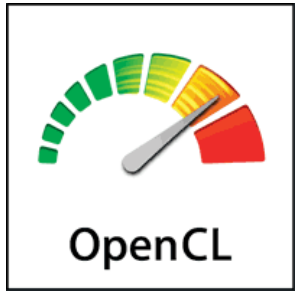


OpenCL

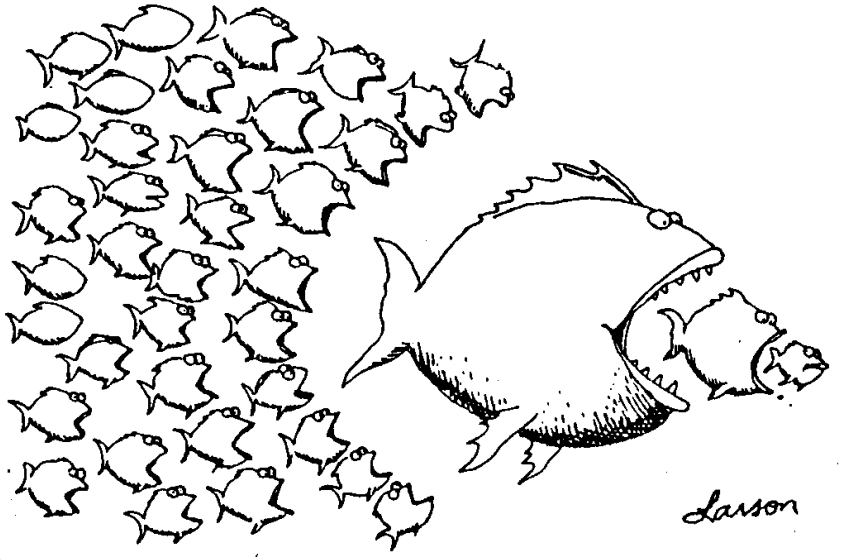
Optimization techniques

Fast or even more fast…

André D. Mora & José M. Fonseca
TAPDI - 2024/2025 DEEC/FCT/UNL

Optimization Techniques

1. **Parallelize as much as possible**
2. Optimize memory usage for maximum bandwidth
3. Maximize occupancy to hide latency
4. Optimize instruction usage for maximum throughput



Parallelize

- Distribute processing by as much threads as possible
- Remember that each work-group is assigned to a computing unit
- If threads from the same *work-group* need to communicate use, ***barrier (cl_mem_fence_flags)***
- If they are from different *work-groups* you should use global memory and have multiple kernels
- Take advantage of asynchronous kernel launches and memory transfers (***clEnqueue…()***)

Warps

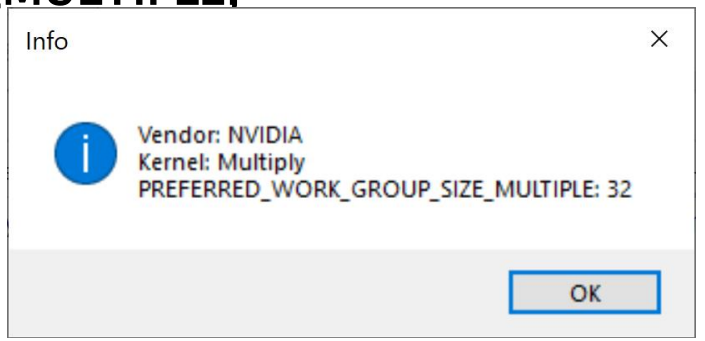
- CUDA *warp*, is the maximum number of threads that can execute the same instruction simultaneously within a single computing unit.
  - For NVIDIA, this warp size is 32
  - AMD cards, have a similar concept, the *wavefront,* whose size is 64.
  - Intel size is 32

✓ Define the work group size as a **multiple** of the Warp size

Ex. Workgroup size = (32, ?)

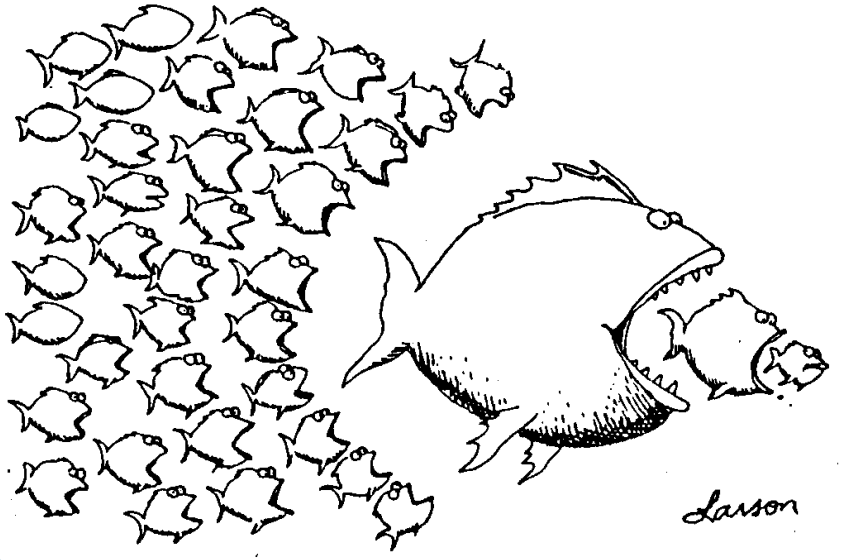
- In OpenCL you can get this size by querying the kernel properties:

<kernel name>.get_work_group_info(
cl.kernel_work_group_info.PREFERRED_WORK_GROUP_SIZE_MULTIPLE,
device)



Optimization Techniques

1. Parallelize as much as possible
2. **Optimize memory usage for maximum bandwidth**
3. Maximize occupancy to hide latency
4. Optimize instruction usage for maximum throughput

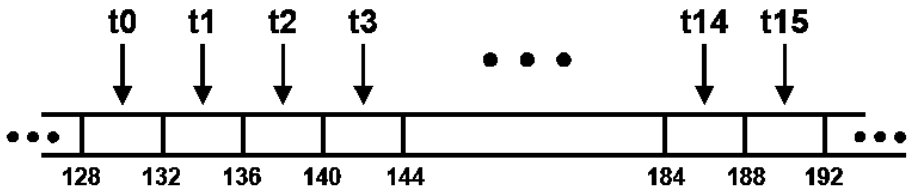


Optimize Memory Usage

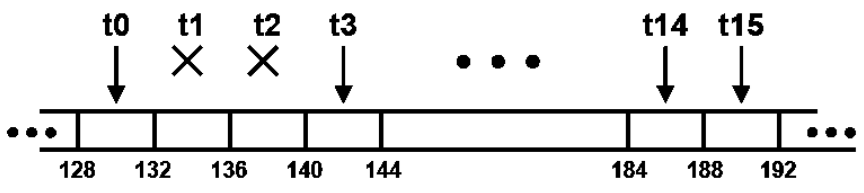
- Use high memory **Bandwidth**
  - Local memory is divided into equally-sized memory banks, which can be accessed simultaneously!
- Reduce Memory **Latency**
  - highest latency instructions that access Global Memory can take 400-600 clock cycles!!
  - Optimize Computing Units occupancy
- Optimize memory access patterns to get:
  - *Coalesced* global memory accesses
  - Local memory accesses with *no or few bank conflicts*
  - *Cache-efficient texture memory accesses*
  - *Same-address constant memory accesses*

Coalesced Memory Access

- *Coalesced* memory access is *aligned* memory access
- Create *coalesced* simultaneous accesses to Global Memory by each thread of a **half-warp** if:
  - The size of the memory element accessed by each thread is either 4, 8, or 16 bytes
  - The elements form a contiguous block of memory
  - The N^{th} element is accessed by the N^{th} thread in the half-warp
  - The address of the first element is aligned to 16 times the element's size

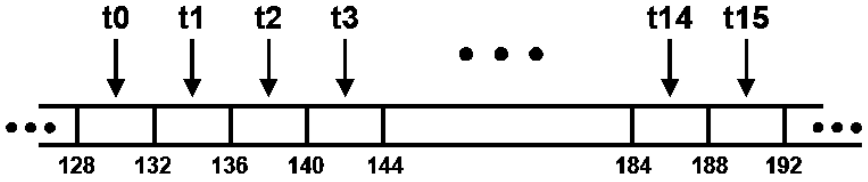


Coalesced `float` memory access



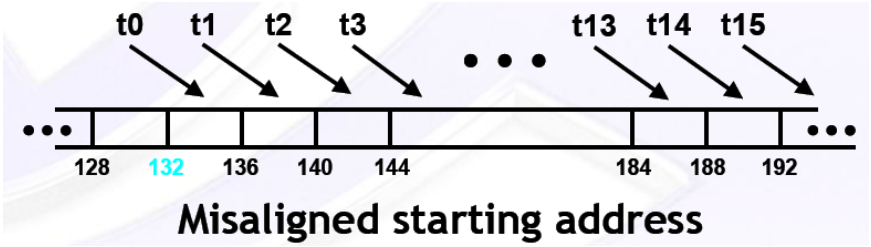
Coalesced `float` memory access (divergent warp)

