High Performance Computing

GPGPU

Local Vs. Global memory

Image Vs. Buffer memory

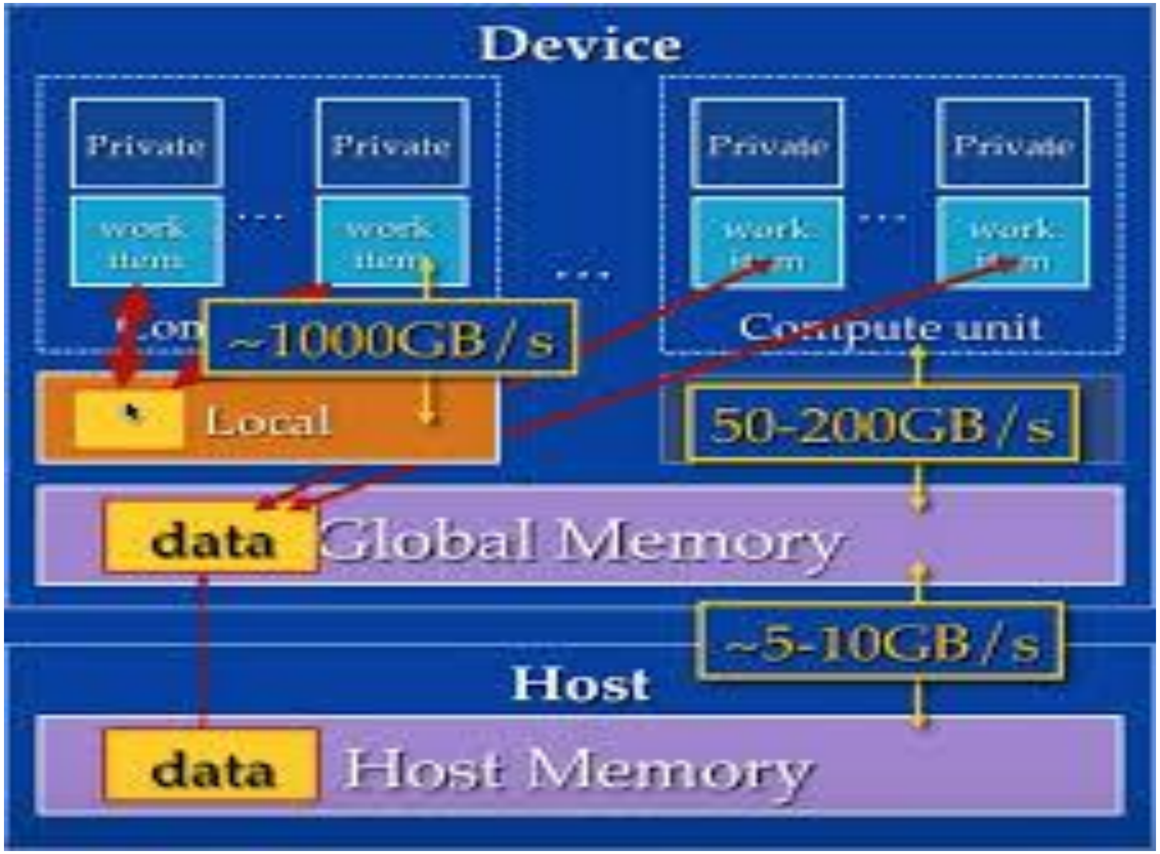
*OpenCL was selected as a parallel programming for these slides

[OpenCL Overview - The Khronos Group Inc]

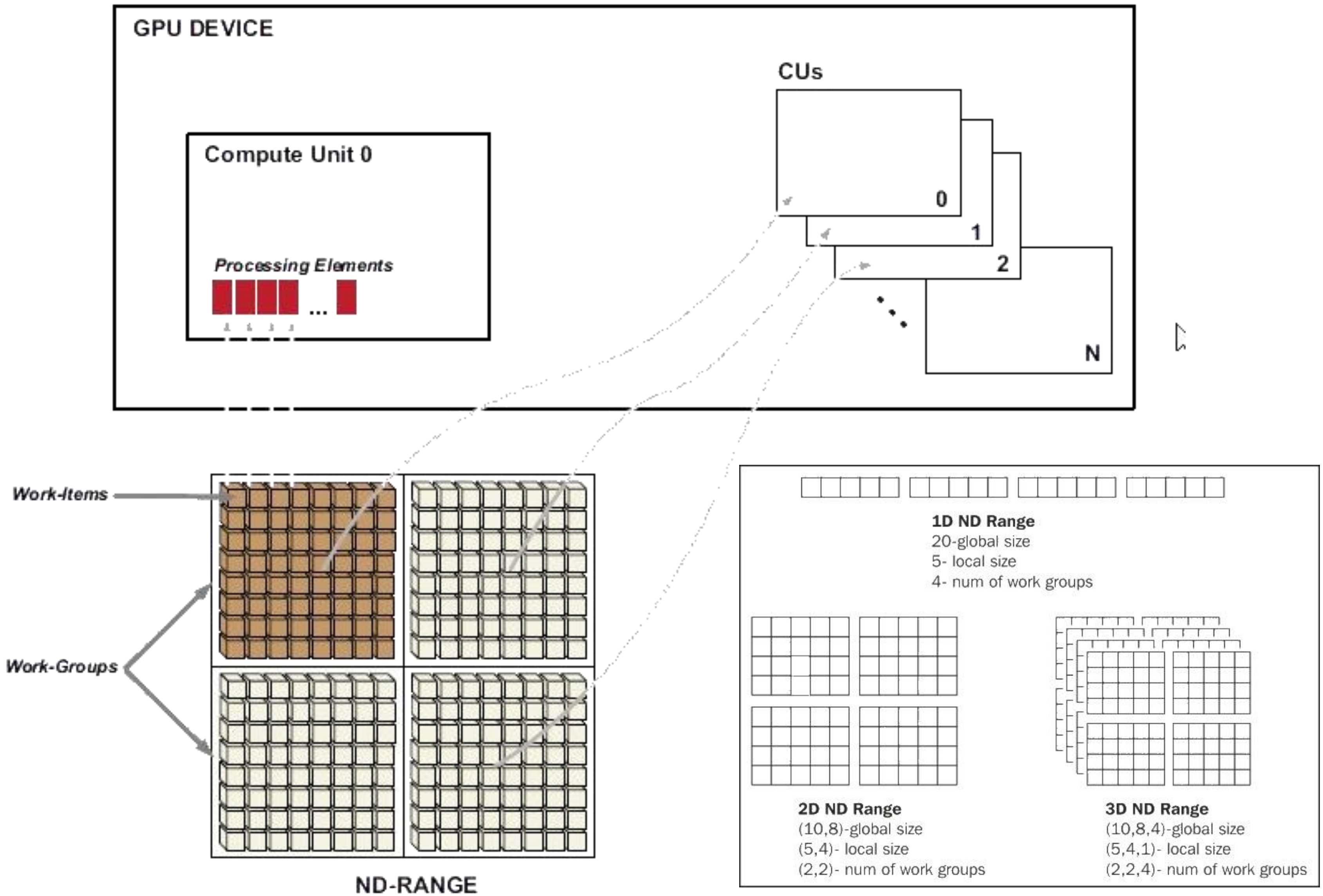
[Khronos OpenCL Registry - The Khronos Group Inc]

