIZE;
dimGrid.z = 1;
```

```
cudaEvent_t startMemoryEvent, StopMemoryEvent, startKernelEvent, StopKernelEvent, startMemo
```

El primer que ens trobem en aquestes funcions és la declaració de variables. Per una part, destacar que per mesurar els diferents temps d’execució de la binarització utilitzem els anomenats “cudaEvent”, els quals són especialment útils perquè ens permeten mesurar temps d’execució de la GPU sense tenir que sincronitzar aquesta amb la CPU.

També tenim el càlcul de les dimensions de block i de grid. En els tres casos, el càlcul és el mateix, ja que per als tres kernels fem que un thread calculi un pixel de la imatge binaritzada. Pel que fa a les dimensions de block, però, com es podrà veure a l’anàlisi, anirem provant entre dos valors: 32x32 threads/block, i 16x16 threads/block, amb l’objectiu de veure quin proporciona més rendiment.

I, pel que fa a les dimensions del grid, el tamany serà el corresponent segons la dimensió del bloc triada per tal de que hi hagi un thread per cada pixel de la imatge.

```
cudaMallocHost((float**)&FinalImageHost, width * height * sizeof(float));
cudaMallocHost((float**)&grayscaledImagePinned, width * height * sizeof(float));
memcpy(grayscaledImagePinned, grayscaledImage, width * height * sizeof(float));

//test_kernel << <1, 1 >> > () ;
cudaEvent_t startMemoryEvent, StopMemoryEvent, startKernelEvent, StopKernelEvent, startMemoryBackEvent, StopMemoryBackEvent;

//Allocating and copia memoria a la gpu
cudaEventCreate(&startMemoryEvent);
cudaEventCreate(&StopMemoryEvent);
cudaEventRecord(startMemoryEvent);
float* grayscaledImageDevice;
float* FinalImageDevice;
cudaMalloc((float**)&grayscaledImageDevice, width*height*sizeof(float));
cudaMalloc((float**)&FinalImageDevice, width * height * sizeof(float));
cudaMemcpy(grayscaledImageDevice, grayscaledImagePinned, width * height * sizeof(float), cudaMemcpyHostToDevice);
cudaEventRecord(StopMemoryEvent);
```

A continuació, ens trobem amb la fase d’alocament de memòria i transferència de memòria de CPU a GPU. Com es pot observar a la imatge, abans de fer la transferència a GPU, fem un malloc a la CPU amb memòria pinned, amb l’objectiu que la transferència es faci de forma molt més ràpida.

A la memòria de GPU hi tindrem dos punters, un serà la imatge original en escala de grisos, i l'altre la imatge binaritzada.

```
//Execution of the kernel
cudaEventCreate(&startKernelEvent);
cudaEventCreate(&stopKernelEvent);
cudaEventRecord(startKernelEvent);
int tamanyMEITATFinestra = tamanyFinestra / 2;
NickKernelMethod1 << dimGrid, dimBlock >> (grayscaledImageDevice, FinalImageDevice, k, width, height, tamanyMEITATFinestra);
//cudaDeviceSynchronize(); // SA DE TREURE
cudaEventRecord(stopKernelEvent);
```

A continuació ens trobem amb la fase de crida al kernel. Aquesta fase no té més història, simplement cridem al kernel corresponent amb els paràmetres corresponents.

```
//Memory Back
cudaEventCreate(&startMemoryBackEvent);
cudaEventCreate(&stopMemoryBackEvent);
cudaEventRecord(startMemoryBackEvent);
cudaMemcpy(FinalImageHost, FinalImageDevice, width * height * sizeof(float), cudaMemcpyDeviceToHost);
cudaEventRecord(stopMemoryBackEvent);

//Free memory, events
cudaFree(grayscaledImageDevice);
cudaFree(FinalImageDevice);

cudaEventSynchronize(stopMemoryEvent);
cudaEventElapsedTime(&millisecondsMemoryEvent, startMemoryEvent, StopMemoryEvent);

cudaEventSynchronize(stopKernelEvent);
cudaEventElapsedTime(&millisecondsKernelEvent, startKernelEvent, StopKernelEvent);

cudaEventSynchronize(stopMemoryBackEvent);
cudaEventElapsedTime(&millisecondsMemoryBackEvent, startMemoryBackEvent, StopMemoryBackEvent);

cudaEventDestroy(startMemoryEvent);
cudaEventDestroy(stopMemoryEvent);
cudaEventDestroy(startKernelEvent);
cudaEventDestroy(stopKernelEvent);
cudaEventDestroy(startMemoryBackEvent);
cudaEventDestroy(stopMemoryBackEvent);
```

Un cop finalitzada l'execució del kernel, farem una transferència de la imatge binaritzada de GPU a CPU, i com que ja haurem acabat de fer tots els càlculs de GPU, ja podrem alliberar i destruir tots aquells objectes que tenim a la GPU.

Finalment, a part d'escriure els diversos temps d'execució a la interfície gràfica, ens caldrà passar en CPU el punter de floats que representa la imatge binaritzada a char, per tal de poder escriure la imatge.

## **Implementació principal Kernel GPU (primer mètode)**

Un cop ja hem explicat aquestes funcions, ja podem començar a explicar la implementació dels tres diversos kernels.

El primer de tot, és tracta del més simple. Diem que es tracta del més simple, perquè només utilitza la memòria global de la GPU. Compte, això no vol dir que sigui el més lent (com podrem observar a l'apartat anàlisi). Per tant, no utilitza altres tipus de memòria també presents a la GPU, com la memòria compartida.

L'únic mètode, per tant, de reduir accessos a la memòria global per al primer mètode, és la caché L<sub>1</sub> i L<sub>2</sub>.

En aquest primer mètode, com que cada thread és responsable de calcular només un sol pixel de la imatge, i la imatge original en escala de grisos ja la tenim a memòria global, això produeix que l'algorisme d'aquest primer kernel sigui casi el mateix que l'algorisme secuencial que hem implementat per a la CPU. La única diferència, però, lògicament és que no tenim el bucle principal que itera per tots els píxels de la imatge, sinó que per a cada thread és mapejarà el seu corresponent index fent-nos saber per tant quin pixel de la imatge representa.

En conclusió, per tant, cada thread de la GPU haurà de calcular el threshold del seu pixel, de forma que haurà d'iterar per tots els píxels de la finestra local que li pertoqui, produint un gran munt d'accessos a memòria global per thread (aquest munt variarà evidentment segons el tamany d'amplada de la finestra local).

## **Implementació segon mètode Kernel GPU**

Degut a que en el primer kernel cada thread fa molts accessos a memòria global, i per tant es pot pensar que el temps d'execució de l'algorisme be marcat per la latència de memòria global, la primera optimització que podem pensa en fer-li al primer kernel és la d'utilitzar altres sistemes de memòria que tingui la GPU,

sistemes de memòria més ràpids, com per exemple la memòria shared, que és memòria que es comparteix entre tots els threads d'un bloc.

És precisament la memòria shared la que implementem en aquest segon kernel. Per fer això, dividim la imatge original en tiles més petits, on cada bloc de threads càrrega un tile de la imatge en la seva memòria shared. Per fer això, per tant, cada bloc tindrà reservat de memòria shared tants bytes (\*4, ja que un float són 4 bytes) com threads tingui aquest.

Aquest fet, però, ens porta dos principals desavantatges. D'una banda, com que principalment per a cada bloc no hi tindrem res a la memòria shared, cada thread del bloc haurà d'accendir a la memòria global per mapejar el valor del seu pixel a la memòria shared. De forma que, per tant, cada thread abans de poder començar a calcular el seu threshold, s'haurà de sincronitzar amb tots els threads del seu bloc per assegurar-se que tota la memòria shared està carregada.

I, d'altra banda, el fet que per a cada bloc a la memòria shared només hi tinguem el valor dels pixels que representen els threads de la imatge, això produirà que igualment se segueixin fent molts accessos a memòria global per calcular els thresholds, degut al càlcul dels pixels de la finestra local. Sobretot pel càlcul d'aquells pixels que es trobin al marge del bloc.

De fet, si la finestra és molt gran, la utilització de la memòria shared vers la memòria global serà inútil, ja que casi tots els accessos els farem a la memòria global.

## Implementació tercer mètode Kernel GPU

El propòsit principal d'aquest kernel és d'intentar solucionar aquest últim problema que acabem de veure, de forma que, almenys per a finestres grans, s'utilitzin uns quants valors més de la memòria shared.

Per aconseguir això, i de forma que s'ajusti amb els límits de la memòria shared que tenim per bloc (això ho veurem a l'apartat següent), farem que a la memòria

shared de cada bloc no tan sols hi emmagatzemen els valors dels pixels del tile que al bloc li pertoca, sinó també els valors del seu tile esquerre, superior-esquerre, superior, superior-dret, dret, inferior dret, inferior, i inferior esquerre. És a dir dels 8 tiles adjacents.