Coalesced Memory Access– II



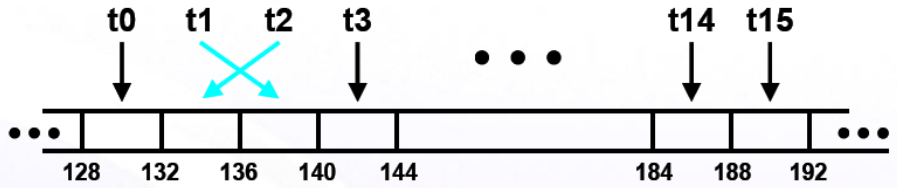
Coalesced `float` memory access

Coalesced access



Misaligned starting address

Non-coalesced access



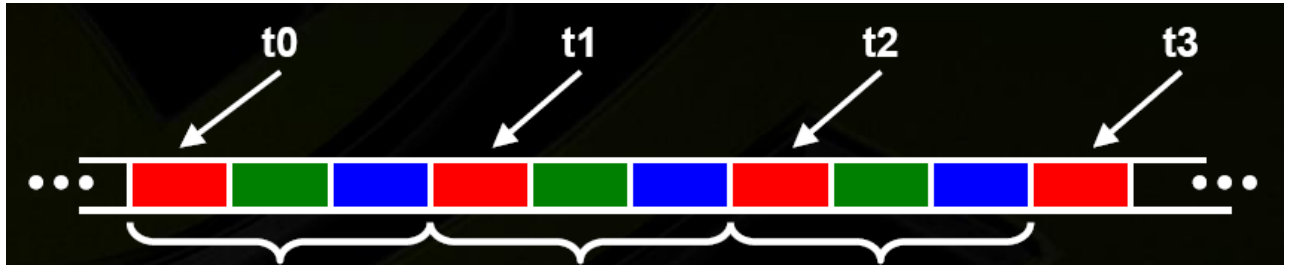
Non-sequential `float` memory access

Timing example: 12K blocks x 256 threads:

- 356µs –coalesced
- 357µs –coalesced, divergent warp
- **3494µs**– misaligned / permuted thread access

Local Memory – I

- Use Local Memory access
  - It is hundreds of times faster than global memory
  - Bandwidth of each bank is 32 bits per 2 clock cycles
- Design thread kernels that cooperate via local memory
  - "Block*ify* your data"…
- Avoid non-coalesced memory accesses
  - Is image memory access friendly for coalesced memory access?

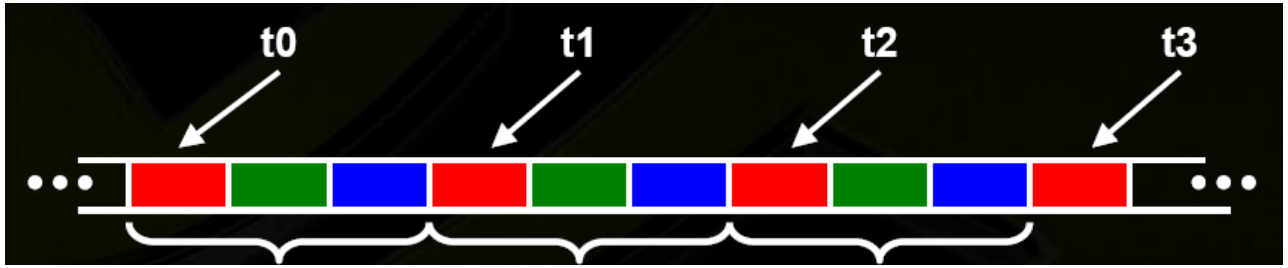


Probably not…

Local Memory –II – Example

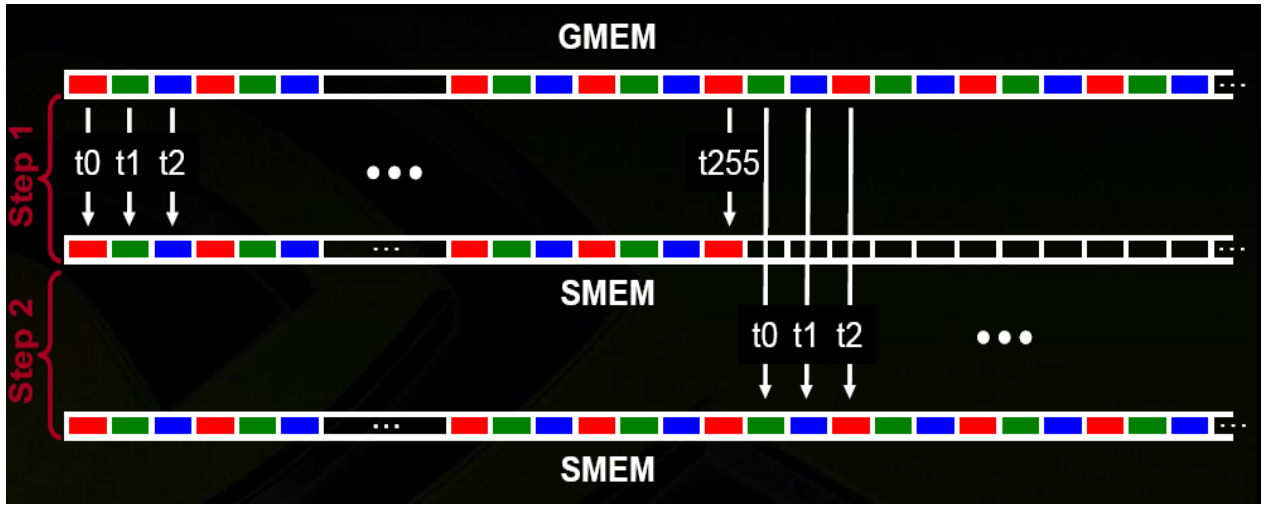
- Non-coalesced access to an image
  - Access RGB values (**float3** – 12bytes) all at the same time from global memory
  - ✗ **3x4bytes is not aligned**

```
int index = get_global_id(0);
float3 a = d_in[index];
```



Local Memory –III – Example

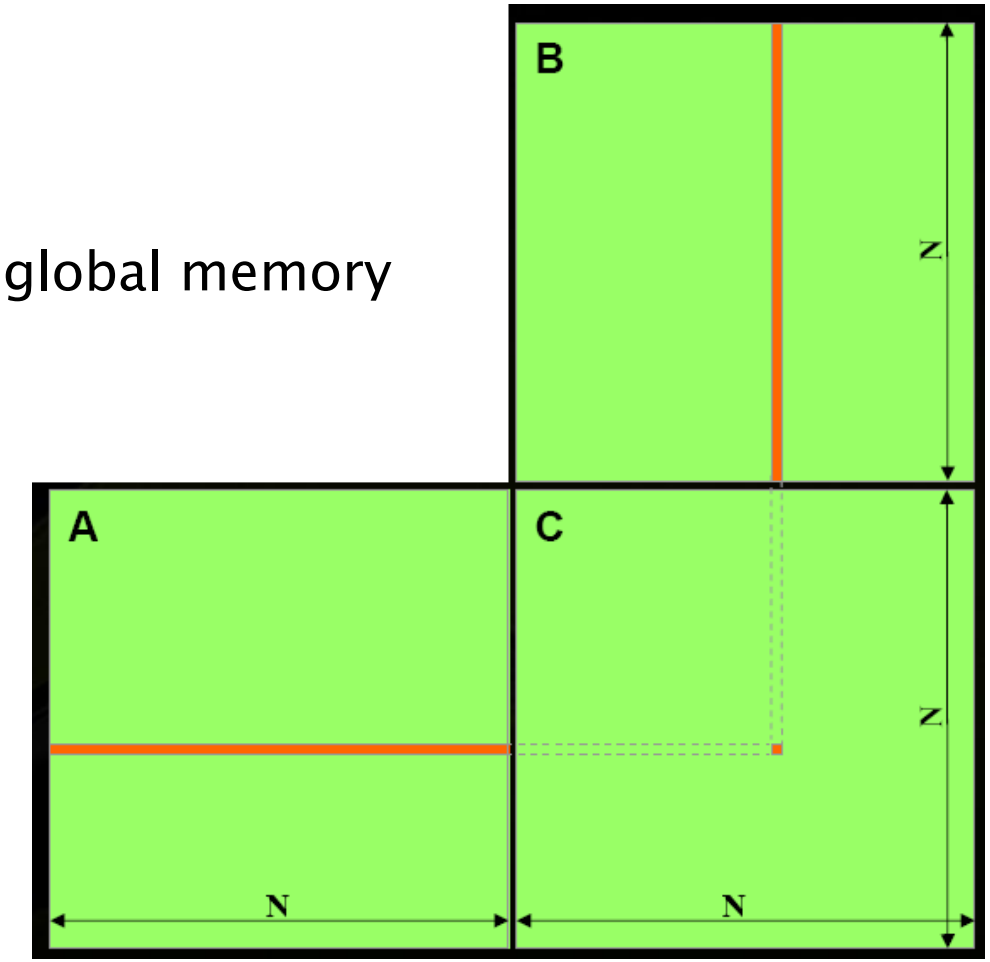
- Coalesced access to an image
  - Each thread reads from Global memory to local memory one RGB component at a time, then read RGB from local memory




```
__local  float s_data[256*3];
int indexG = get_global_id(0);
int indexL = get_local_id(0);

s_data[indexL] = g_in[indexG];
s_data[indexL +256] = g_in[indexG+256];
s_data[indexL +512] = g_in[indexG+512];

barrier(CLK_GLOBAL_MEM_FENCE);

float3 a=((float*)s_data)[threadIdx.x];
```

