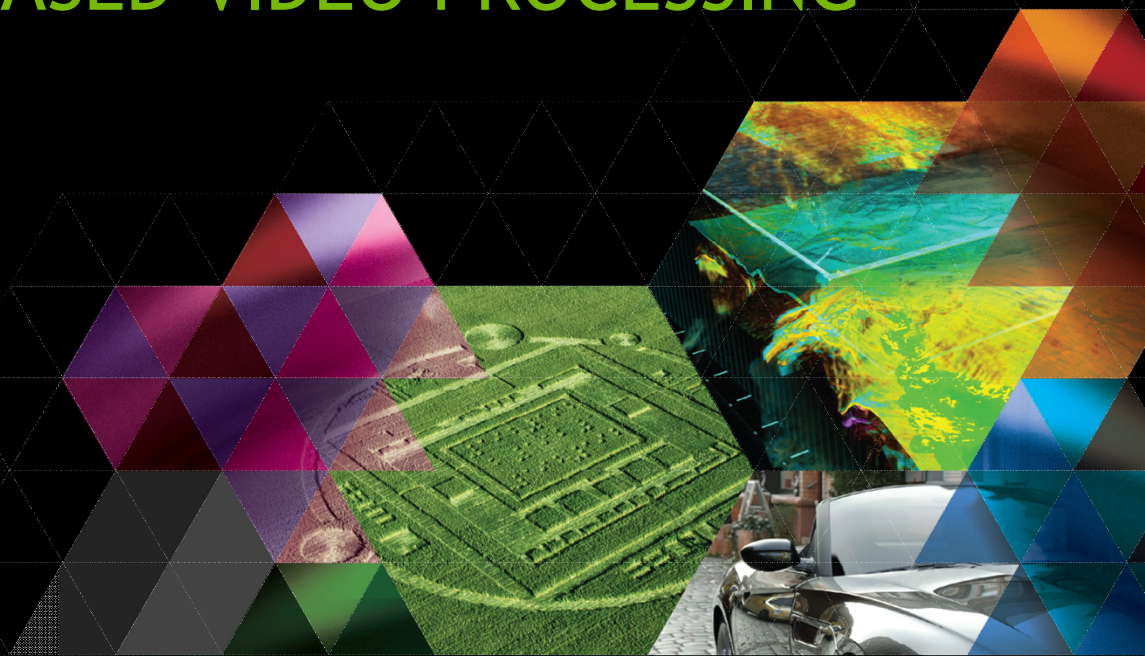
GPU TECHNOLOGY CONFERENCE

NVIDIA

TOPICS IN GPU-BASED VIDEO PROCESSING

Thomas J. True

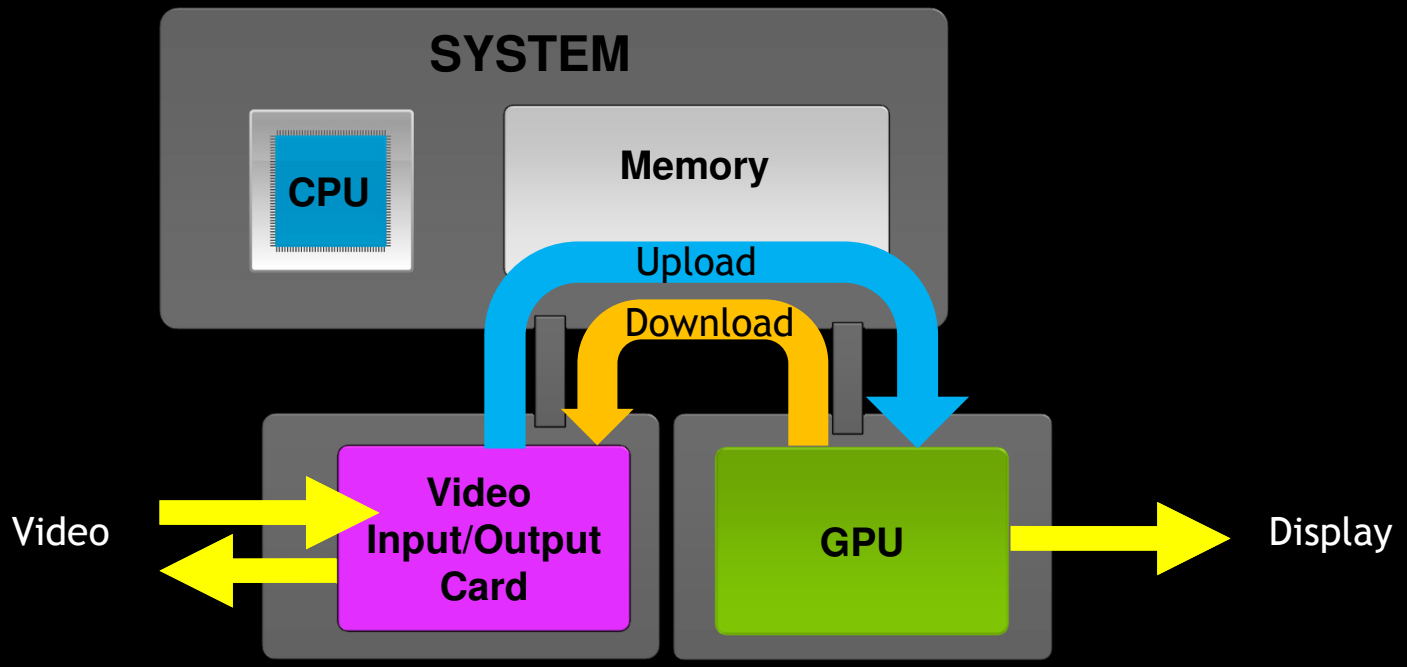
ttrue@nvidia.com



AGENDA

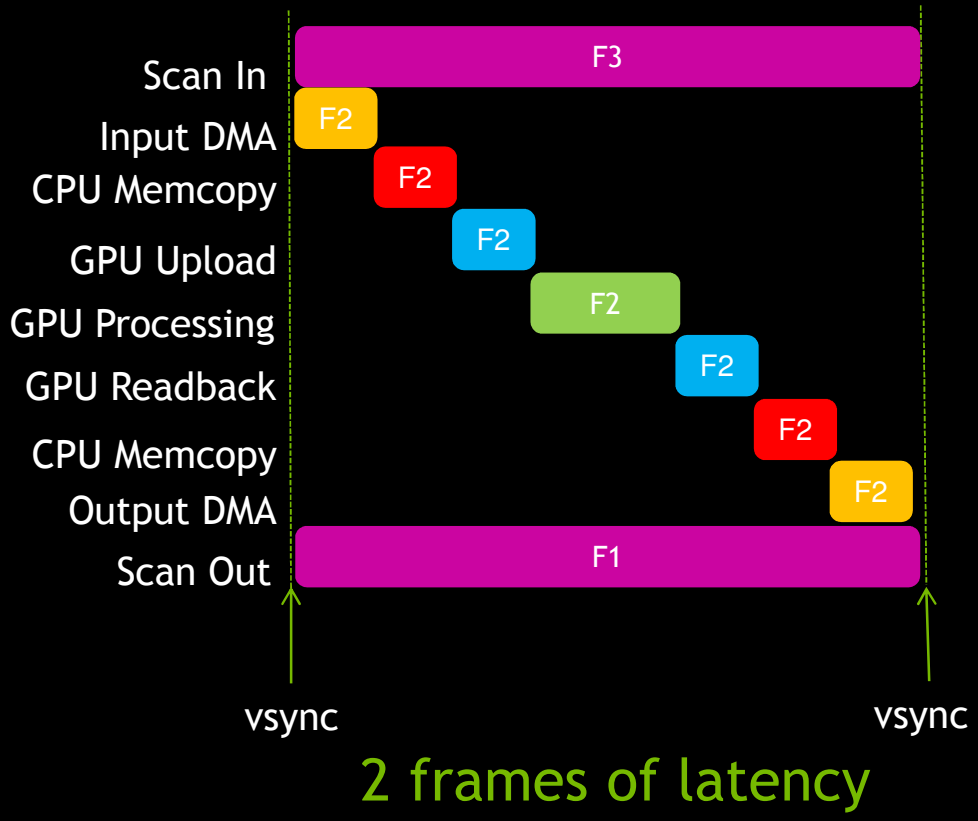
1. *GPU-Based Video Processing Pipeline*
2. *Optimized Data Transfers – Overlap Copy and Compute*
3. *Minimizing the Impact of Compute/Graphics Interop*
4. *OpenGL Compute*
5. *Compute Kernel Design and Optimization*
6. *High Bit Depth Considerations*
7. *High Resolution Display Considerations*