Això, fa que a la memòria shared hi tinguem 9 vegades més valors de píxels, i que per tant hi poguem fer més accessos a aquesta. Per aconseguir això, però, cada pixel primerament haurà de carregar 9 valors a la memòria shared: el valor del seu pixel corresponent del tile del seu bloc, i dels 8 blocs adjacents.

Cal destacar, que per aconseguir portar tots aquests pixels a la memòria shared, cada thread haurà d'executar un seguit d'instruccions condicionals considerable, la qual podria afectar al temps d'execució.

Finalment, hem de dir que voliem implementar un quart kernel, el qual no tan sols utilitzés la memòria shared, sinó també memòria constant. No obstant, degut al tamany limitat de la memòria constant, això sen's a fet impossible d'implementar.

## Anàlisi

Comencem l'apartat d'anàlisi, on farem un seguit de comparacions i anàlisis sobre com es comporten els algorismes que hem implementat quan els comparem entre ells, i en quines coses hi falla cada un en termes d'eficiència.

### Prestacions del sistema utilitzat

Abans, però, de començar amb les comparacions, és important explicar amb quin sistema, quines prestacions, les realitzarem. Sabem que el programa que hem dissenyat ha estat amb el pur propòsit que pugui ser executat desde qualsevol ordinador que disposi d'una GPU amb tecnologia CUDA, i no en un servidor en concret, però és important clarificar el sistema que fem servir, doncs

els resultats que obtenim de l'anàlisi podrien variar notablement segons si utilitzem, per exemple, una GPU o una altre.

Per a, posterior anàlisi utilitzarem un ordinador portàtil, amb els següents components:

- I7-108700H. 8 nuclis, 16 threads, max Freq: aprox 4.7 Ghz.
- 32 GB RAM (16x2) DDR4 3200 MHz.
- **GPU:** RTX 3070 a 100 Watts.

El que principalment ens interessa són les característiques CUDA d'aquesta GPU.

Per fer això, podem executar la comanda deviceQuery:

```
CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "NVIDIA GeForce RTX 3070 Laptop GPU"
  CUDA Driver Version / Runtime Version      11.6 / 11.6
  CUDA Capability Major/Minor version number: 8.6
  Total amount of global memory:            8192 MBytes (8589410304 bytes)
  (040) Multiprocessors, (128) CUDA Cores/MP: 5120 CUDA Cores
  GPU Max Clock rate:                     1290 MHz (1.29 GHz)
  Memory Clock rate:                      6001 Mhz
  Memory Bus Width:                       256-bit
  L2 Cache Size:                          4194304 bytes
  Maximum Texture Dimension Size (x,y,z)   1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers 1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers 2D=(32768, 32768), 2048 layers
  Total amount of constant memory:          65536 bytes
  Total amount of shared memory per block:  49152 bytes
  Total shared memory per multiprocessor:  102400 bytes
  Total number of registers available per block: 65536
  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): (2147483647, 65535, 65535)
  Maximum memory pitch:                  2147483647 bytes
  Texture alignment:                     512 bytes
  Concurrent copy and kernel execution: Yes with 5 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
  CUDA Device Driver Mode (TCC or WDDM): WDDM (Windows Display Driver Model)
  Device supports Unified Addressing (UVA): Yes
  Device supports Managed Memory:        Yes
  Device supports Compute Preemption: Yes
  Supports Cooperative Kernel Launch: Yes
  Supports MultiDevice Co-op Kernel Launch: No
  Device PCI Domain ID / Bus ID / location ID: 0 / 1 / 0
  Compute Mode:
    < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.6, CUDA Runtime Version = 11.6, NumDevs = 1
Result = PASS
```

D'entre el que més ens interessa d'aquí, podem destacar que es tracta d'una GPU amb CUDA capability 8.6, la qual disposa de 5120 CUDA Cores distribuïts en 40 processadors.

Observem que el warp size és 32, és per això que utilitzem mides del blocksize múltiples de 32 (16x16 o 32x32). També, degut a que els maxims threads/block son 1024, el màxim tamany de bloc que podem utilitzar és 32x32.

La màxima memòria shared per bloc és de 49152 bytes, per tant, no hauríem de tenir cap problema ni en el kernel 2 ni el 3, ja que, per exemple, si utilitzem un blocksize de 32x32, al kernel 3 reservariem per a cada bloc:  $(32*3)*(32*3)*4 = 36864$  bytes < 49152 bytes.

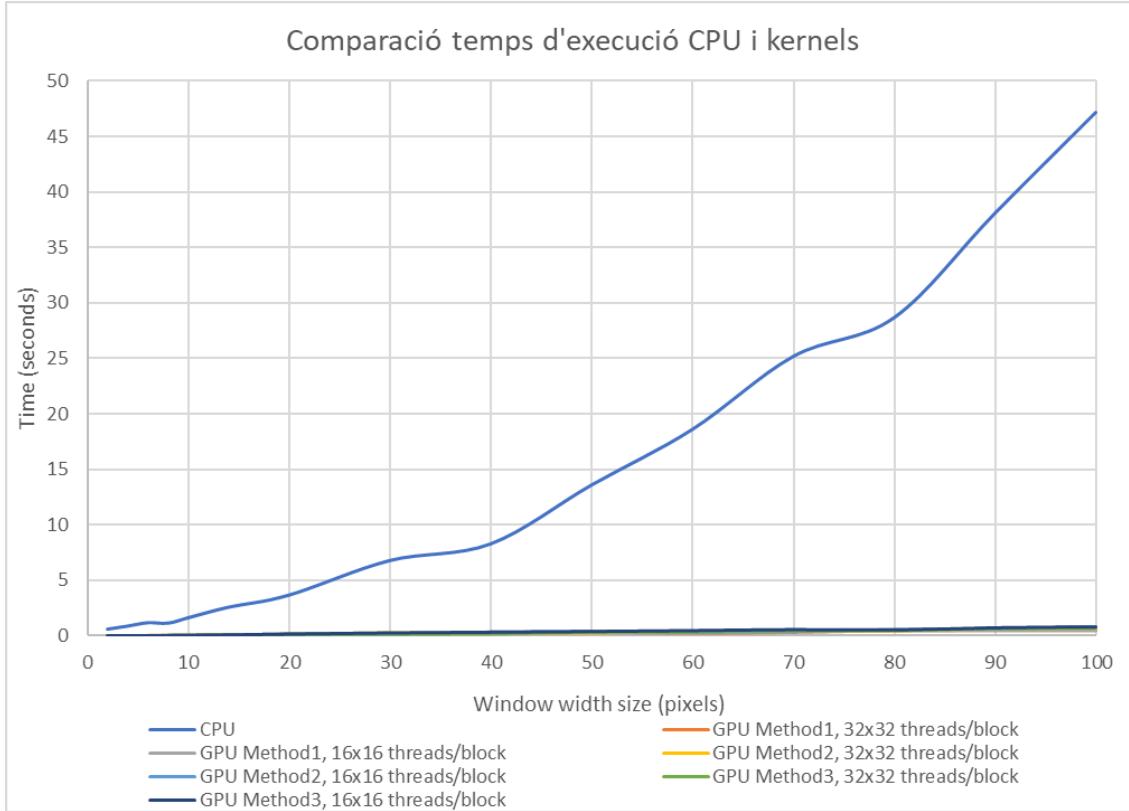
Finalment observem que el tamany màxim de la memòria constant és de 65536 bytes. Aquest és el fet pel qual se'ns a fet impossible dissenyar el quart kernel, ja que la nostra intenció era emmagatzemar la imatge original a memòria constant, però això hauria produit que només haguem pogut treballar amb imatges molt petites.

## Comparació CPU vs Kernels GPU

Comencem amb la comparació més important de totes, comparant els temps d'execució dels kernels de la GPU amb l'algorisme de CPU. Aquí, si hem programat correctament els kernels, hauríem d'observar almenys un speedup per a tots els 3 kernels respecte la CPU.