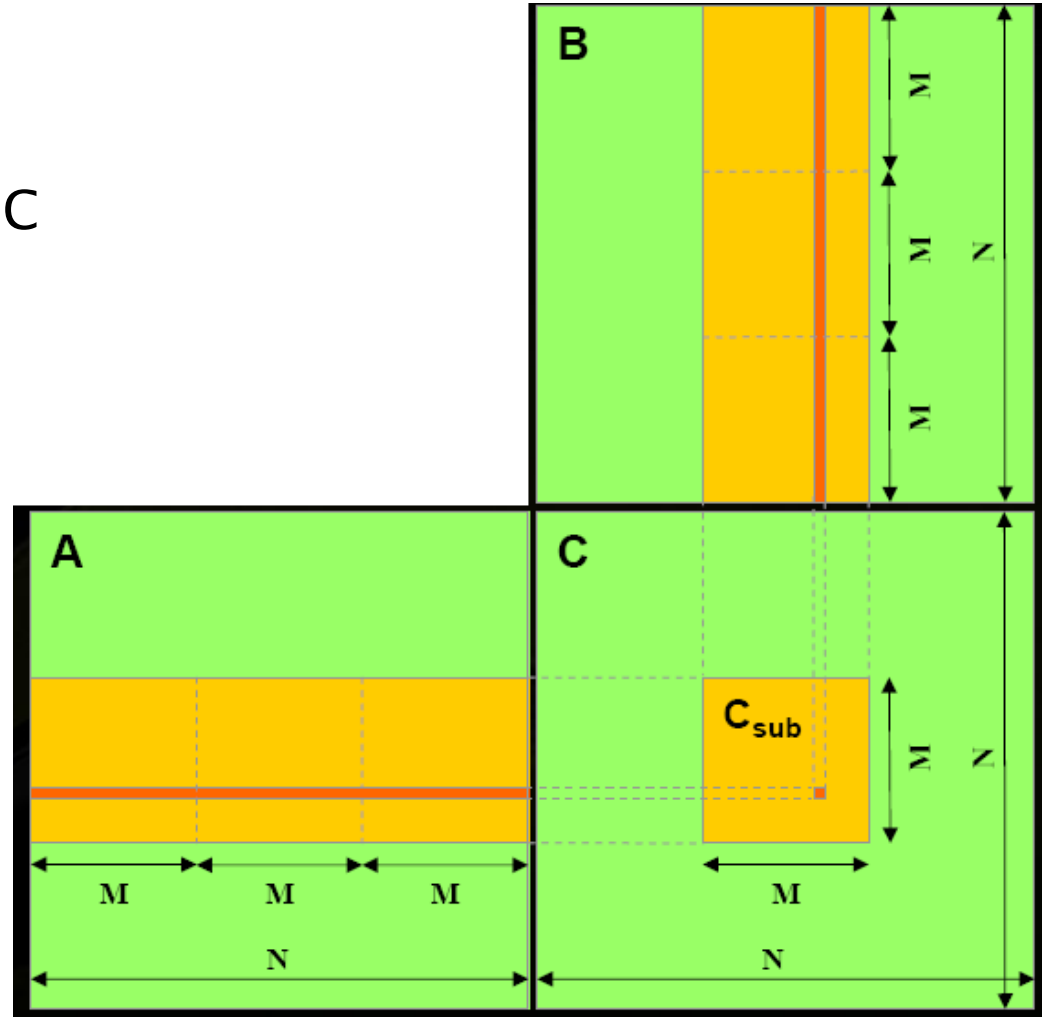
Local Memory – IV – Example

- Square Matrix multiplication
  - C=A*B (size NxN)
  - One thread for each element in C
  - A and B are loaded N^3 times from global memory
  - Waste bandwidth
  - Low efficiency



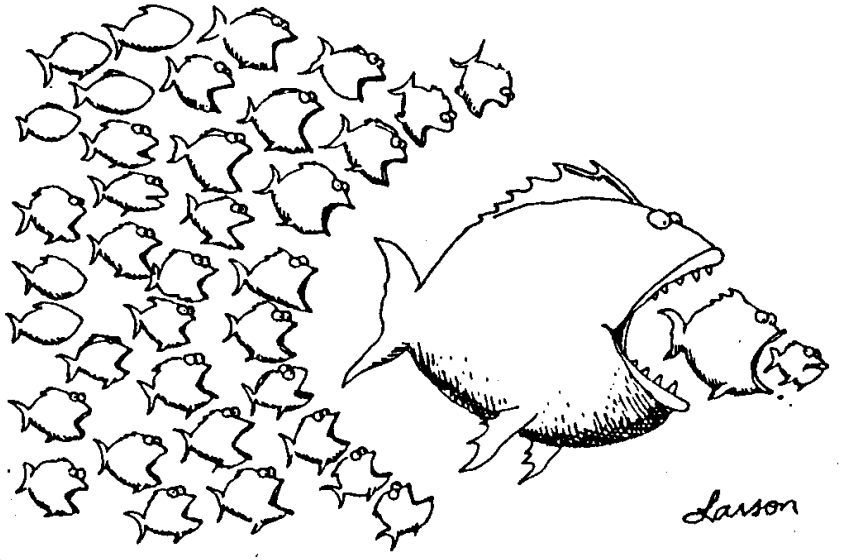
Local Memory – V – Example

- Square Matrix multiplication
  - C=A*B (size NxN)
  - One work-group for each MxM block in C
  - A and B are loaded (N^3/M^2)times from global memory to local memory
  - **Take advantage of Local memory High bandwith**



Optimization Techniques

1. Parallelize as much as possible
2. Optimize memory usage for maximum bandwidth
3. **Maximize occupancy to hide latency**
4. Optimize instruction usage for maximum throughput



Maximize Occupancy

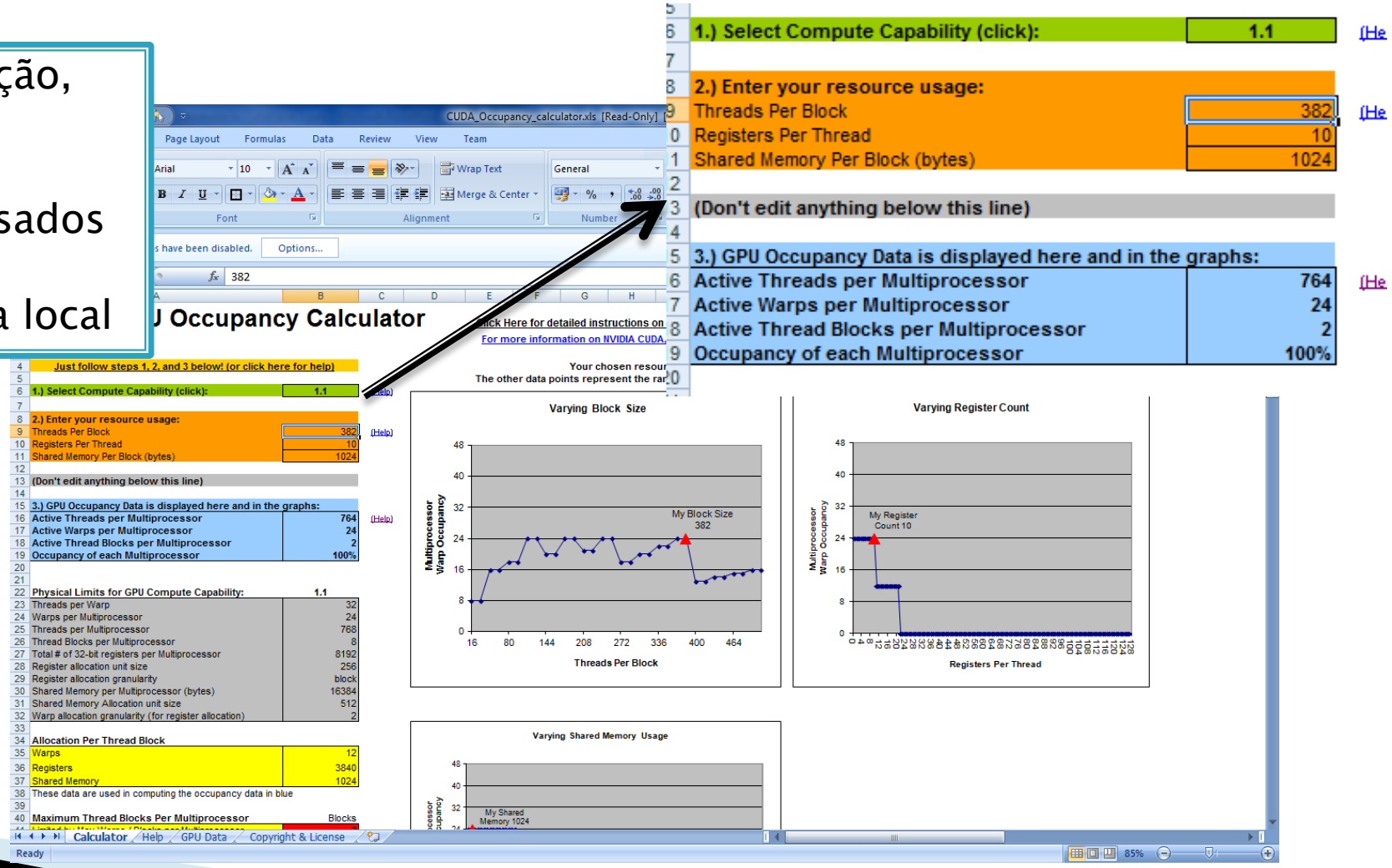
- Memory access latency will block dependent instructions in the same thread, but instructions in other threads are not blocked!
- Hide memory latency by running as many threads per computing unit as possible!
- Choose execution configuration to maximize
  - occupancy = (# of active warps) / (maximum # of active warps)