Rationale

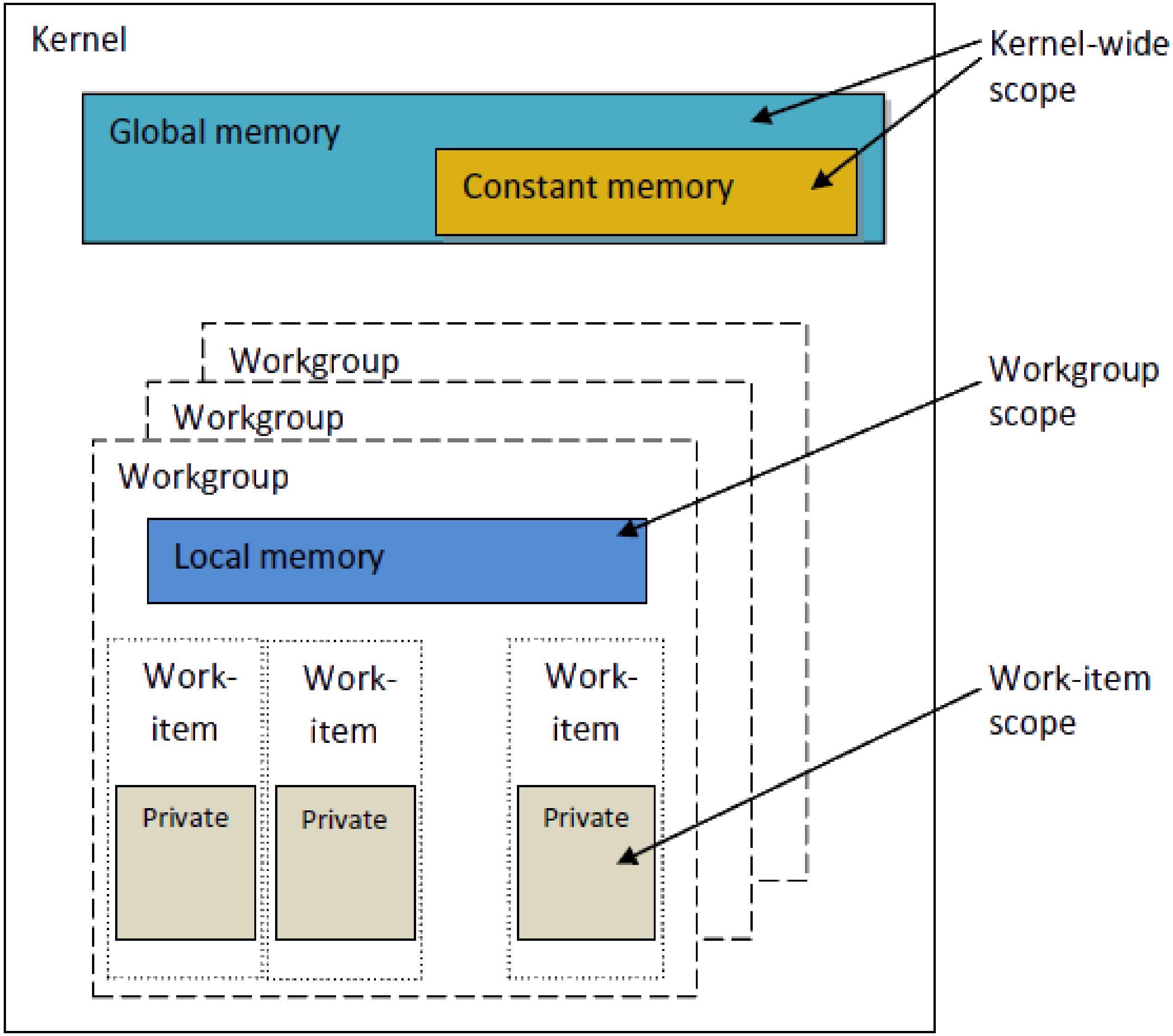
- Increase the achieved frame per second (fps)
- Reduce the end2end latency time
- Root cause:



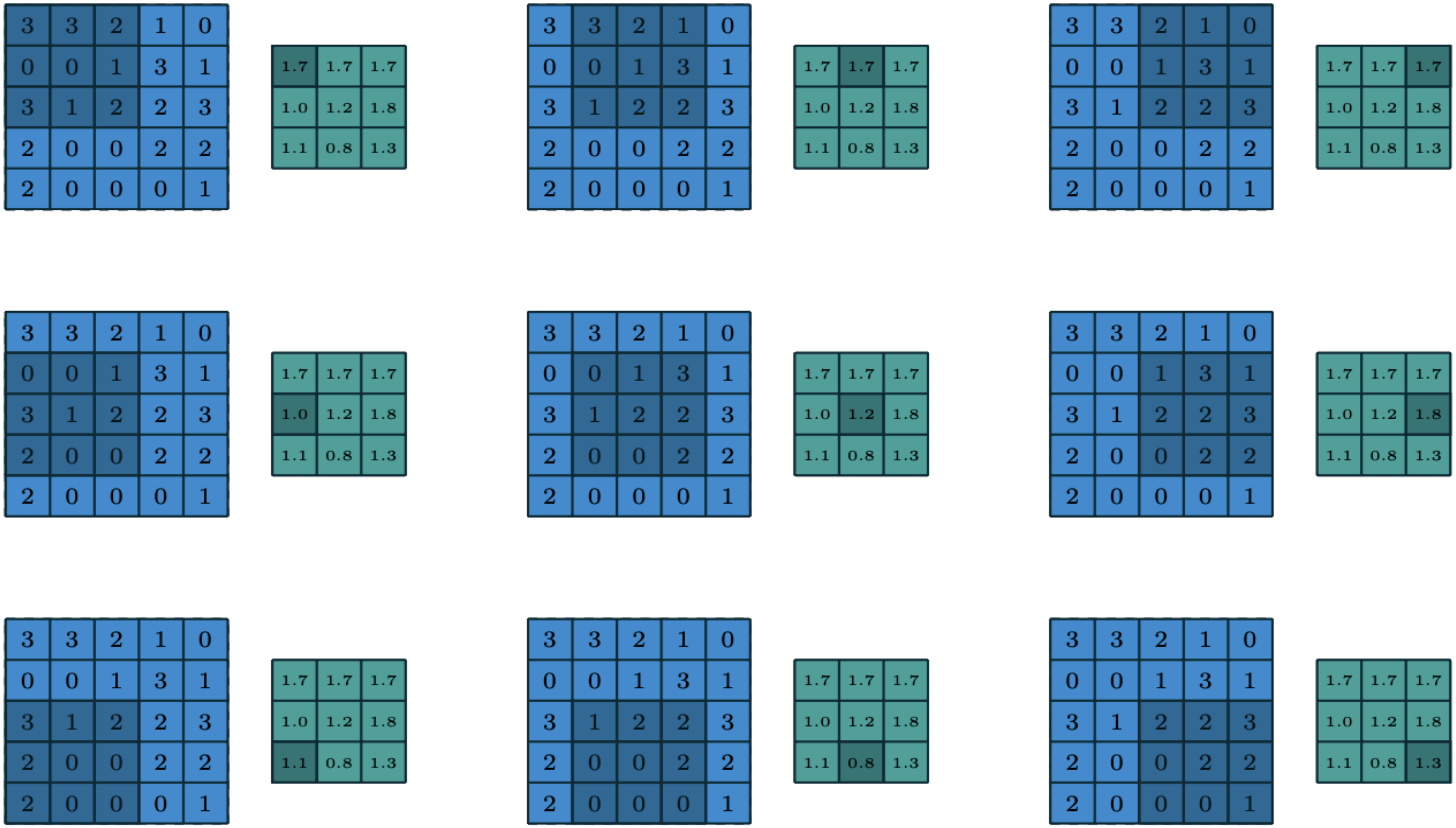
ND Range


GPU DEVICE
CUs
Compute Unit 0
Processing Elements
...
0
1
2
N
Work-Items
Work-Groups
ND-RANGE
1D ND Range
20-global size
5- local size
4- num of work groups
2D ND Range
(10,8)-global size
(5,4)- local size
(2,2)- num of work groups
3D ND Range
(10,8,4)-global size
(5,4,1)- local size
(2,2,4)- num of work groups

Memory Model



Test case - Convolution



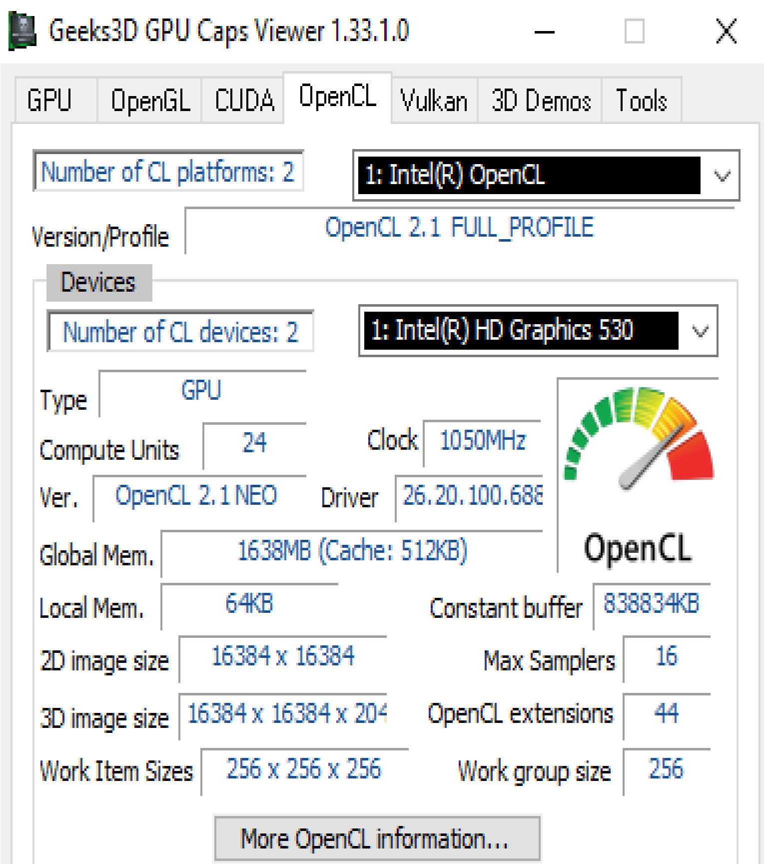
Which memory cell is the most accessed? How much times? How many of these cells?