Per comprovar això, fem un recull de dades dels temps d'execució, de només la **fase de binarització**, tant per la CPU com pels tres kernels, ja que el que més ens interessa és la comparació de la binarització de la imatge, (i no per exemple el pas a grayscale, o les transferències de memòria que hi ha pel mig, les quals les fem precisament per guardar la imatge). El recull de dades el farem pels límits que hem establert a la interfície gràfica de l'amplada de finestra (de dos píxels

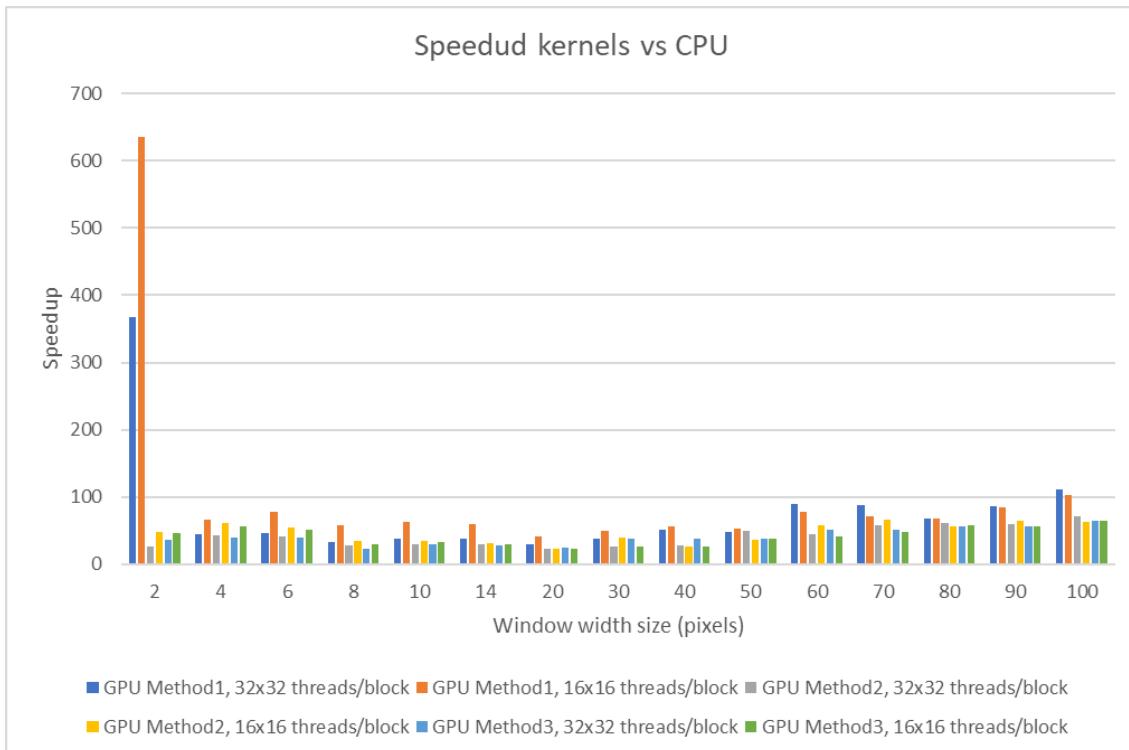
d'amplada a com a molt 100 pixels d'amplada). Els resultats els podem observar al gràfic següent:



Observant el gràfic, a primera vista ja podem observar que sigui el kernel que sigui, i el tamany de blocksize que sigui, el temps d'execució de l'algorisme de CPU és molt més gran, arribant a superar els 45 segons per a una finestra d'amplada 100. En canvi, pels kernels el temps d'execució en cap cas supera el segon.

De fet, la diferència entre CPU - kernels és tan gran, que no hi podem observar diferència al gràfic entre els temps d'execució dels kernels.

Aprofitant aquestes dades, anem a interpretar-les ara de forma que observem realment el speedup que obtenim executant els algorismes dels kernels respecte el de la CPU:



En el gràfic superior, primer de tot cal comentar que en la majoria de casos, per a tots els kernels hi observem un speedup d'almenys 25-50 més ràpid. Observem, a més, que sembla que a mesura que augmenta el tamany de finestra el speedup també sembla augmentar a poc a poc en el cas general. És tracta d'una cosa curiosa, ja que el fet que augmenti el tamany de finestra no fa que augmenti més la paralelització a la GPU, sinó que fa que els threads hagin de fer més feina.

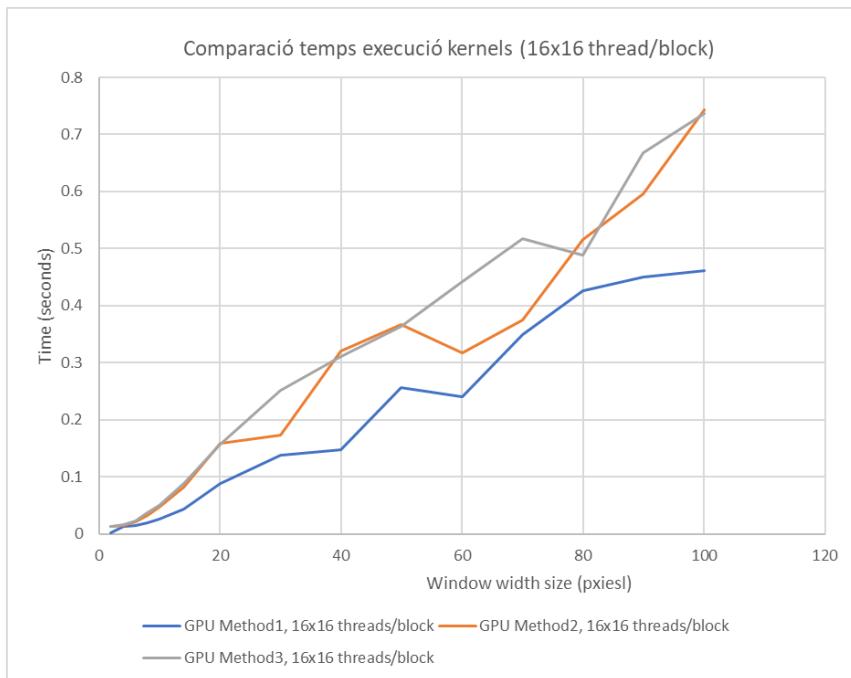
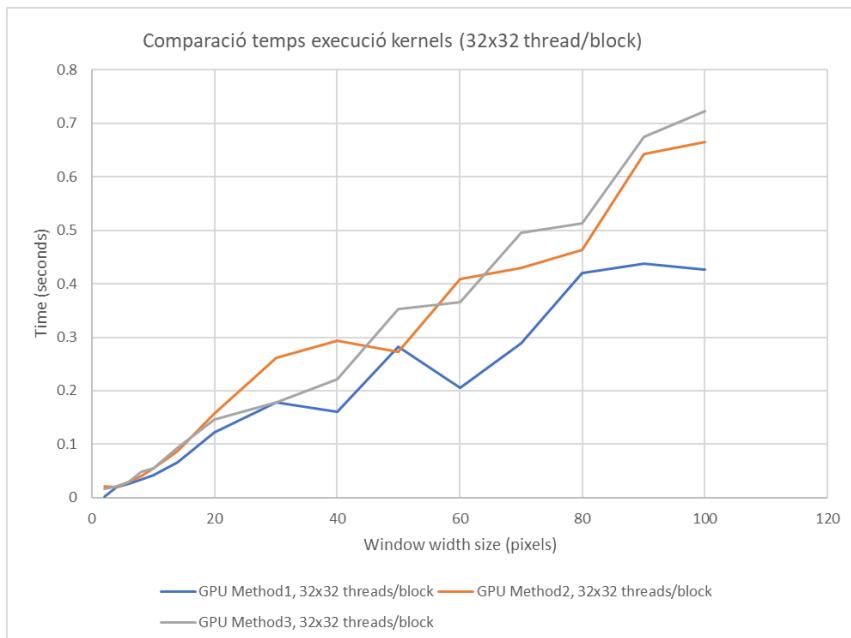
No obstant, la cosa més destacada del gràfic superior és l'enorme speedup que obtenim per a un tamany de finestra dos al primer kernel, tant per a 32x32 threads/block com per 16x16 threads/block. Un speedup molt més superior que a la resta dels casos, el qual segurament es deurà al poc nombre d'instruccions que han d'executar els threads i els pocs accessos a memòria global. És tracta d'una cosa que haurem d'observar posteriorment amb els profilers de Nvidia.

## Comparació Kernels GPU

Una vegada ja hem comprovat que els kernels són molt més ràpids que la CPU, ara toca comparar els kernels entre ells, per veure quin és més ràpid, i perquè el que és més lent ho és.

L'anterior gràfic ja ens ha donat pistes sobre quin seria el més ràpid, el mètode 1. Cosa que principalment ens sorprèn, ja que això vol dir que llavors els intents d'optimització amb la memòria shared no han servit de res.