Tools – Occupancy calculator

https://developer.download.nvidia.com/compute/cuda/4_0/sdk/docs/CUDA_Occupancy_Calculator.xls

Permite estudar a ocupação, variando:

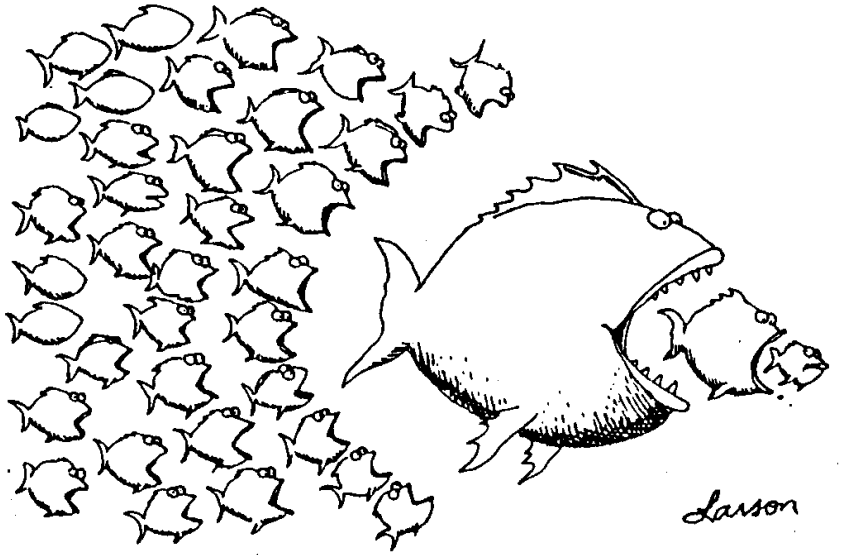
- Threads por bloco
- Número de registos usados por um kernel
- Dimensão da memória local



Occupancy.xls

Optimization Techniques

1. Parallelize as much as possible
2. Optimize memory usage for maximum bandwidth
3. Maximize occupancy to hide latency
4. **Optimize instruction usage for maximum throughput**



Optimize Instruction Usage

- Strategies:
  - Minimize use of low-throughput instructions
  - Use high precision only where necessary
  - Avoid divergent kernels (*with different instruction sequences*)

    Ex: if (threadId.x == 0) then action2
- There are two types of math operations:
  - **Faster, but lower accuracy**
    - native hardware implementation: native_sin(x)
    - half precision: half_sin(x), fast_normalize
  - **Slower, but higher accuracy** (software implementation)
    - Examples: sin(x), exp(x), pow(x,y)

Features Available in Device Code

- Standard mathematical functions
  - native_sin, native_powr, native_tan, native_sqrt, etc.
- Mad (multiply and addition) operation: *mad(floatn a, floatn b, floatn c)*

  result = a * b + c
  - Implementation: *fma*
  - Fast: *mad* ← not available on OpenCL, just CUDA
- Integer atomic operations in global memory
  - atomicAdd, atomicMin, atomicAnd, atomicCAS, etc.

Compiler adjustments

- Use the *-cl-mad-enable* compiler option.
  - The mad computes a * b + c with reduced accuracy and can lead to large performance gains

- The *-cl-fast-relaxed-math* compiler option enables many aggressive compiler optimizations:
  - *-cl-unsafe-math-optimizations* – no error checking
  - *-cl-finite-math-only* – no infinite or NaN values

Profiling

Profiling

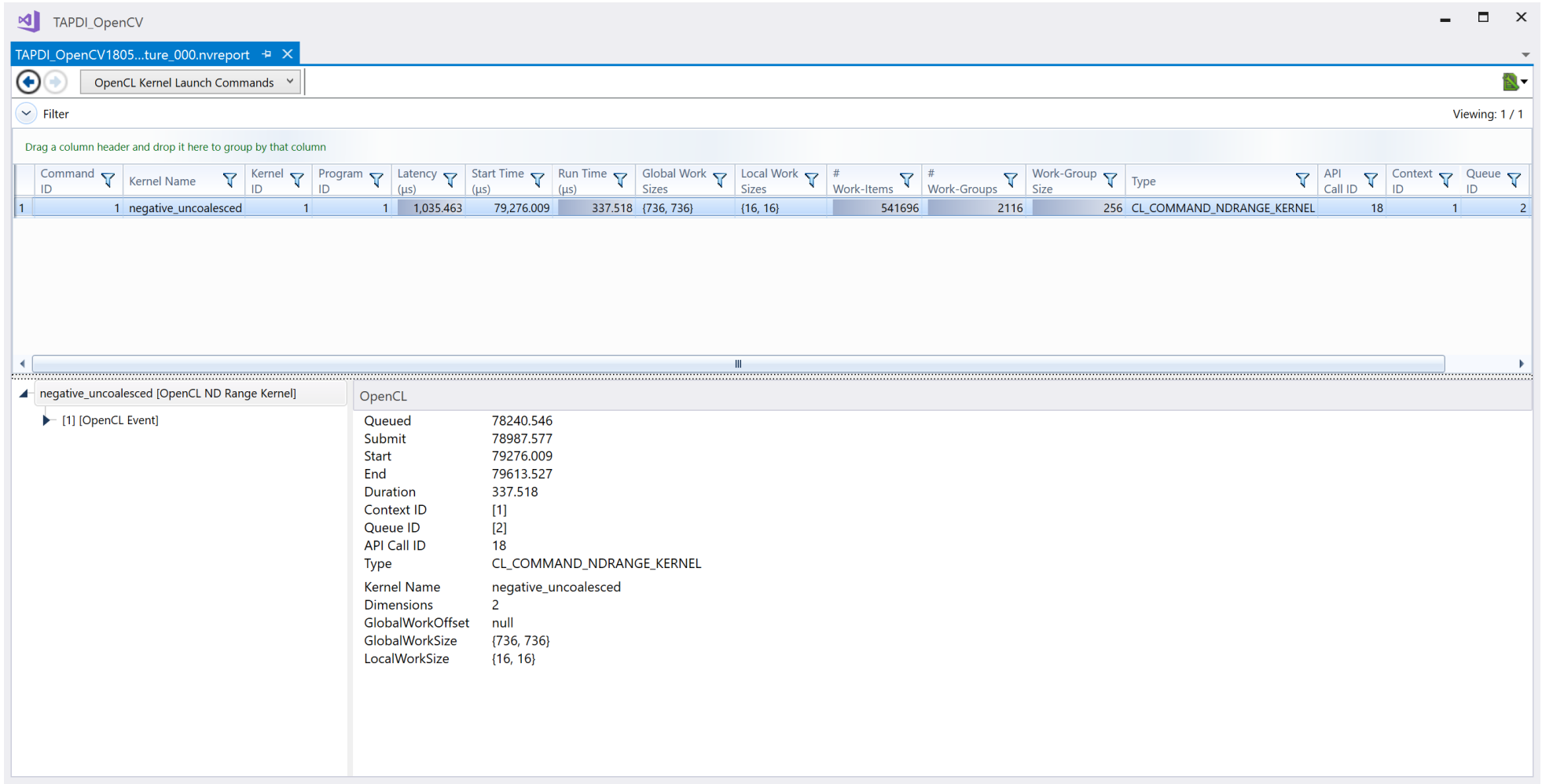
- Are a set of tools that helps you to optimize your code and identify code bottlenecks;

- It allows you to analyse in depth:
  - Memory usage (by type, including cache)
  - Code usage (single or double precision, integer, etc.)
  - Identify divergence in code execution
  - Identify the most problematic kernels
  - Code stall reasons