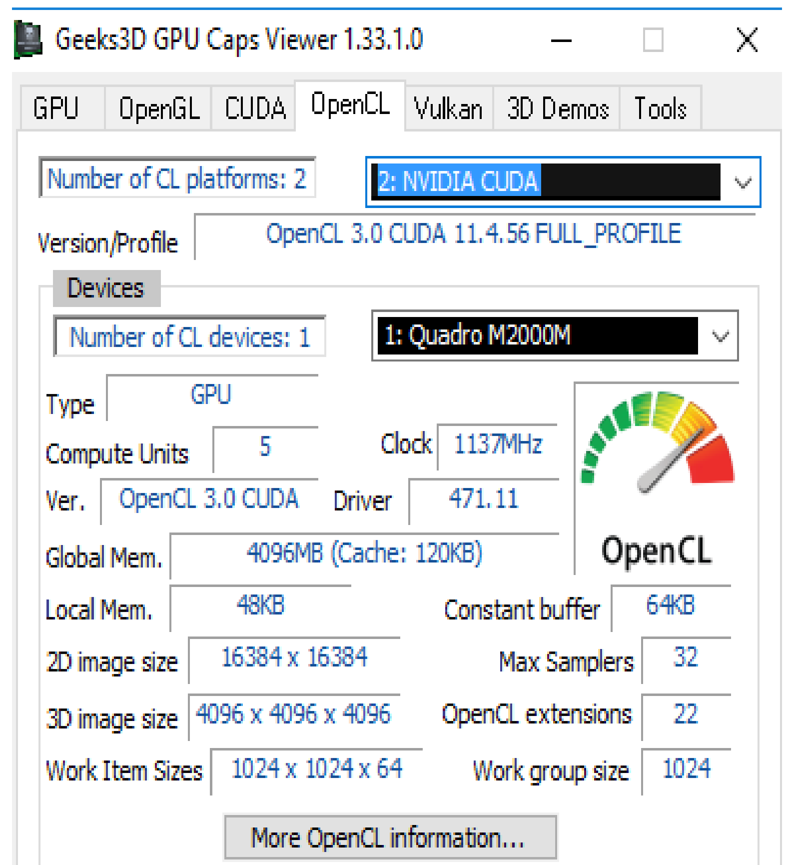
Test Case - Solution

- Local (OpenCL)\Shared (CUDA) memory instead of global memory
- Private memory instead of global memory for small buffer
- Unique memory type, such as image (OpenCL) which have a smart cache instead of regular buffer