Intentem observar aquesta diferència entre els temps d'execució amb una mica més de presisió, tant per blocs 32x32 threads com de 16x16 threads:

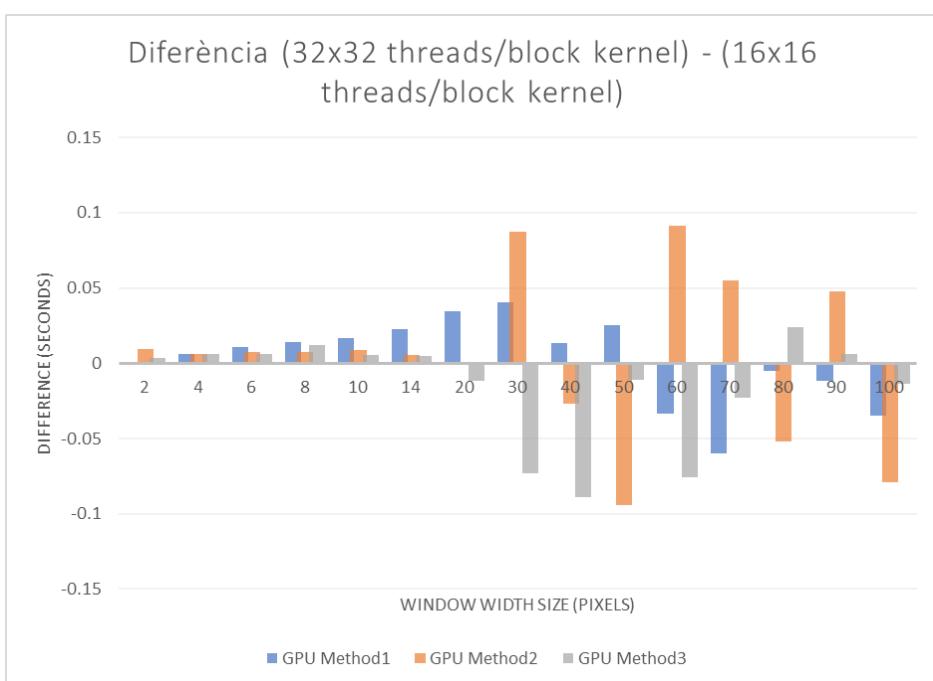


Efectivament, veient els gràfics superiors sembla ser per a quasi tots els casos, el mètode 1 sembla ser el més ràpid. Mentre que pel que fa al mètode 2 i 3, no sembla estar clar quin dels dos és millor.

Cal destacar, però, que a part que estem parlant de temps de d'execució molt baixos (i, on per tant, segurament hi haurà certs errors visibles de presició), a l'hora de recollir les dades ens ha marcat l'atenció **l'inestabilitat** dels temps d'execució dels mètodes 2 i 3, respecte al mètode 1. Mentre que el primer kernel quasi sempre proporciona uns temps d'execució similars, el kernel 2 i 3 a vegades per la mateixa amplada de finestra el temps d'execució podia arribar a variar unes quantes dècimes de segon, produint que haguéssim de fer la mitjana per calcular els gràfics. És tracta d'una inestabilitat, per tant, que ha d'estar marcada per l'ús de la memòria shared.

Un cop vist que el kernel 1 és lleugerament inferior en temps d'execució, anem a veure que passa si ara comparem els kernels segons si els executem amb blocs de 16x16 threads o 32x32 threads, calculant la diferència per a cada kernel de la seva execució amb blocs de 32x32 threads - la seva execució amb blocs de 16x16 threads.

Aquí, la nostra hipòtesi és que en la majoria de casos aquesta diferència sigui negativa, és a dir que les execucions de blocs de 32x32 threads sigui inferior, ja que pensem que com més threads tingui un bloc, llavors més threads tindran la majoria de warps i per tant de mitjana hi haurà més threads en paral·lel executant-se al mateix temps. Observem el gràfic:



Vegem que la nostra hipòtesi no sembla complir-se, i que sembla haver-hi una mena de punt d'inflexió entre una finestra de 30 pixels d'amplada i 40 pixels d'amplada, on sembla ser que es canvia el comportament en que primerament blocs de 16x16 threads trigaven menys que blocs de 32x32 threads i cada cop o feien menys, a trobar-nos amb casos on de sobte blocs de 32x32 threads triguen menys que blocs de 16x16 threads, sobretot en el cas més estable, el mètode 1, la tendència de la diferència passa a ser la contrària.

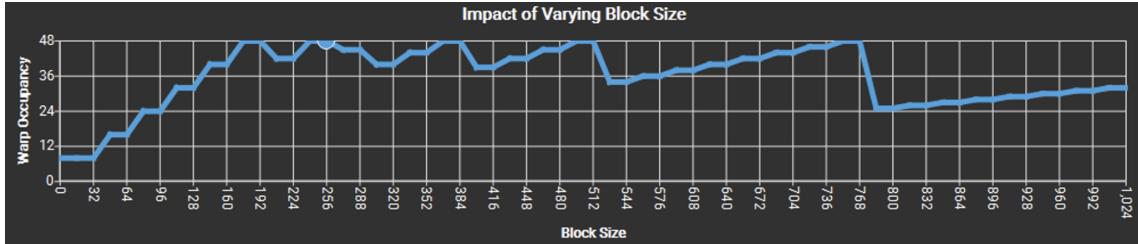
Llavors, en conclusió pel que acabem de veure, hi ha tres coses principals que ens han marcat l'atenció, i que hem d'analitzar amb més profunditat: d'una banda, què passa exactament amb aquest comportament que acabem de notar amb execucions de blocs de 32x32 threads respecte 16x16 threads; d'altre banda, perquè el mètode 1 acaba essent més eficient que els dos altres, sobretot que el mètode 3.

Per fer aquests anàlisis, és important dir que hem utilitzat les noves eines de profiling de Nvidia, Nvidia Nsight **Compute**, i Nvidia Nsight **Systems**. Es tracta de eines que donen un analisi molt complet del que passa en l'execució d'un fitxer CUDA (Systems) o d'un kernel (Compute), i no tant sols ens proporcionen tota la informació que comandes com nvprof ens donen, sinó també consells i recomanacions sobre perquè el que està passant passa, i com arreglar-ho.

## Comparació 16x16 threads/block, 32x32 threads/block

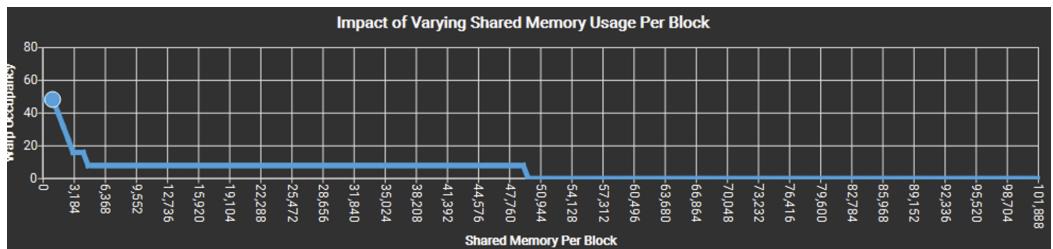
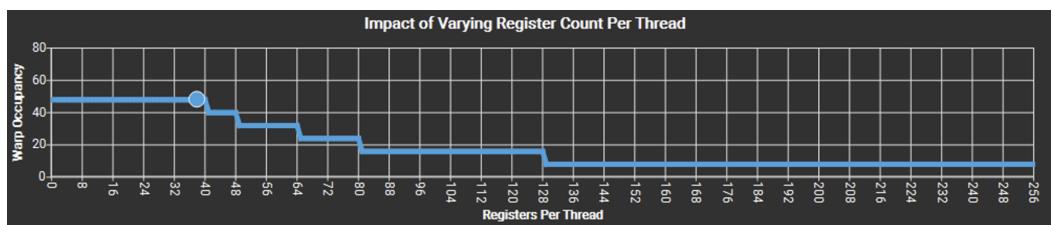
Per a aquest anàlisis, primer hem volgut esbrinar perquè per al mètode 1, des de que l'amplada de finestra és de 2 pixels fins 30 pixels el temps d'execució per a blocs de 16x16 threads és millor, i cada cop més inferior. La resposta a això ho hem trobat mirant les dades sobre warp occupancy que ens ha proporcionat Nvidia Nsight Compute. Recordem que Warp Occupancy consisteix en el ratio del nombre actiu de warps per multiprocessador respecte el màxim nombre possible de warps.

La theoretical Warp Occupancy per a un kernel pot ser impactada/limitada per molts factors, i el blocksize resulta esser el factor per al cas que estem analitzant, tant com podem observar al següent gràfic, per al mètode 1:

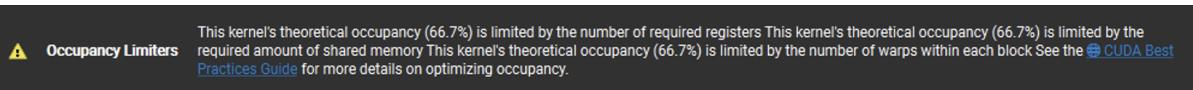


Al gràfic superior podem observar que, quan triem un blocksize de 16x16 threads (256 threads) podem arribar a una warp occupancy màxima superior a quan utilitzem un blocksize de 32x32 threads (1024).