GPU-BASED VIDEO PROCESSING PIPELINE



Video I/O With GPU Processing

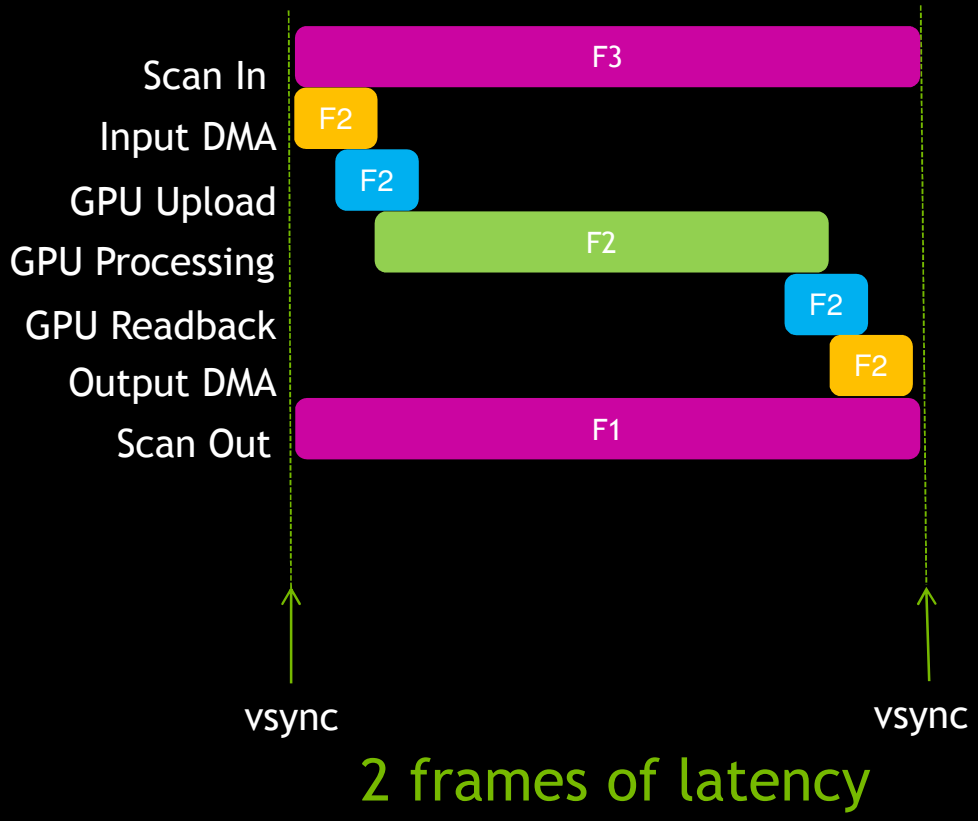
INEFFICIENT GPU-BASED VIDEO PROCESSING



2 frames of latency

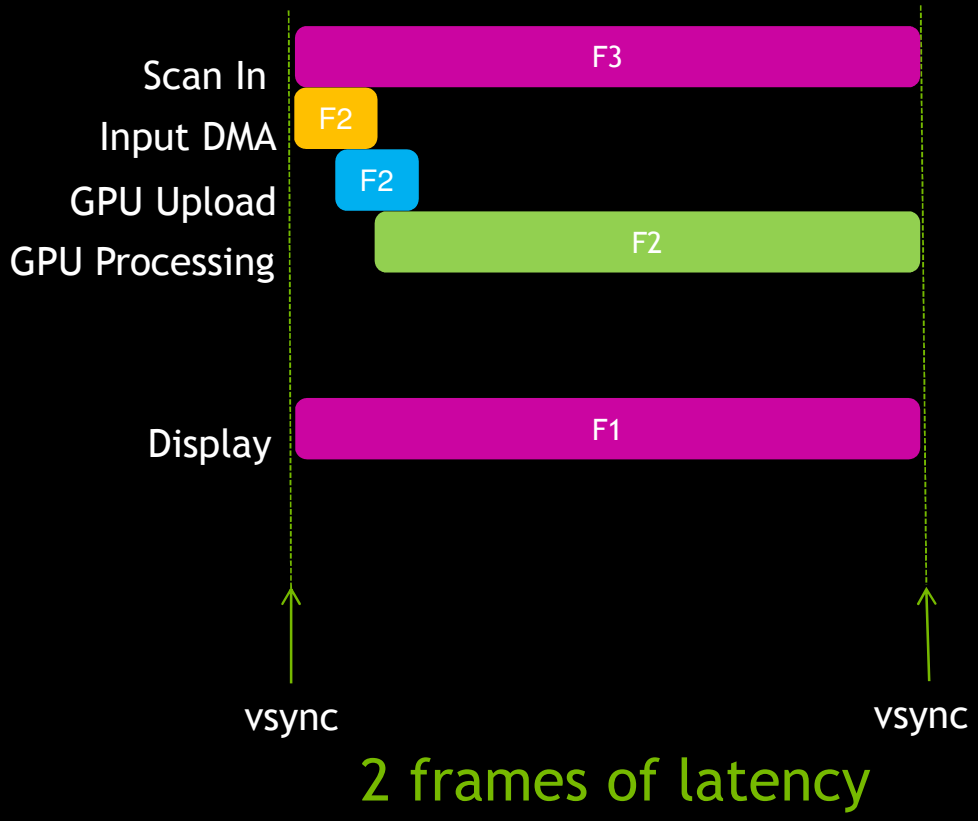
- I/O <-> GPU through GPU unshareable system memory buffer
- GPU synchronous transfers

OPTIMAL GPU-BASED VIDEO PROCESSING



- Pinned system memory
- Overlap I/O DMA transfers with GPU transfers and compute