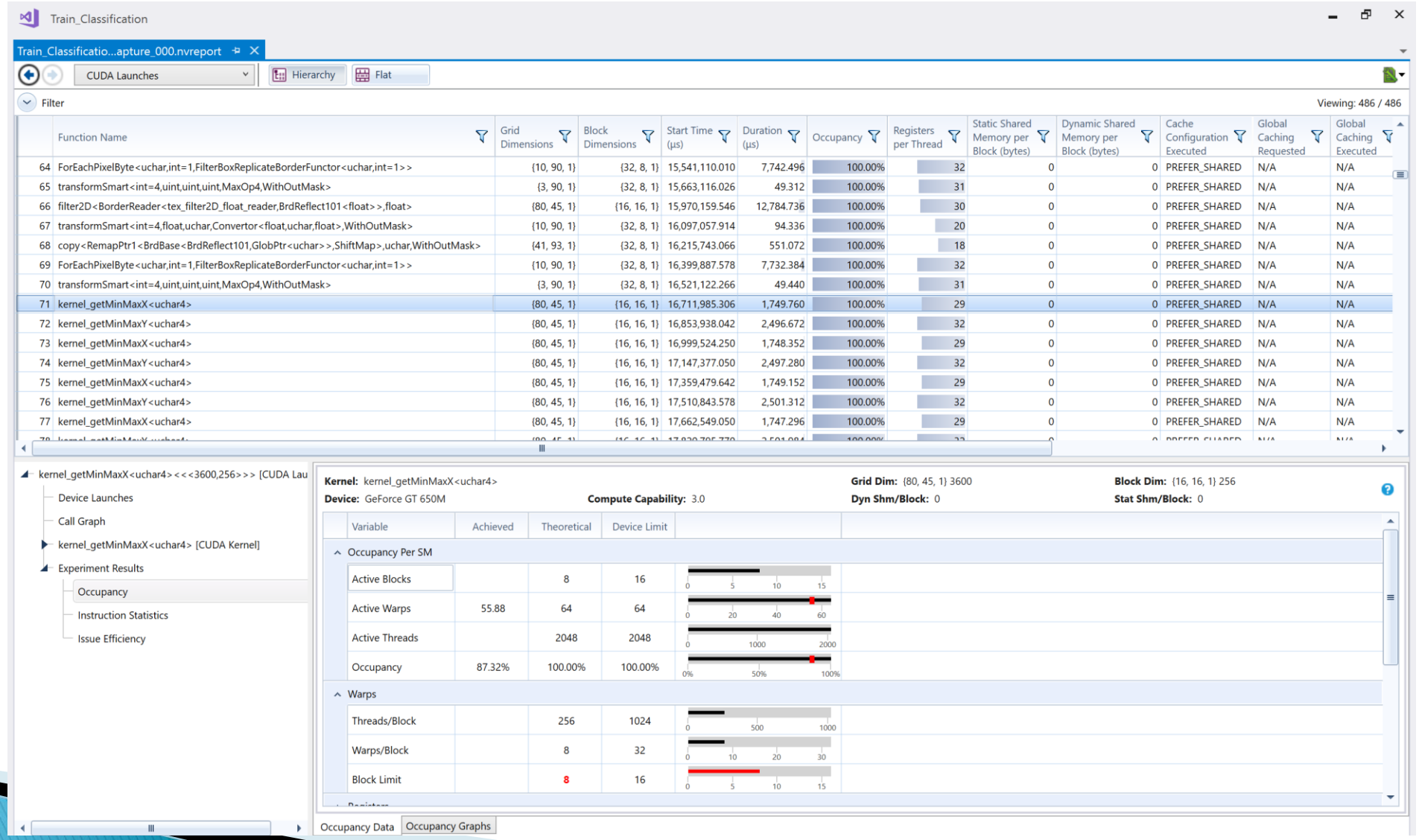
Vendors

- Profiling OpenCL:
  - Intel® SDK for OpenCL™ Applications
  - AMD Accelerated Parallel Processing (AMD APP) Software
  - **NVIDIA Nsight Visual Studio Edition**

Nsight – OpenCL Profile Output



Nsight – Just for CUDA



Nsight – Just for CUDA

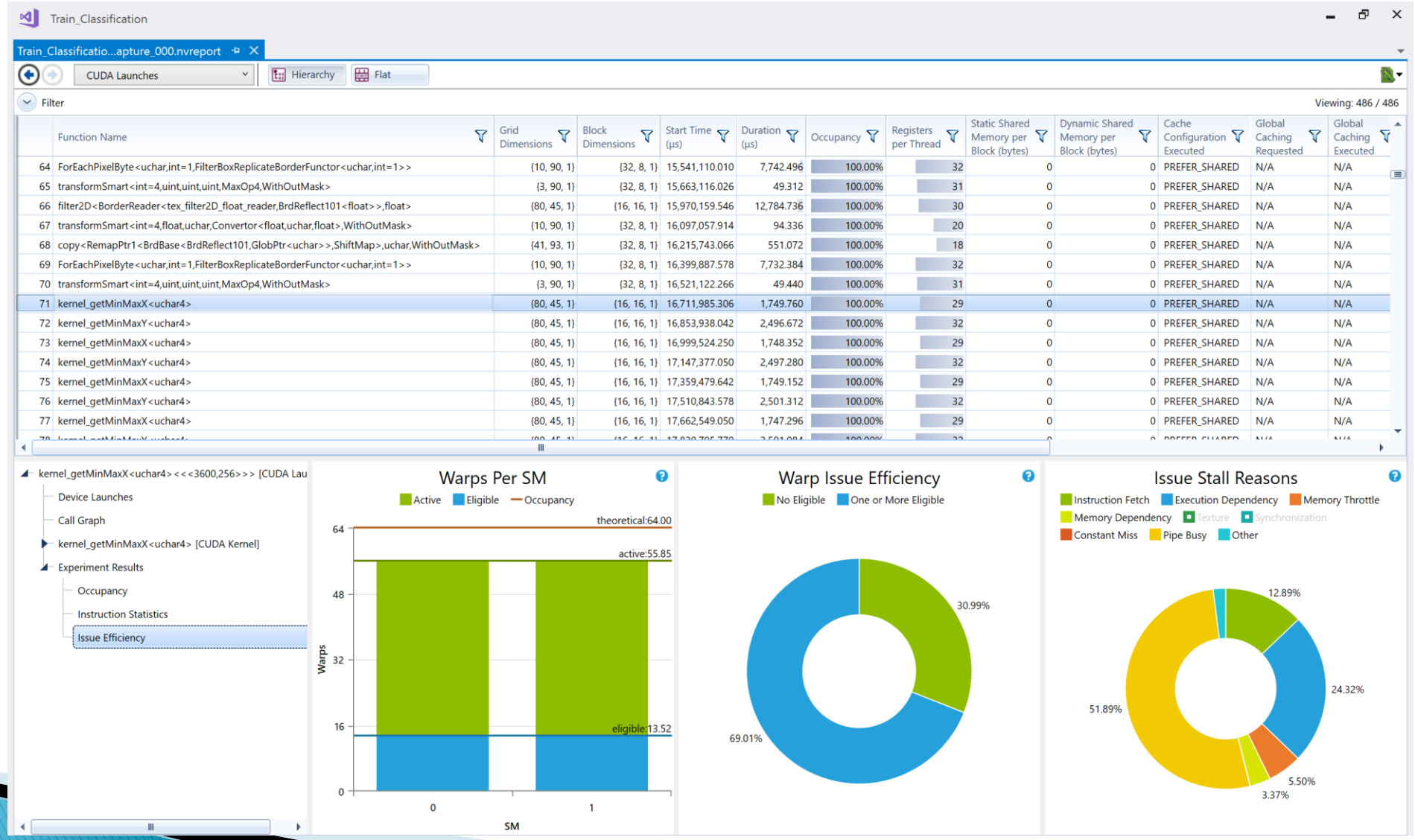
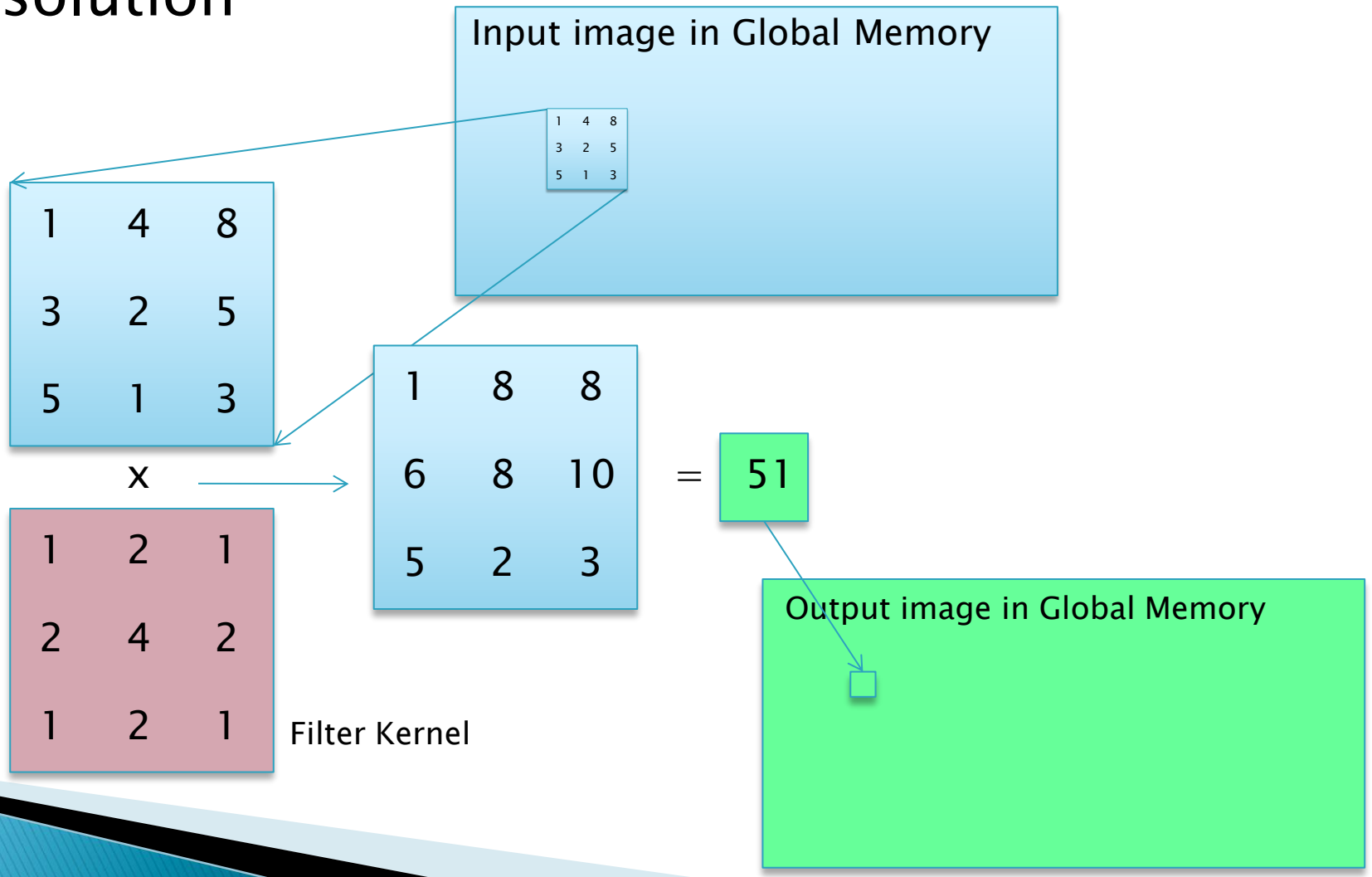