De fet, estem segurs que per a finestres petites i per al mètode 1, aquesta és una de les principals raons per les quals el temps d'execució per a blocs de 16x16 threads és inferior, ja que si mirem altres possibles factors que poden afectar la warp Occupancy (com la shared memory, que en el mètode 1 no en tenim) aquests no l'affecten, com podem veure en els gràfics posteriors:



En canvi, per als mètodes 2 i 3, l'estabilitat que bé donada i que fa que no trobem un cert patró de quin blocksize és millor, es deu a què per a aquests dos kernels la warp occupancy també es veu afectada per altres factors, tal com ens informa el següent missatge:



Hauríem de veure, també, pel mètode 1, perquè per a finestres grans veiem una tendència a que el kernel executat amb blocs de 32x32 threads trigui menys que executat amb blocs de 16x16 threads. Per fer això, fem un anàlisi amb Nsight Compute tant executant el kernel en aquests dos casos per a una finestra d'amplada de 100 pixels.

Primerament, relacionant-ho amb lo anterior, observem igualment que per a 16x16 threads la warp occupancy segueix essent millor per la raó que hem explicat abans:

Warp occupancy execució amb blocs de 16x16 threads: (46.28)

|  |       |                                |    |
|--|-------|--------------------------------|----|
| Theoretical Occupancy [%]              | 100   | Block Limit Registers [block]  | 6  |
| Theoretical Active Warps per SM [warp] | 48    | Block Limit Shared Mem [block] | 8  |
| Achieved Occupancy [%]                 | 97.68 | Block Limit Warps [block]      | 6  |
| Achieved Active Warps Per SM [warp]    | 46.88 | Block Limit SM [block]         | 16 |

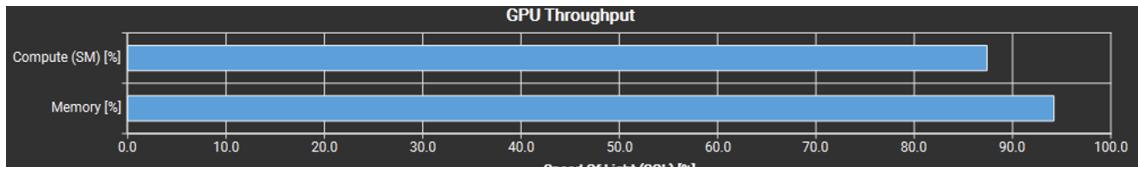
Warp occupancy execució amb blocs de 32x32 threads (30.70)

|  |       |                                |    |
|--|-------|--------------------------------|----|
| Theoretical Occupancy [%]              | 66.67 | Block Limit Registers [block]  | 1  |
| Theoretical Active Warps per SM [warp] | 32    | Block Limit Shared Mem [block] | 8  |
| Achieved Occupancy [%]                 | 63.95 | Block Limit Warps [block]      | 1  |
| Achieved Active Warps Per SM [warp]    | 30.70 | Block Limit SM [block]         | 16 |

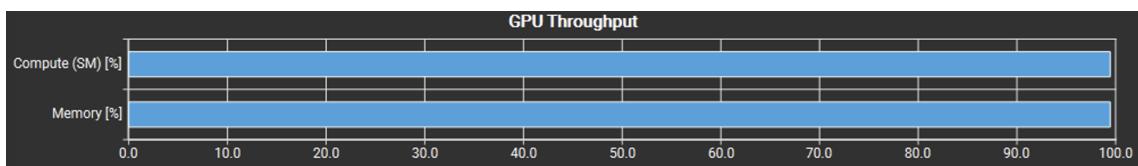