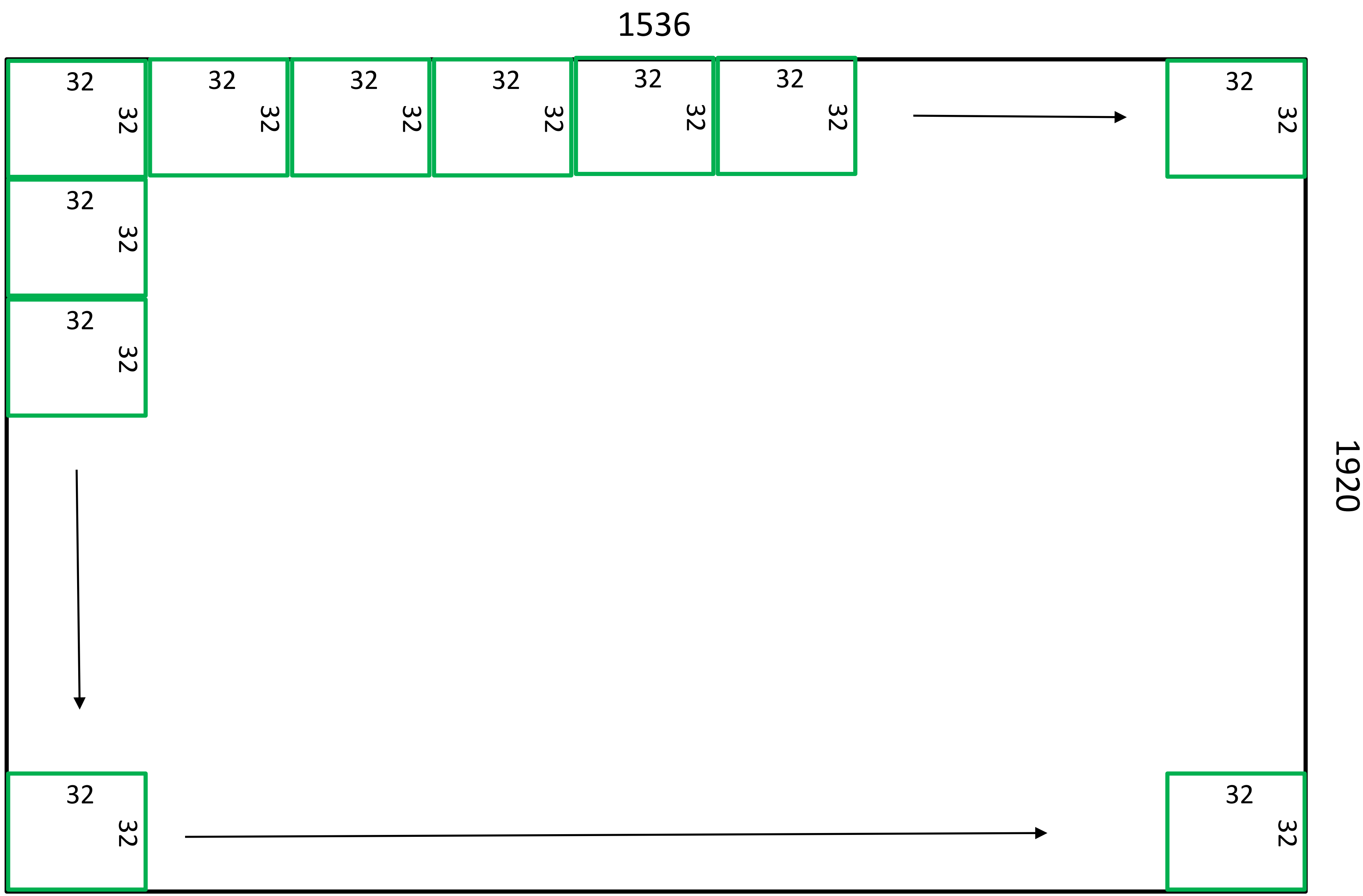
Test Case – General Information

- **How we can know how much memory we have?**
- **How much memory we can use in parallel?**





Test case – Divided to Work Groups



Test case – Divided to Work Groups

ND range information:

- 2D
- Y global size is 1920
- X global size is 1536
- Max work items for all range is 1024 (can be divided to axis 32 X 32 for example)
- Work groups count:
  - Y – 1920 / Max work items for Y (32) – 60
  - X – 1536 / Max work items for X (32) – 48

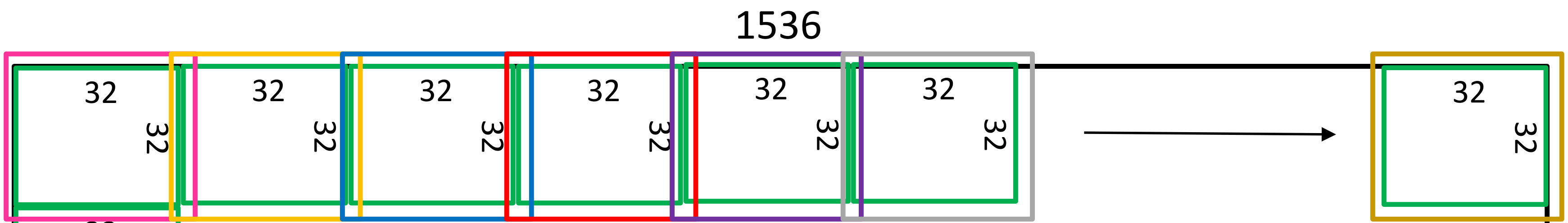
- **Each work item mapped to a dedicated frame element (pixel\voxel etc.)**

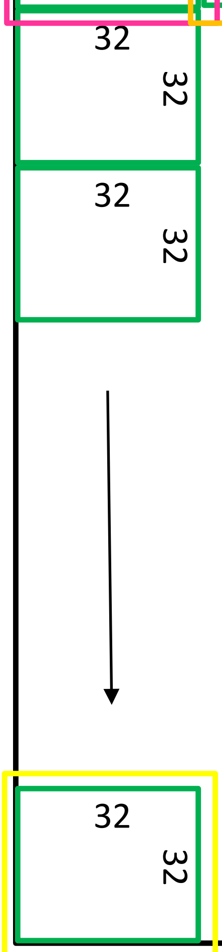
Test case – Divided to Work Groups

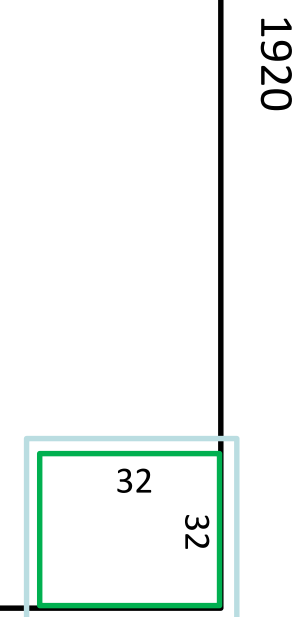
- **As we said early, local memory can be shared by the work items which belongs to the same work group only**
- **Each work group has its own local memory**
- **There is a limit of how much local memory can be parallel used by the compute units (entire device):**
  - Cause a limitation of how much work groups will parallel work
- **The purpose of each work item of each workgroup is to convolve its mapped element**

- **Convolution parameters:**
  - Filter size – F X F, F is odd value
  - With padding – duplicated on borders, its size is F / 2 in each border
  - Stride - 1