Image Processing

- Parallelizing Techniques

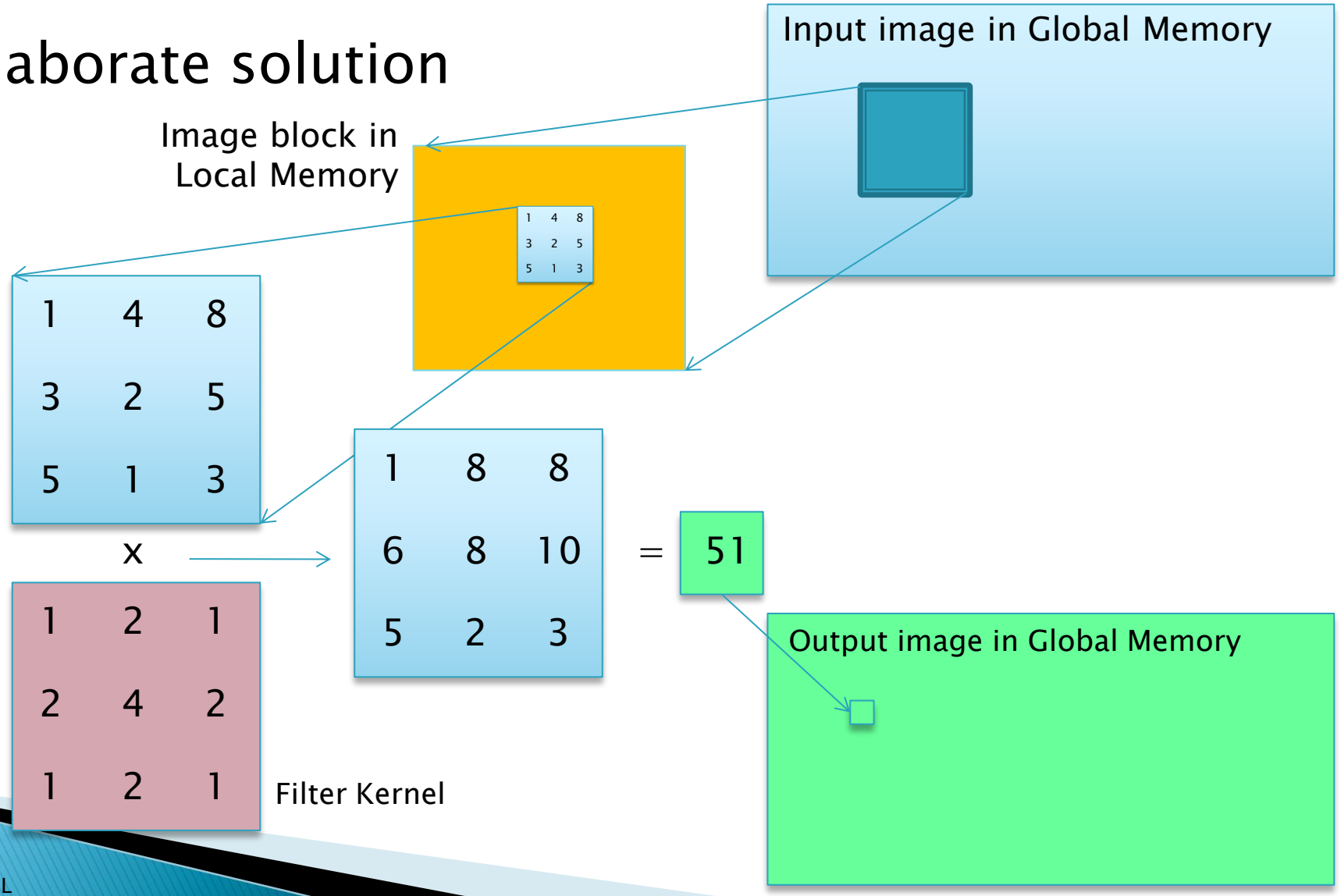
Convolution filter implementation

- Simplest solution



Convolution filter implementation

- More elaborate solution

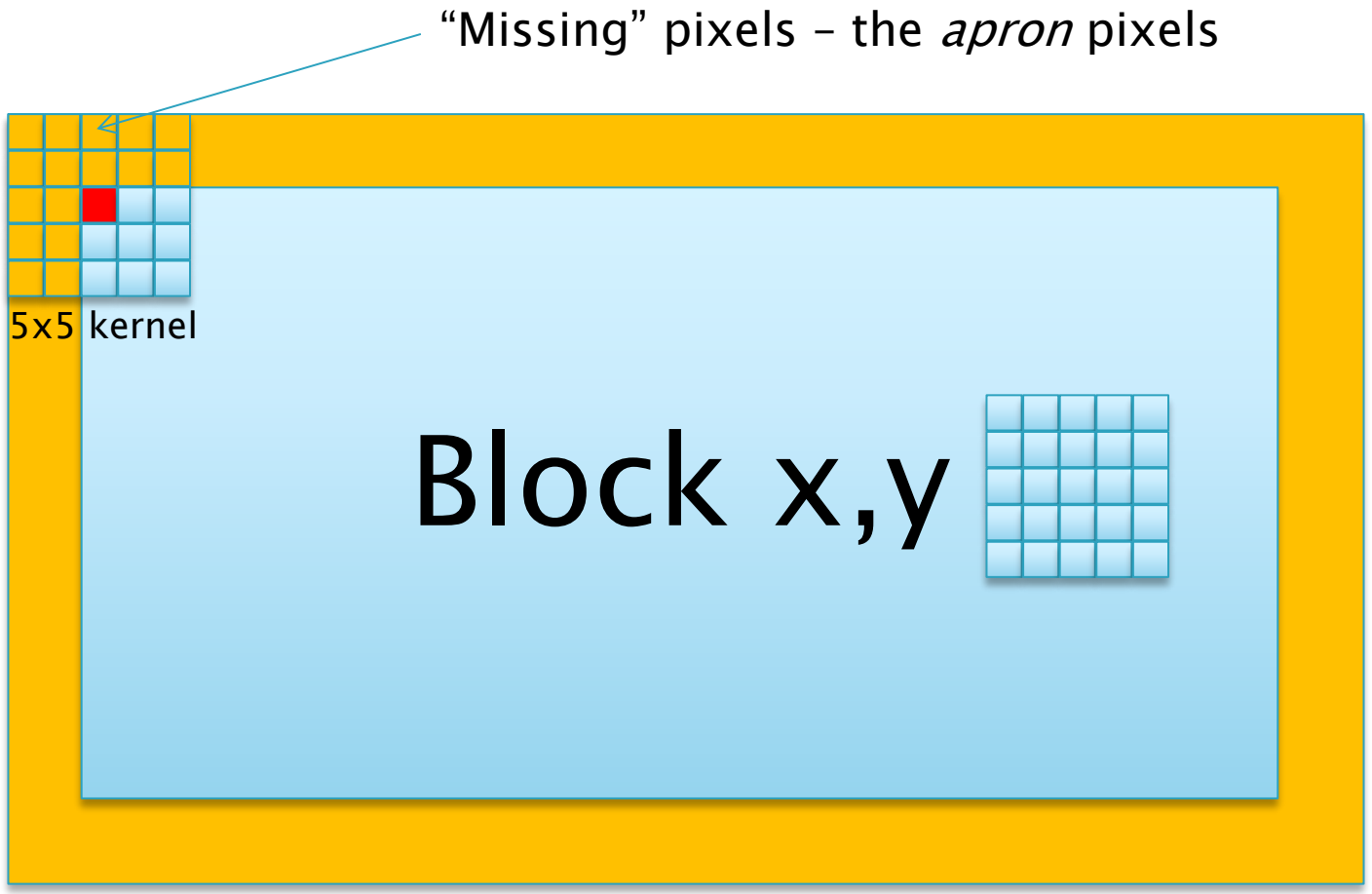


The image subdivision

Input Image

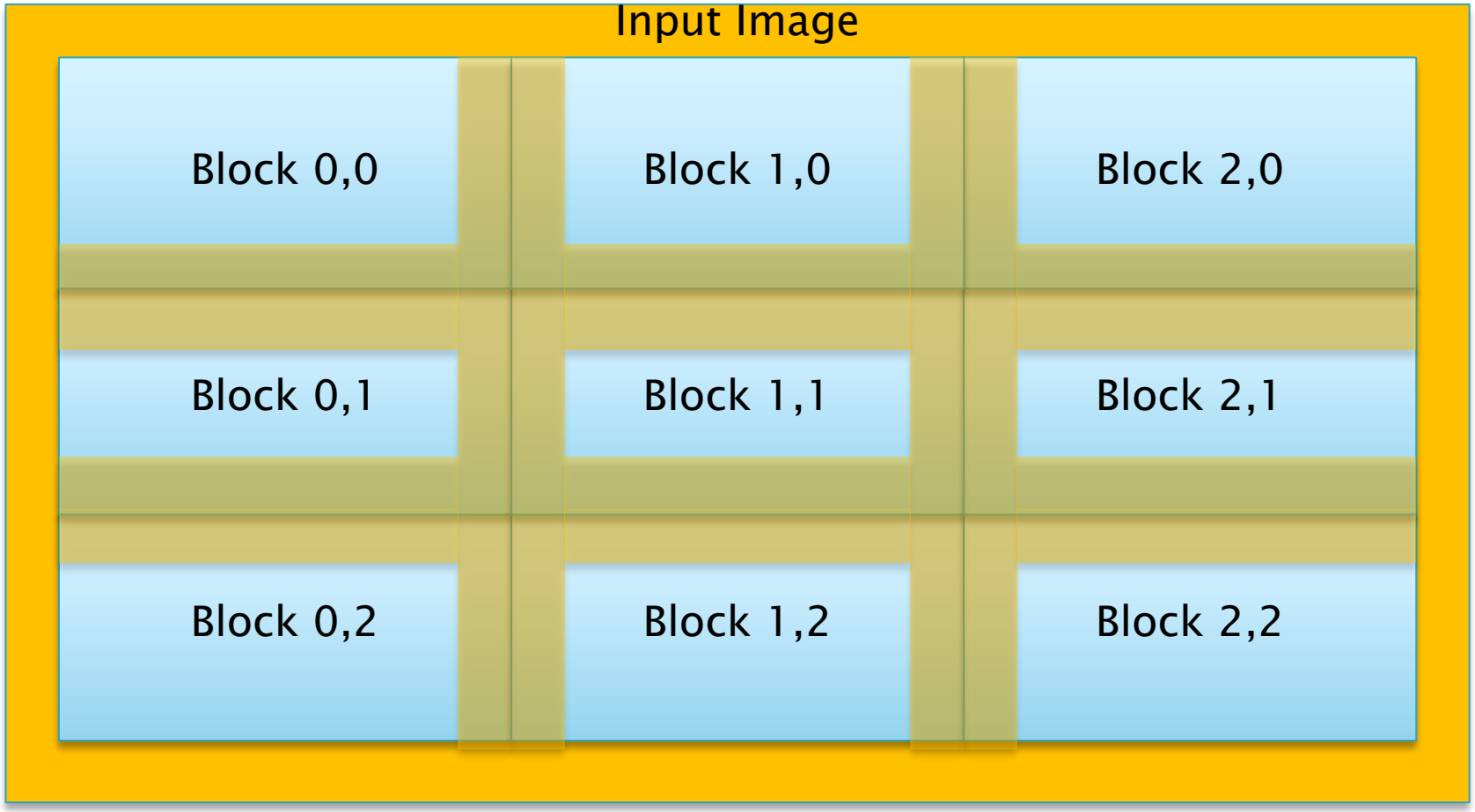
Block 0,0	Block 1,0	Block 2,0
Block 0,1	Block 1,1	Block 2,1
Block 0,2	Block 1,2	Block 2,2

The *apron* pixels



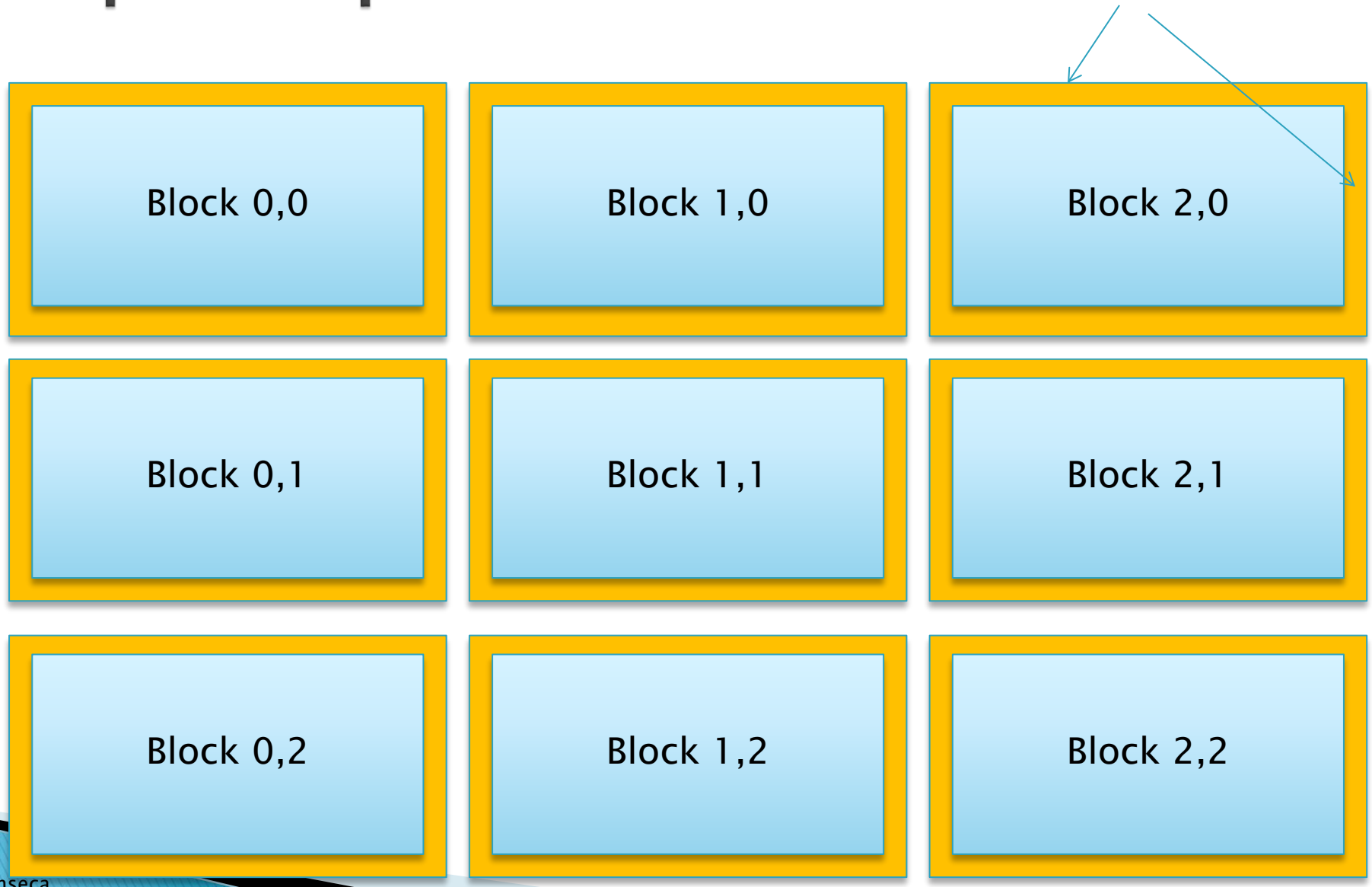
Therefore, each block must load into local memory the pixels to be filtered plus the *apron* pixels

Blocks plus Apron



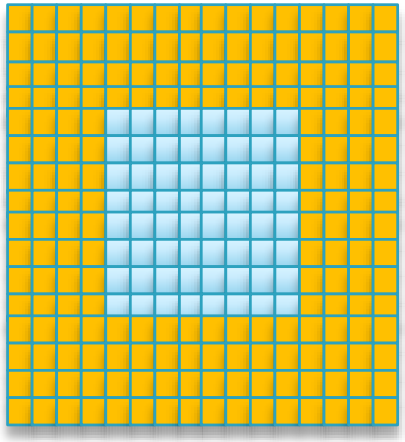
Blocks plus Apron

The apron in the edges can either be clamped to the pixels in the edge or set to zero



How to load the *apron* ?

- **1st solution**: use one thread for each pixel, including the apron pixels
  - Facilitates computation