Ara ens falta trobar allò que penalitza el temps d'execució per sobre de la occupancy dels warps, és a dir, allò que fa que tot i que tinguem una warp occupancy major, igualment tardem més.

Si comencem a comparar els anàlisis, ens adonem que l'execució amb blocs de  $32 \times 32$  threads és capaç d'utilitzar més recursos de GPU que l'execució de blocs de  $16 \times 16$  threads:

Throughput GPU execució amb blocs  $16 \times 16$  threads:

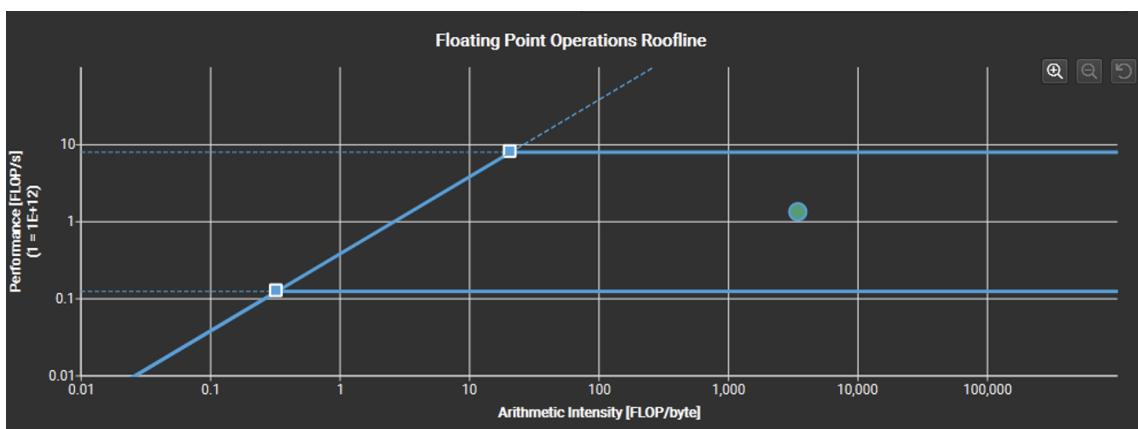


Throughput GPU execució amb blocs  $32 \times 32$  threads:

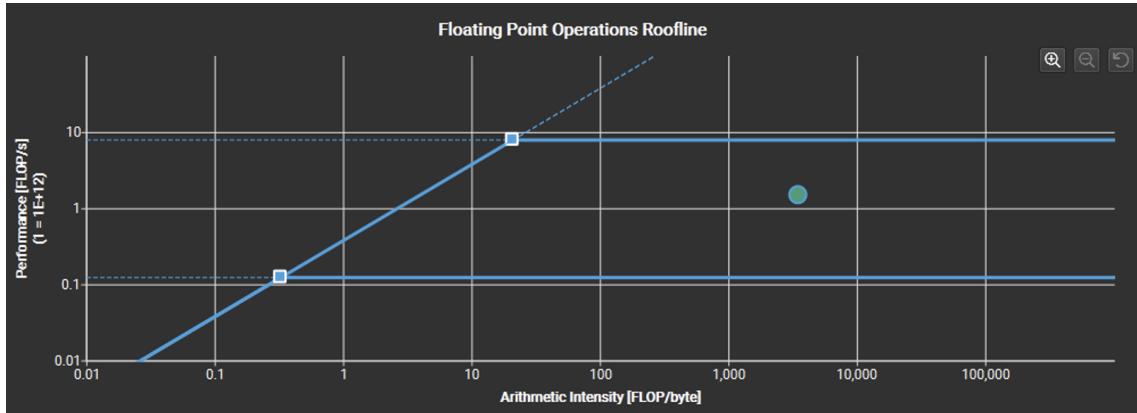


Podem observar que realment amb blocs de  $32 \times 32$  threads som capaços d'utilitzar pràcticament tots els recursos del sistema, fent que sigui tant el processador de la GPU com la memòria de la GPU el bottleneck de l'execució, i per tant obtenint lleugerament més FLOPS/s que amb  $16 \times 16$  threads (tot i que en els dos casos se supera el teraflop/s, valor que ja és molt alt (tot i que la GPU que estem utilitzant pugui arribar als 16 teraflops/s)):

Flops execució amb blocs  $16 \times 16$  threads:

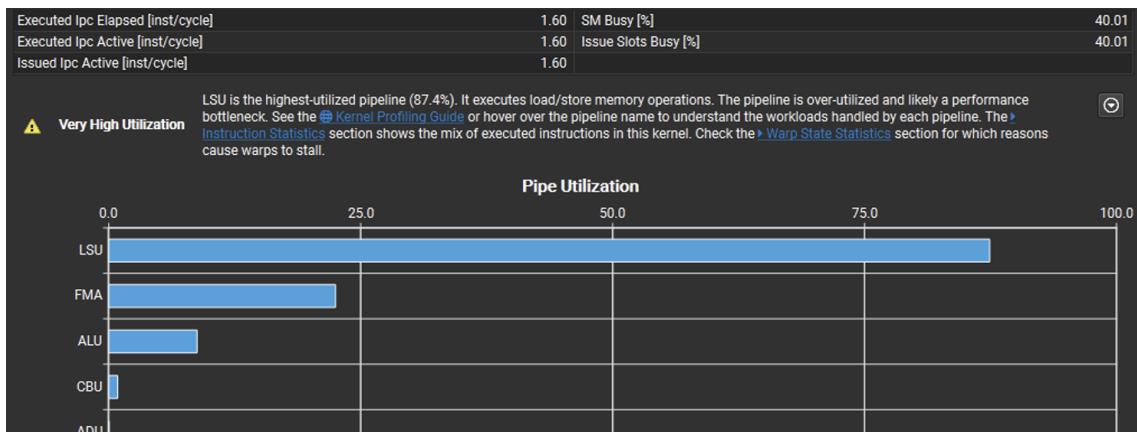


Flops execució amb blocs 32x32 threads:

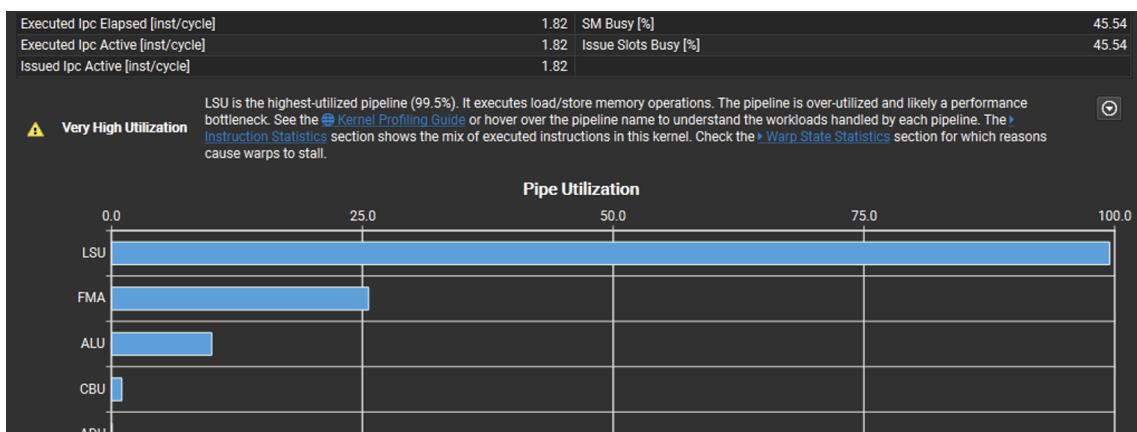


Degut a aquest major aprofitament dels recursos, també podem veure que obtenim més instruccions/cicle en el cas de blocksize 32x32 threads i som capaços de bottleneckjar el pipeline d'instruccions LSU (de carregar i guardar memòria), a diferència de amb blocksize 16x16 threads en que no som capaços de bottleneckjar cap pipeline d'instruccions:

Execució blocksize 16x16 threads:

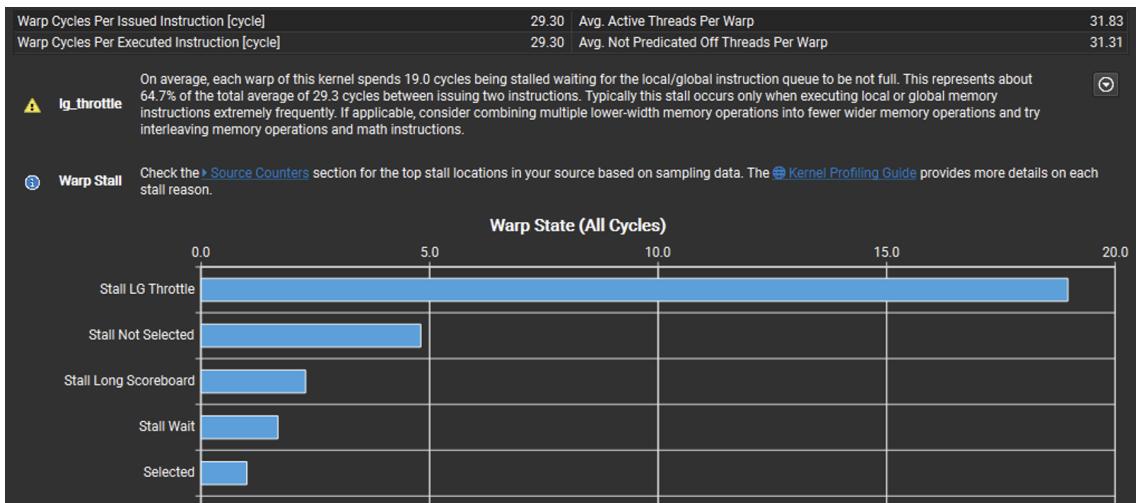


Execució blocksize 32x32 threads:

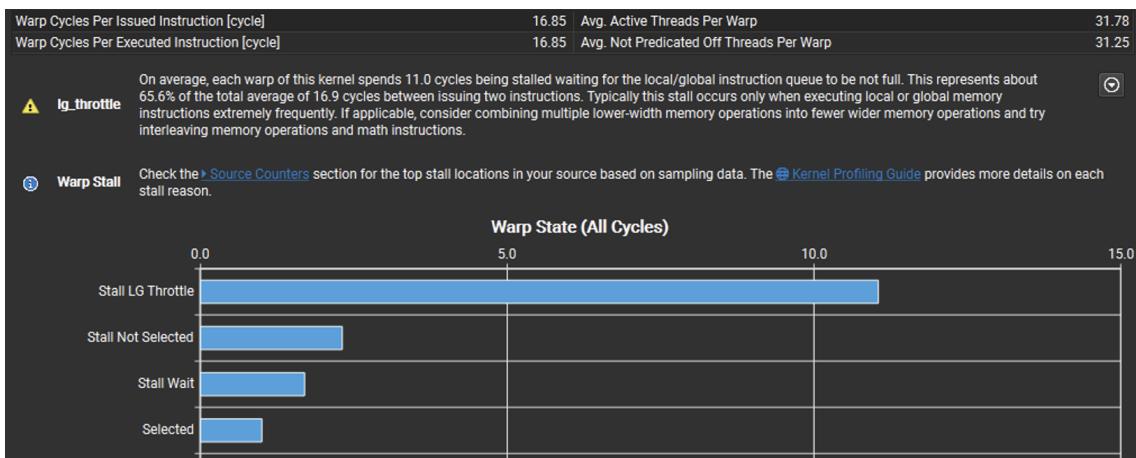


La qüestió ara és: i per què no som capaços d'obtenir un equivalent ús dels recursos amb blocksize de 16x16 threads? Doncs bé, la resposta és perquè amb blocksize 16x16 threads cada warp del kernel es passa molt més temps stalled (congelat) esperant-se a que la cua d'instruccions no estigui plena. Això ho podem observar en els gràfics següents, en que per a blocksize 16x16 els warps es passen molt més temps en un stall de tipus LG Throttle, és a dir un stall degut a que la cua d'instruccions de memòria està plena.