- GPU Asynchronous
- Partial frame support

OPTIMAL GPU-BASED VIDEO PROCESSING



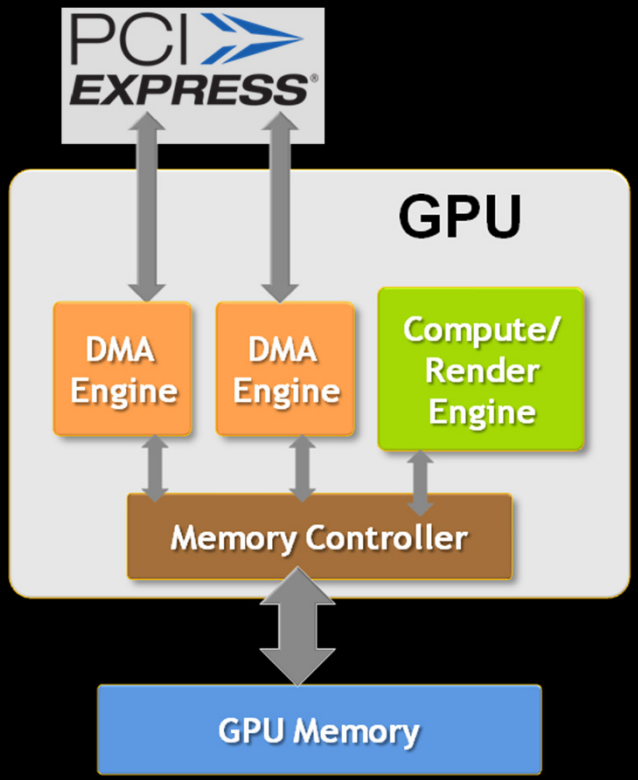
- Pinned system memory
- Overlap I/O DMA transfers with GPU transfers and compute
- GPU Asynchronous
- Partial frame support

OPTIMAL DATA MOVEMENT

- Minimize frame time consumed by GPU upload and download
- Overlap data transfers with compute / rendering