Test case – Divided to Work Groups





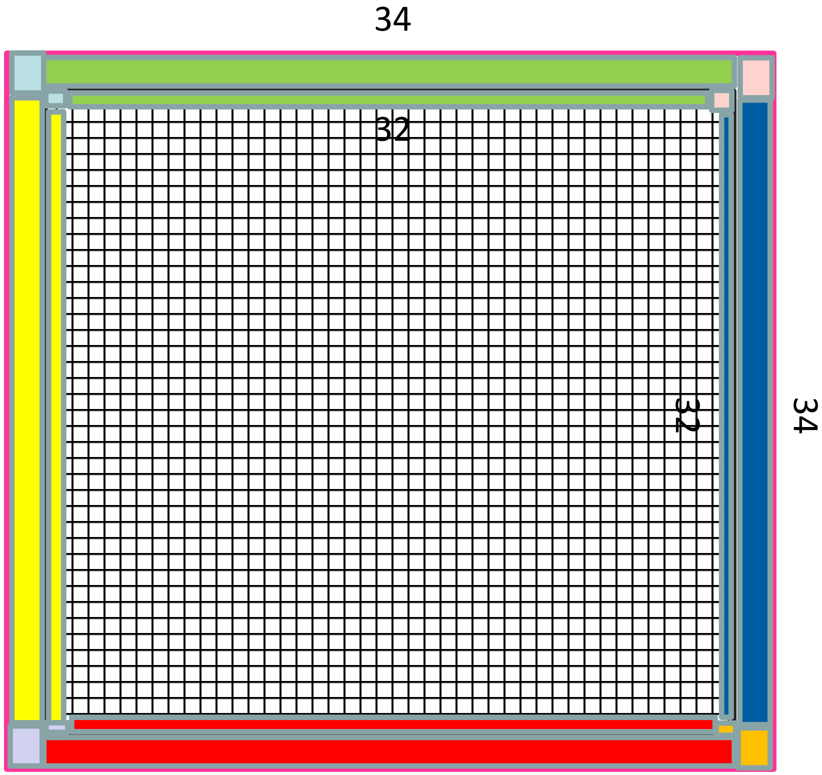


- Each work group items loads the relevant elements from global to local memory
  - Several work items loads more than one element
  - Barrier is used to sync loading process between all work group items
  - After all data resides in he local memory the convolution start
  - Results are store in output global memory
- There are 8 scenarios of padding:
  - (X < 0, Y < 0) - most upper left pixel (Actually the first pixel in frame)
  - (0 =< X < FRAME_WIDTH, Y < 0) - first row in the same X
  - (FRAME_WIDTH =< X, Y < 0) - most upper right pixel
  - (FRAME_WIDTH =< X, 0 =< Y < FRAME_HEIGHT) - last column in the same Y
  - (FRAME_WIDTH =< X, FRAME_HEIGHT <= Y) - most lower right pixel
  - (0 =< X < FRAME_WIDTH, FRAME_HEIGHT =< Y) - last row in the same X
  - (X < 0, FRAME_HEIGHT =< Y) - most lower left pixel
  - (X < 0, 0 =< Y < FRAME_HEIGHT) - first column in the same Y

Test case – Load from global to local

- Using work item global index each work item know which element to load
  - int iYPos, iXPos = get_global_id(0u), get_global_id(1u);
  - int localY,localX = get_local_id(0), get_local_id(1);
  - int globalLoadY, globalLoadX = iYPos \iXPos - CONV_FILTER_HALF_SIZE;

CONV_FILTER_HALF_SIZE - CONV_FILTER_SIZE >> 1
BLOCK_SIZE – 32
LOCAL_BUFFER_SIZE - BLOCK_SIZE + (CONV_FILTER_HALF_SIZE << 1)



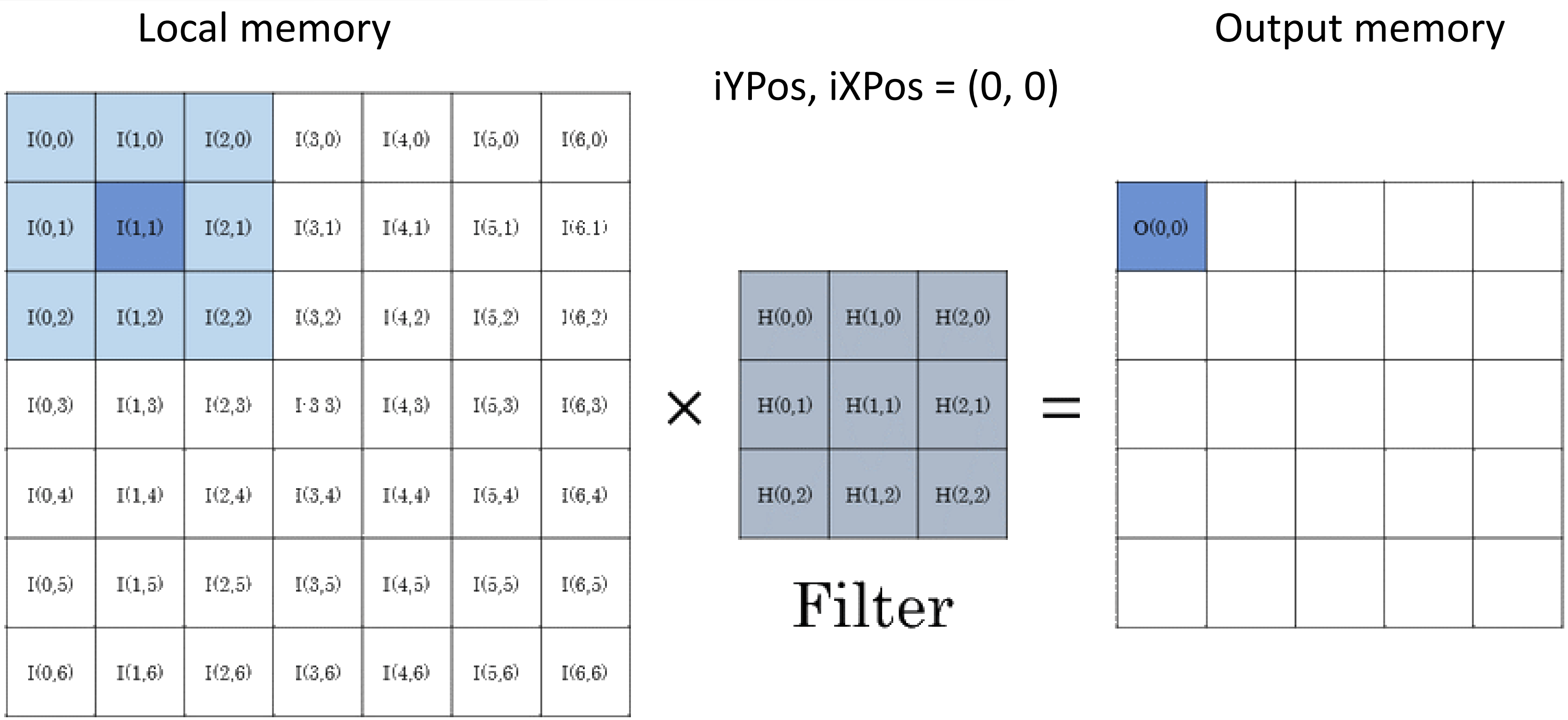
- Loading algo (pseudo):

```
While localX < LOCAL_BUFFER_SIZE
    globalLoadY = iYPos - CONV_FILTER_HALF_SIZE
    localY = get_local_id(0)
    While localY < LOCAL_BUFFER_SIZE
        If (globalLoadY, globalLoadX) inside the frame borders:
            globalCoords.Y\X = globalLoadY\globalLoadX
        else:
            Check which padding area (colored) is active:
            Set globalLoadY & globalLoadX based on the colored area
            which required for padding and update globalCoords.
        localMem(localY, localX) =
          read_imagef\i\ui(image, sampler, globalMem(globalCoords))

        localY\globalLoadY   += BLOCK_SIZE
     end loop
     localX\globalLoadX   += BLOCK_SIZE
end loop
Barrier
```

```
const sampler_t sampler =
     CLK_NORMALIZED_COORDS_FALSE | //Natural coordinates
     CLK_ADDRESS_CLAMP | //Clamp to zeros
     CLK_FILTER_NEAREST; //Don't interpolate
```