Execució blocksize 16x16 threads:



Execució blocksize 32x32 threads:

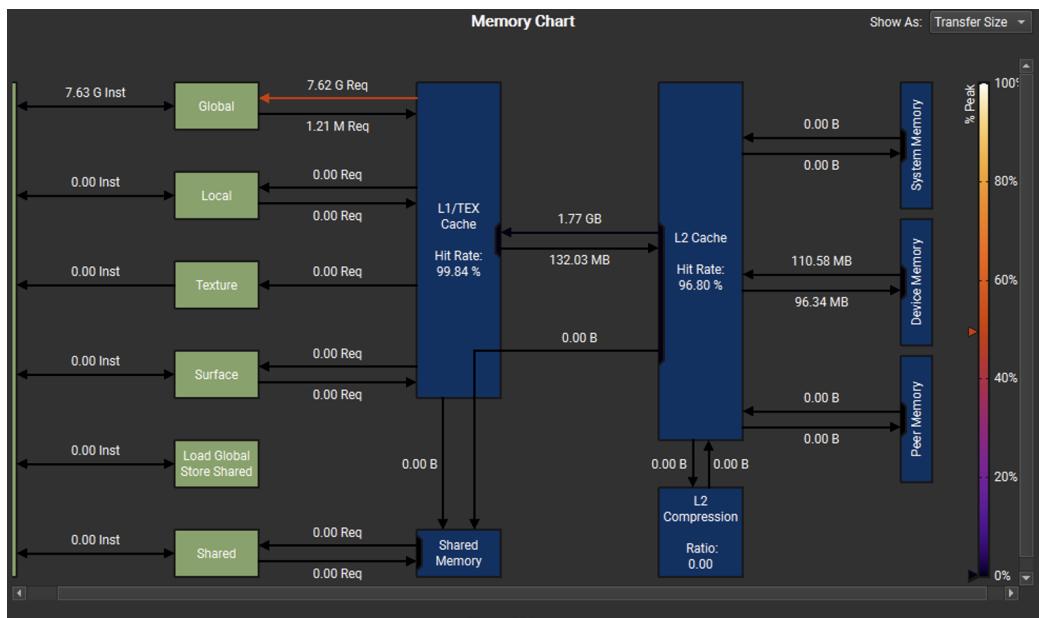


## Comparació kernel mètode 1, kernel mètode 3

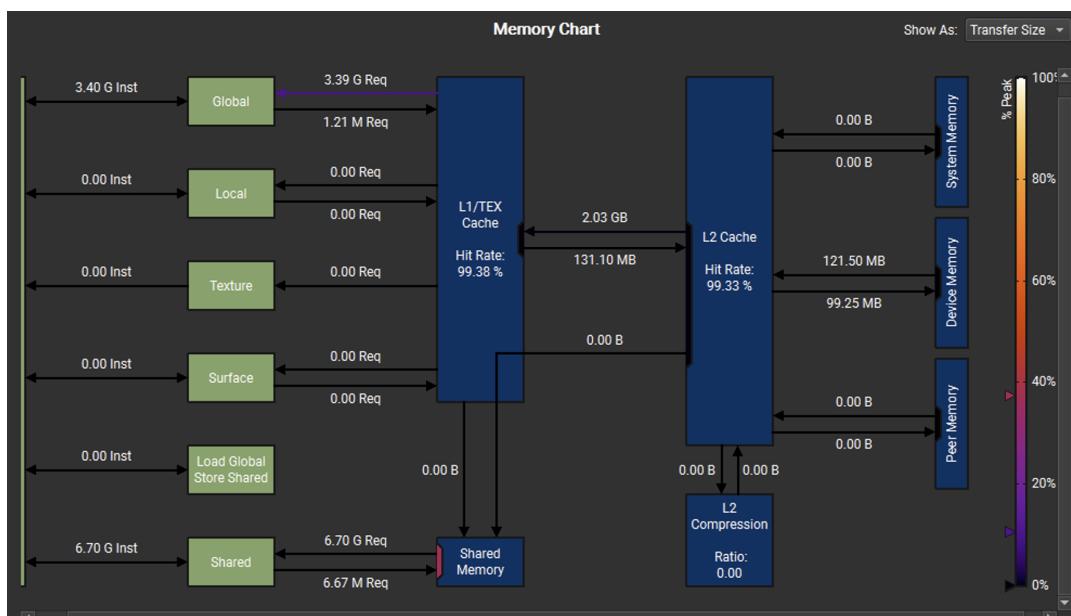
Anem ara a veure perquè el kernel 3 acaba essent més lent que el primer, és a dir el contrari del que ens pensavem. Per fer això, executarem un anàlisi de l'execució per a una finestra de 100 pixels d'amplada amb el mètode 1, i un amb el mètode 3 amb nsight compute.

Primer, però, anem a comprovar que la optimització que hem volgut aplicar al kernel 3 fa realment algun efecte, i que almenys ara no ens veiem penalitzats per la cua plena del pipeline d'instruccions de memòria:

Execució mètode 1:



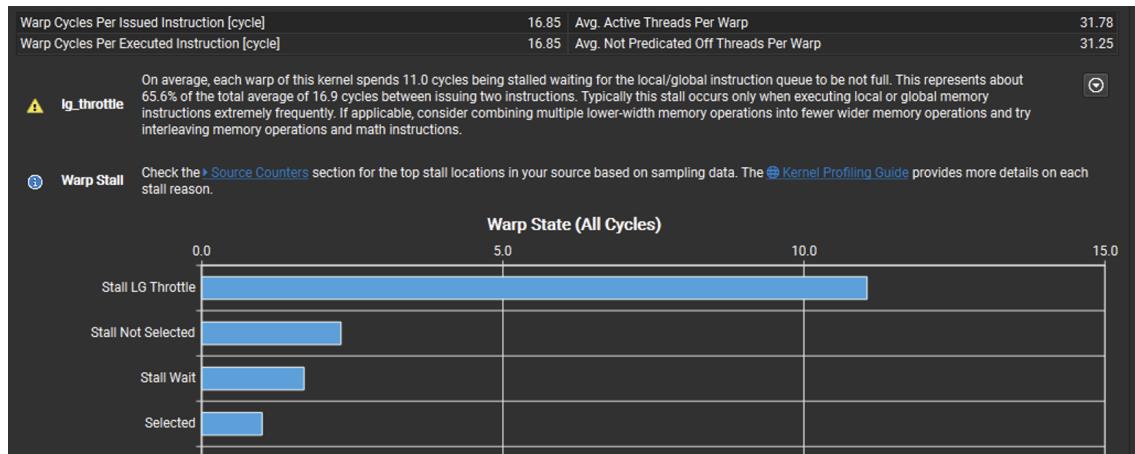
Execució mètode 3:



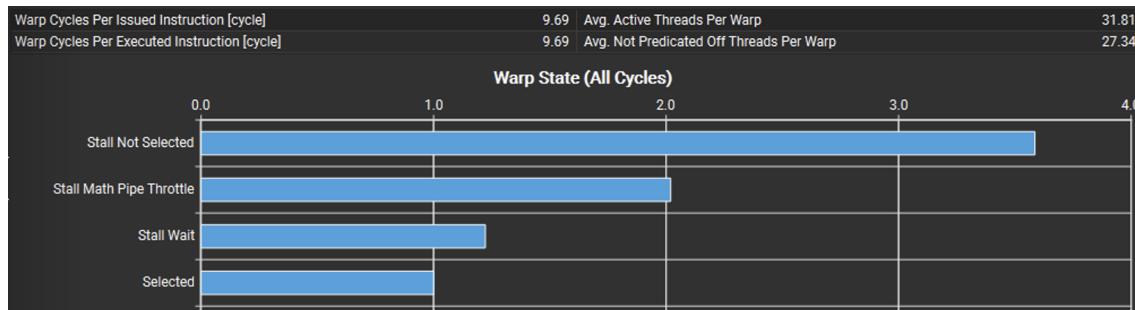
Efectivament, observant els gràfics anteriors podem veure com ara el bus que va del processador a la cache L1 està molt menys saturat amb el mètode 3, i tot i que el bus amb la memòria shared ho estigui una mica, no ho està tant.