  - Creates a lot of idle threads (apron threads will be idle during filter calculation) wasting important resources



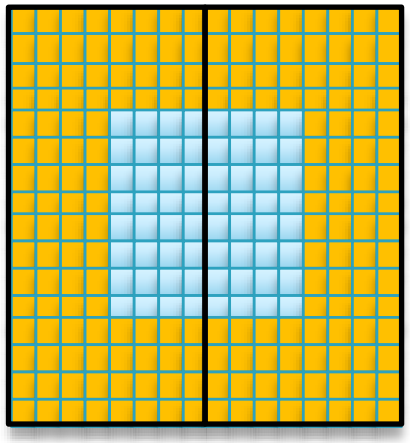
Example

- A 8x8 block (64 pixels) and a 9x9 kernel filter requires: 16x16=256 threads (4 pixels apron)
- Only 64/256 threads are active!!! (25%)

Note that the number of threads in a work-group should be a multiple of the warp size (32 threads on NVidia)

How to load the *apron* ?

- **2nd solution**: each thread loads several pixels
  - More complex computation
  - Reduces the number of idle threads
  - Depending on the kernel/block sizes treads might be idle on loading or computation phases

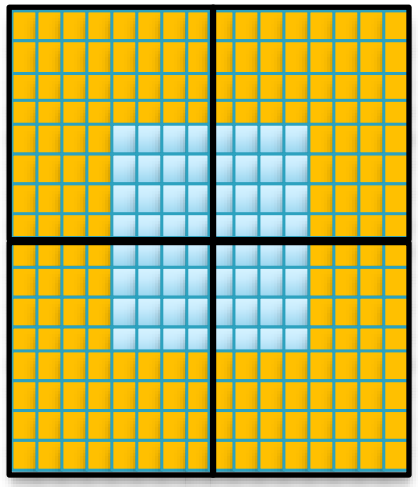


Example

- Each thread loads 2 pixels (16/2 = 8)
- A 8x8 block and a 9x9 kernel filter takes 16x8=128 threads (4 pixels apron)
- 64/128 threads are active!!! (50%)

How to load the *apron* ?