Test case – The convolution



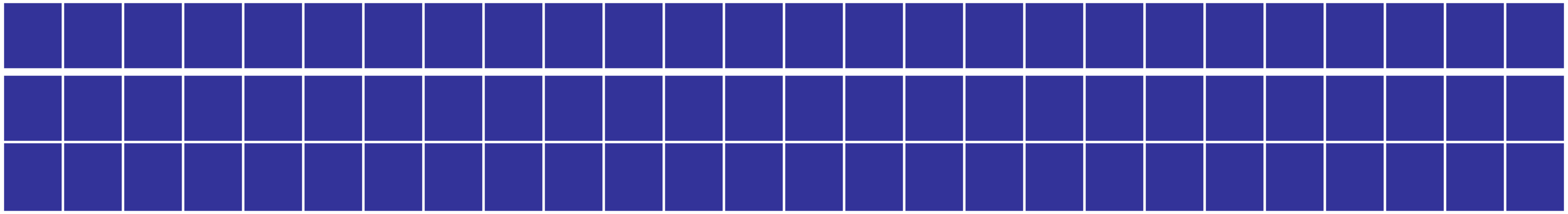
- Convolution:
  - The best results achieved from manual multiplication code, but it reasonable only up to filter size of 7X7 – 49 lines (for 9x9 there will be 81 lines…etc.)
  - For all other filter sizes just write a double loop and add the #pragma unroll to each one

Test case – The convolution (Con't)

```
localY = get_local_id(0) + CONV_FILTER_HALF_SIZE;
localX = get_local_id(1) + CONV_FILTER_HALF_SIZE;
float pixelFiltered = 0.0f;
int filterRowIdx = 0;
int outPos = iYPos * FRAME_WIDTH + iXPos;
#pragma unroll
for (int row = localY - CONV_FILTER_HALF_SIZE; row <= localY + CONV_FILTER_HALF_SIZE; row++)
{
        int weightBufferOffset = filterRowIdx * CONV_FILTER_SIZE;
        #pragma unroll
        for (int col = localX - CONV_FILTER_HALF_SIZE; col <= localX + CONV_FILTER_HALF_SIZE; col++)
        {
                pixelFiltered += (localBuffer[row][col] * weightBuffer[weightBufferOffset++]);
        }
        filterRowIdx++;
}
imageOut[outPos] = pixelFiltered;
```

Image mem Vs. Buffer mem

- Buffer memory has a linear access which isn't rapid



- While image memory stored in a unique global memory called texture which is cached for rapid access

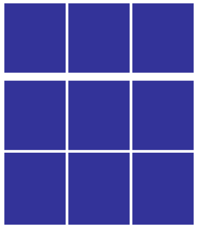


Image mem Vs. Buffer mem

Additionally:

- The functions used to read and write image data can be invoked without regard to how the pixel data is formatted, so long as the format is supported by OpenCL
- Special data structures called samplers make it possible to configure how color information is read from an image
- OpenCL provides functions that return image specific information, such as image's dimensions, pixel format and bit depth
- Was:

```
__kernel void Filter(__global float*  imageIn, __global float* imageOut)
```

- Is:

```
__kernel void FilterOptimized(
    __read_only image2d_t imageIn, __write_only image2d_t imageOut)
```

  - value = read_imagef(imageIn, smp, coords);
  - write_imagef(imageOut, outCoords, outValue);
  - Both work with float4 type
  - For one channel pixel (CL_R) set the float4 to – (R, 0.0f, 0.0f, 1.0f), OpenCL "skip" on irrelevant channels