Gràcies a això, podem veure també que per al mètode 3 els warps ja no es passen tant temps congelats per un stall lg throttle, i, en general és passen molts menys cicles congelats:

Execució mètode 1:

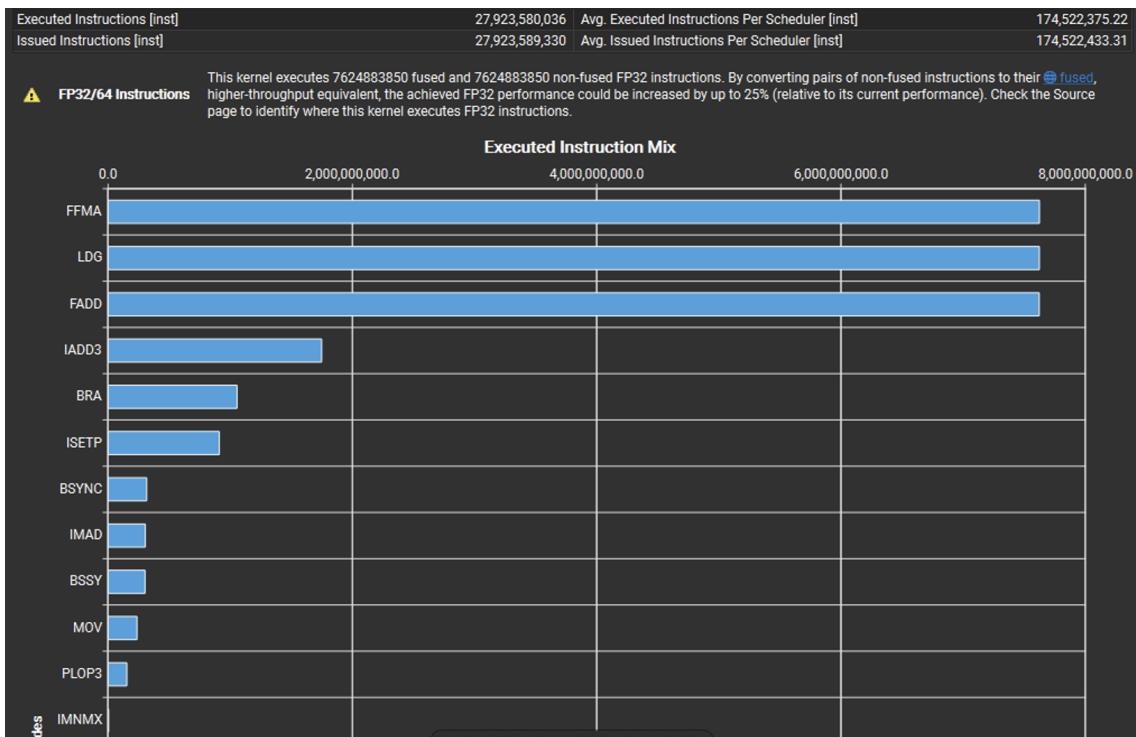


Execució mètode 3:

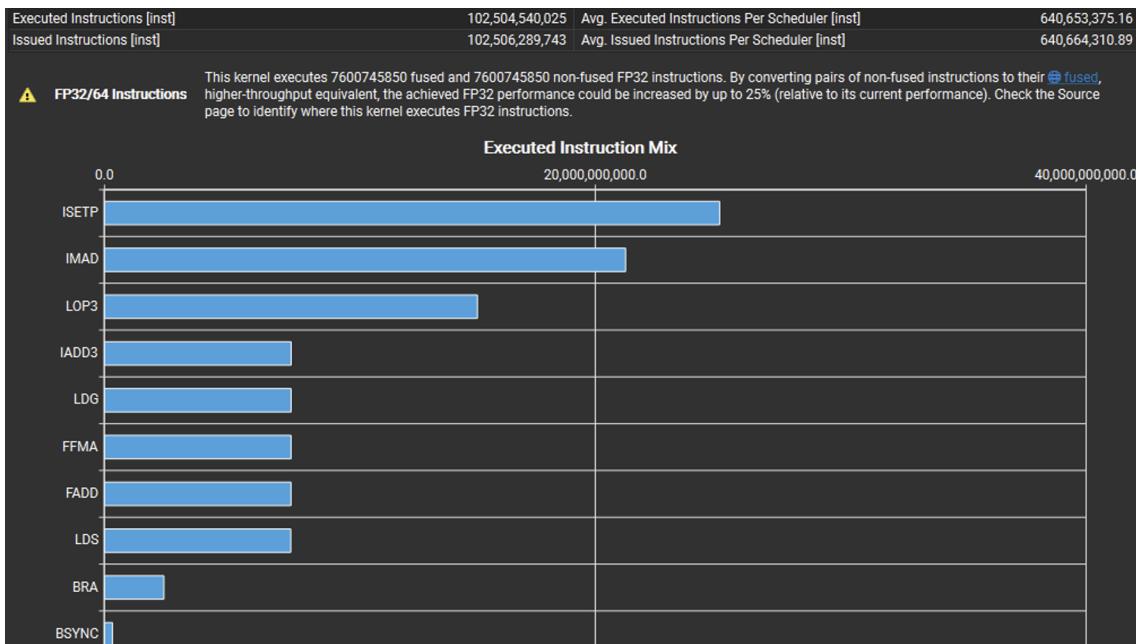


Llavors, si realment aquesta optimització a tingut efecte, perquè en canvi triguem més temps en executar el kernel? La resposta la tenim amb el nombre d'instruccions executades:

## Execució mètode 1:



## Execució mètode 2:



Observem que amb blocksize 32x32 tenim un total **molt més gran** d'instruccions totals executades. De fet, executem quasi 4 vegades més instruccions. Això, és degut, tal com hem explicat a la implementació del kernel 3, a que per aconseguir portar el valor de tots els corresponents pixels a la memòria shared, cada thread haurà d'executar un seguit d'instruccions condicionals considerable, fent que el codi quedi molt llarg, i que per tant cada thread hagi d'executar moltes més execucions.

Com podem observar a les imatges anteriors la instrucció més utilitzada (utilizada més de 20 bilions de vegades) per al kernel 3 és ISETP, la qual es tracta d'una comparació entre integers, mentre que aquesta instrucció en el kernel 1 només s'executa aproximadament 1 bilió de vegades, i la instrucció més utilitzada en el kernel 1, la qual és FFMA (una “fused” instruction, de forma que incrementa el rendiment en coma flotant) no s'executa ni 8 bilions de vegades.

Tot aquest munt d'instruccions més executades respecte el kernel 1, també fa que hi hagi més branques divergents entre threads (com podem veure a la imatge de sota), de forma que hi hagi menys capacitat de paralelització entre els threads:

Execució mètode 1:

|  |               |                         |
|--|---------------|-------------------------|
| Source metrics, including branch efficiency and sampled warp stall reasons. Warp Stall Sampling metrics are periodically sampled over the kernel runtime. They indicate when warps were stalled and couldn't be scheduled. See the documentation for a description of all stall reasons. Only focus on stalls if the schedulers fail to issue every cycle. |               |                         |
| Branch Instructions [inst]   | 1,373,514,162 | Branch Efficiency [%]   |
| Branch Instructions Ratio  | 0.05          | Avg. Divergent Branches |

Execució mètode 3:

|                            |               |                         |            |
|----------------------------|---------------|-------------------------|------------|
| Branch Instructions [inst] | 2,746,166,971 | Branch Efficiency [%]   | 96.23      |
| Branch Instructions Ratio  | 0.03          | Avg. Divergent Branches | 569,527.38 |

## Anàlisi altres crides CUDA

Fins ara ens hem centrat només en els kernels de la GPU, ja que es tracta de principal focus d'atenció, i punt de millora. No obstant, també hem de tenir present que per a fer computacions a la GPU, s'han de fer transferències de dades

CPU-GPU-CPU, les quals també triguen el seu temps. Tal com hem explicat abans, aquest temps no hauria de ser molt gran gràcies a la memòria pinned, de fet no ho és, ho podem comprovar directament amb la sortida que ens dona la interfície gràfica a la part inferior de la pantalla, però comprovem això fent una traçada del programa amb la eina Nsight Systems amb el kernel 1:

Visual trace:



Trace:

| Start Time | Duration   | Name                  |
|------------|------------|-----------------------|
| 9.85573s   | 303 ns     | cuDeviceGetLuid       |
| 9.85578s   | 329.301 ms | cudaHostAlloc         |
| 10.1851s   | 127.340 ms | cudaHostAlloc         |
| 10.3198s   | 17.213 µs  | cudaEventCreate       |
| 10.3198s   | 963 ns     | cudaEventCreate       |
| 10.3198s   | 18.173 µs  | cudaEventRecord       |
| 10.3198s   | 1.338 ms   | cudaMalloc            |
| 10.3211s   | 705.118 µs | cudaMalloc            |
| 10.3218s   | 13.787 ms  | cudaMemcpy            |
| 10.3356s   | 6.586 µs   | cudaEventRecord       |
| 10.3356s   | 2.252 µs   | cudaEventCreate       |
| 10.3356s   | 724 ns     | cudaEventCreate       |
| 10.3356s   | 1.347 µs   | cudaEventRecord       |
| 10.3356s   | 53.145 µs  | cudaLaunchKernel      |
| 10.3357s   | 4.126 µs   | cudaEventRecord       |
| 10.3357s   | 917 ns     | cudaEventCreate       |
| 10.3357s   | 623 ns     | cudaEventCreate       |
| 10.3357s   | 1.004 µs   | cudaEventRecord       |
| 10.3357s   | 316.766 ms | cudaMemcpy            |
| 10.6525s   | 8.345 µs   | cudaEventRecord       |
| 10.6525s   | 1.215 ms   | cudaFree              |
| 10.6537s   | 678.580 µs | cudaFree              |
| 10.6544s   | 15.462 µs  | cudaEventSynchronize  |
| 10.6544s   | 3.258 µs   | cudaEventSynchronize  |
| 10.6544s   | 3.406 µs   | cudaEventSynchronize  |
| 10.6544s   | 1.172 µs   | cudaEventDestroy      |
| 10.6544s   | 468 ns     | cudaEventDestroy      |
| 10.6544s   | 405 ns     | cudaEventDestroy      |
| 10.6544s   | 389 ns     | cudaEventDestroy      |
| 10.6544s   | 382 ns     | cudaEventDestroy      |
| 10.6544s   | 397 ns     | cudaEventDestroy      |
| 10.6544s   | 2.603 µs   | cudaDeviceSynchronize |
| 11.4196s   | 8.275 ms   | cudaFreeHost          |
| 11.4279s   | 9.816 ms   | cudaFreeHost          |

Observant el trace no veiem res que es faci de manera incorrecte. Amb el visual trace podem veure com les crides que triguen més són cudaHostAlloc, és a dir

l’alociació de memòria pinned a la RAM, mentre que memcpy amb memòria pinned no triga tant. La raó per la qual al visual trace veiem que hi ha una triga cudaMemcpy que triga molt, és degut a que s’està esperant a que s’acabi d’executar el kernel.