- 3rd solution: each thread loads more pixels
  - Even more complex computation
  - Reduces the number of idle threads
  - Depending on the kernel/block sizes treads might never be idle



Example

- Each thread loads 2x2 pixels (4 pixels)
- A 8x8 block and a 9x9 kernel filter takes 8x8=64 threads (4 pixels apron)
- 64/64 threads are active!!! (100%)

Image Processing example

- Negative

Image Negative

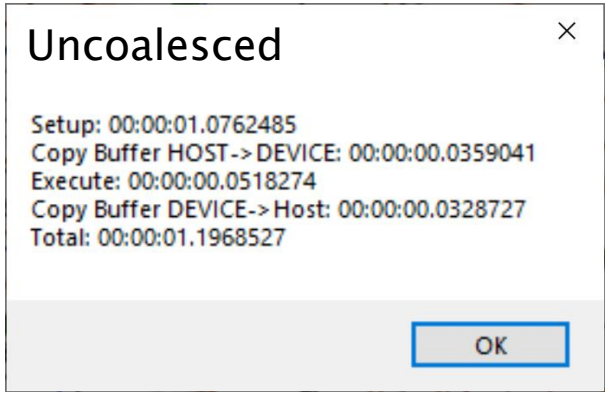
- Kernel using RGBA image (uchar4)

```
__kernel void negative_coalesced(
        __global uchar4* image,
        int w, int h, int padding,
        __global uchar4* imageOut)
{
  int i = get_global_id(0);
  int y = get_global_id(1);
  int idx = y * (w + padding/4) + x ;
  uchar4 neg = (uchar4)(255,255,255,0);

  if ((x < w) && (y < h)) {
    imageOut[idx] = neg - image[idx];
  }
}
```

PYTHON

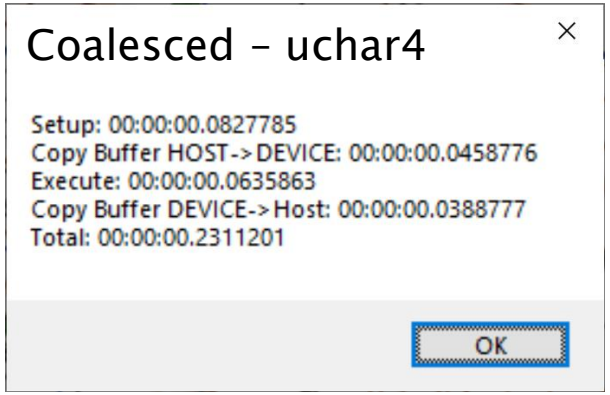
C#

- Copy H->D - 0.13183
- Execute - 0.01400
- Copy D->H - 0.04305



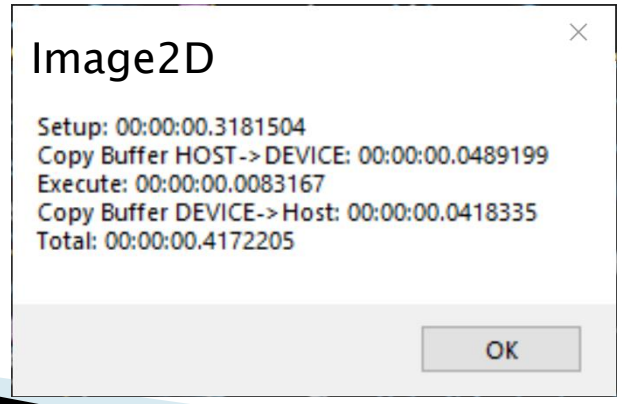

```
__kernel void negative_uncoalesced(__global uchar* image,
    int w, int h, int padding, __global uchar* imageOut) {
    int i = get_global_id(0);
    int y = get_global_id(1);
    int idx = x*3 + y * (w*3 + padding);
    if ((x < w) && (y < h)) {
      imageOut[idx] = 255 - image[idx];
      imageOut[idx+1] = 255 - image[idx+1];
      imageOut[idx+2] = 255 - image[idx+2];
    }
}
```

- Copy H->D - 0.15961
- Execute - 0.01540
- Copy D->H - 0.05450



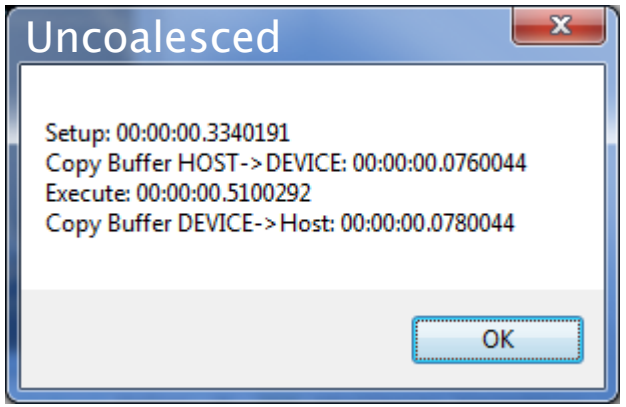

```
__kernel void negative_coalesced(__global uchar4* image,
    int w, int h,  int padding, __global uchar4* imageOut){
    int i = get_global_id(0);
    int y = get_global_id(1);
    int idx = y * (w + padding/4) + x ;
    uchar4 neg = (uchar4)(255,255,255,0);

    if ((x < w) && (y < h)) {
      imageOut[idx] = neg - image[idx];
    }
}
```

- Copy H->D - 0.18767
- Execute - 0.00706
- Copy D->H - 0.05043

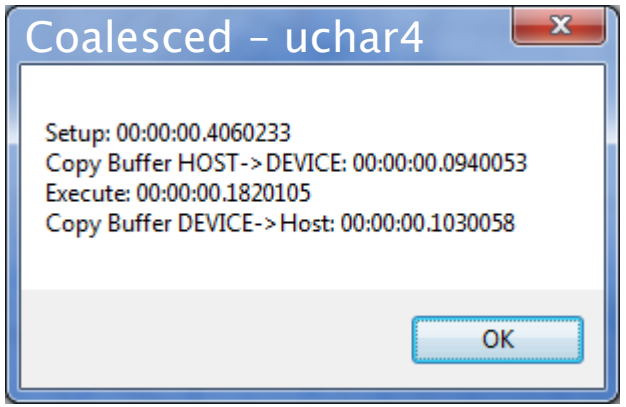



```
__kernel void negative_image2D(__read_only image2d_t image,
    __write_only image2d_t imageOut,  int w, int h) {
    int iX = get_global_id(0);
    int iY = get_global_id(1);
    uint4 neg = (uint4)(255, 255, 255 , 255);

    if ((iX >= 0)&&(iX < w) && (iY >= 0)&&(iY < h)) {
      uint4 pixelV = read_imageui( image, sampler, (int2)(iX,iY));
      write_imageui( imageOut, (int2)(iX,iY) , neg - pixelV );
    }
}
```

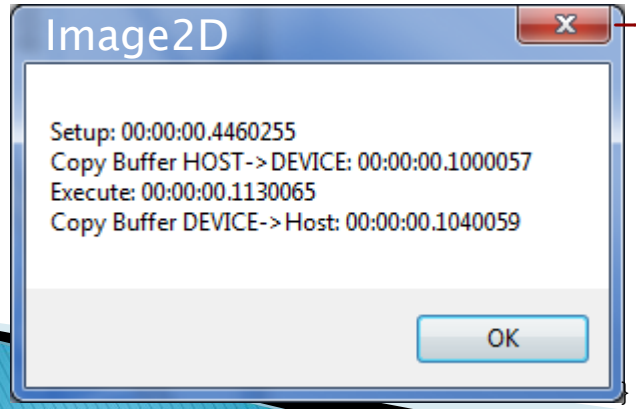
OpenCV
- execute - 0.14373




```
__kernel void negative_uncoalesced(__global uchar* image,
    int w, int h, int padding, __global uchar* imageOut) {
    int i = get_global_id(0)+get_global_id(1)*get_global_size(0)
    int idx = i*3 + (i/w) * padding;
    if (i < w*h) {
      imageOut[idx] = 255 - image[idx];
      imageOut[idx+1] = 255 - image[idx+1];
      imageOut[idx+2] = 255 - image[idx+2];
    }
}
```




```
__kernel void negative_coalesced(__global uchar4* image,
    int w, int h,  int padding, __global uchar4* imageOut){
    int i = get_global_id(0)+get_global_id(1)*get_global_size(0);
    int idx = i + (i/w) * padding/4 ;
    uchar4 neg = (uchar4)(255,255,255,0);

    if (i < w*h) {
      imageOut[idx] = neg - image[idx];
    }
}
```




```
__kernel void negative_image2D(__read_only image2d_t image,
  __write_only image2d_t imageOut,  int w, int h) {
  int iX = get_global_id(0);
  int iY = get_global_id(1);
  uint4 neg = (uint4)(255, 255, 255 , 255);

  if ((iX >= 0)&&(iX < w) && (iY >= 0)&&(iY < h)) {
    uint4 pixelV = read_imageui( image, sampler, (int2)(iX,iY));
    write_imageui( imageOut, (int2)(iX,iY) , neg - pixelV );
  }
}
```