Private mem

- **Small arrays like filter weights shall be placed in the private memory and not passed by the kernel launch and stored in the global memory**
  - Usually filter weights values are constant
  - For filter with reasonable small size such as – 3x3…7x7

- **In order to map them to the private memory just declare them inside the kernel body:**
  - Was:

```
__kernel void Filter(__global float *weightBuffer)
```

  - Is:

```
__constant float weightBuffer[9] =
{
 0.05f, 0.02f, 0.5f, 0.05f, 0.07f, 0.1f, 0.05f, 0.09f, 0.05f
};
```

Test case - results

	Conv V1	Conv optimized	improvement	Filter size	Loop unroll	Mem type	Conv count	Filter func name
Quadro M2000M	~320ms	~131ms	~60%	3x3	True	Buffer	50	FilterOptimized3X3
Quadro M2000M	~320ms	~151ms	~52%	3x3	False	Buffer	50	FilterOptimized
Quadro M2000M	~320ms	~93ms	~71%	3x3	True	Image	50	FilterOptimized3X3
Quadro M2000M	~320ms	~114ms	~64%	3x3	False	Image	50	FilterOptimized
Quadro M2000M	NA	~184ms	NA	7x7	True	Buffer	50	FilterOptimized7X7
Quadro M2000M	NA	~236ms	NA	7x7	False	Buffer	50	FilterOptimized
Quadro M2000M	NA	~149ms	NA	7x7	True	Image	50	FilterOptimized7X7
Quadro M2000M	NA	~211ms	NA	7x7	False	Image	50	FilterOptimized
Quadro RTX 3000	~118ms	~50ms	~57%	3x3	True	Buffer	50	FilterOptimized3X3
Quadro RTX 3000	~118ms	~53ms	~55%	3x3	False	Buffer	50	FilterOptimized
Quadro RTX 3000	~118ms	~34ms	~71%	3x3	True	Image	50	FilterOptimized3X3
Quadro RTX 3000	~118ms	~37ms	~68.6%	3x3	False	Image	50	FilterOptimized
Quadro RTX 3000	NA	~75ms	NA	7x7	True	Buffer	50	FilterOptimized7X7
Quadro RTX 3000	NA	~75ms	NA	7x7	False	Buffer	50	FilterOptimized
Quadro RTX 3000	NA	~59ms	NA	7x7	True	Image	50	FilterOptimized7X7
Quadro RTX 3000	NA	~61ms	NA	7x7	False	Image	50	FilterOptimized

Typical OpenCL Flow - Pseudo

Prepare resources:

- `clGetPlatformIDs(0, nullptr, &numPlatforms)`
- `clGetDeviceIDs(platformIds[i], CL_DEVICE_TYPE_ALL, 0, nullptr, &numDevices);`
- `cl_context_properties contextProperties[] = { CL_CONTEXT_PLATFORM, (cl_context_properties)platformIds[platformID], 0};`
  - `context = clCreateContext(contextProperties, numDevices, &deviceIds[platformID][deviceID], &pfn_notify, nullptr, &errNum);`
  - `clGetPlatformInfo, clGetDeviceInfo & clGetContextInfo` – Query all exist platforms, their devices and the created contexts supported capabilities
  - `commandQueue = clCreateCommandQueue(context, deviceIds[platformID][deviceID], NULL /*CL_QUEUE_PROFILING_ENABLE*/, &errNum);`
- `buffer = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, FRAME_INPUT_SIZE_BYTES, frame, &errNum);`
- `image = clCreateImage2D(context, CL_MEM_READ_WRITE, &imgInter, FRAME_WIDTH, FRAME_HEIGHT, 0, nullptr, &errNum);`

Typical OpenCL Flow - Pseudo

- **Analyze, compile, build and load OpenCL kernels:**
  - First, textually load the entire OpenCL *.cl text file into a std::string using *std::ifstream*
  - `program = clCreateProgramWithSource(context, 1, (const char**)&kernelsProgramSourceStr, &kernelsProgramLength, &errNum);`
  - `errNum = clBuildProgram(program, 1, &deviceIds[platformID][deviceID], "", nullptr, nullptr);`
    - In case `errNum` isn't CL_SUCCESS, call to: `clGetProgramBuildInfo(program, deviceIds[platformID][deviceID], CL_PROGRAM_BUILD_LOG, buildLogLength, szMsg, nullptr);`
- **Create OpenCL kernel and register its inputs:**
  - `kernel = clCreateKernel(program, kernelName, &errNum);`
  - `errNum = clSetKernelArg(kernel, 0, sizeof(cl_mem), &frameInput);`
  - `errNum = clSetKernelArg(kernel, 1, sizeof(cl_mem), &buffers[0]);`

Typical OpenCL Flow - Pseudo

Execute kernel:

- `std::size_t globalWorkSize[] = { FRAME_HEIGHT, FRAME_WIDTH };`
- `std::size_t localWorkSize[] = { BLOCK_SIZE, BLOCK_SIZE };`
-
```
errNum = clEnqueueNDRangeKernel(commandQueue, kernel,
2u, nullptr, globalWorkSize, localWorkSize,
0, nullptr, nullptr/*&myEvent*/);
```

For debug purpose:

-
```
errNum = clEnqueueReadImage(commandQueue,buffer,
CL_TRUE,origin,region,FRAME_WIDTH * sizeof(float),0,result,0,
nullptr, nullptr);
```
-
```
errNum = clEnqueueReadBuffer(commandQueue, buffer,
CL_TRUE, 0, FRAME_SIZE_BYTES, result, 0, nullptr, nullptr);
```

Release resources:

- `clReleaseKernel(kernel)`
- `clReleaseMemObject(buffer)`
- `clReleaseCommandQueue(commandQueue)`
- `clReleaseContext(context) & clReleaseProgram(program);`