DUAL COPY ENGINES

- Allows copy-to-device + compute + copy-to-host to overlap simultaneously
- Compute/CUDA/OpenCL
  - Using async API's for memcpy with CUDA streams
  - Memory is allocated as page-locked
- Graphics/OpenGL
  - Using PBO's in multiple threads



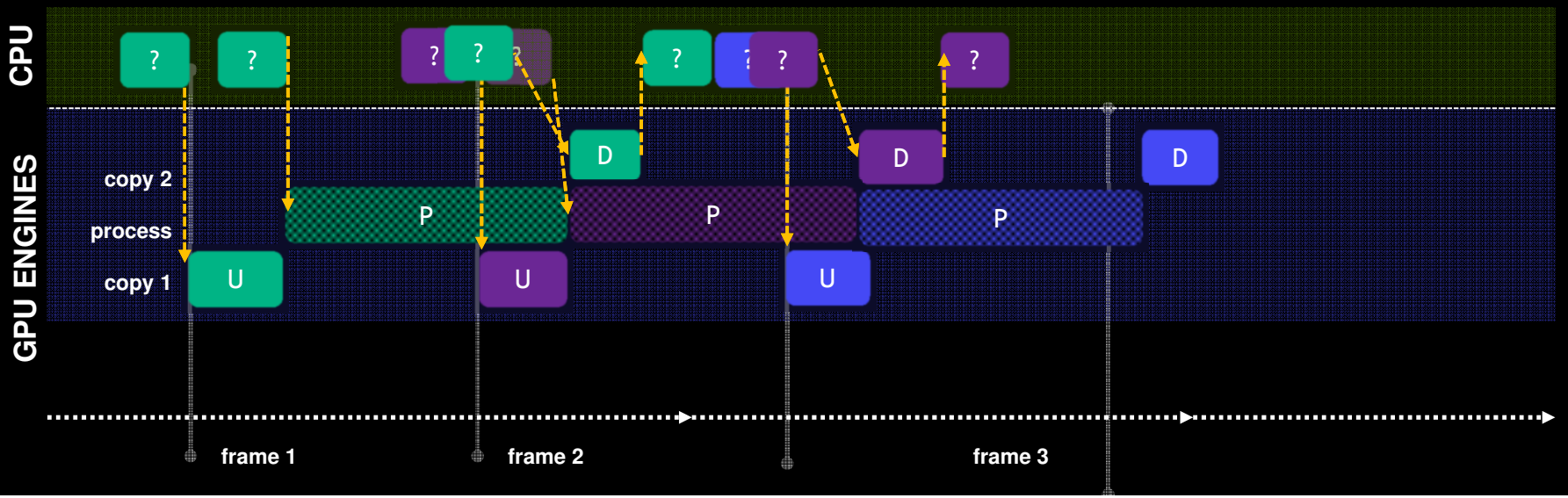
PINNED HOST MEMORY ALLOCATION IN CUDA

```
void *pmem
cudaHostAlloc(pmem, size, cudaHostAllocDefault);
```

```
void *pmem
pmem = malloc(size);

cudaHostRegister(pmem, size, flags);
```

GOAL: OVERLAP I/O WITH PROCESSING

- How can we design the application so that GPU processing can last almost the entire frame time and we don't drop frames?



VIDEO CAPTURE, PROCESS AND PLAY - CUDA

```
while (!done){
    // Start download of processed frame
    cudaMemcpyAsync(pCPUbuf[(bufNum+1)%3],pGPUbuf[bufNum],
                    size, cudaMemcpyDeviceToHost, downloadStream);

    // Start upload of captured frame
    cudaMemcpyAsync(pGPUbuf[bufNum+2],pCPUbuf[(bufNum+2)%3],
                                    size, cudaMemcpyHostToDevice, uploadStream);


    // GPU-based video processing
    myKernel1<<<gridSz,BlockSz,0,computeStream>>>(pGPUbuf[(bufNum+1)%3]…);
    myKernel2<<<gridSz,BlockSz,0,computeStream>>>(pGPUbuf[(bufNum+1)%3]…);

    // Execute other asynchronous GPU stuff

    // Capture next video frame
    GrabMyFrame(pCPUbuf[bufNum]);
    // Do other CPU stuff
    cudaThreadSynchronize();
    bufNum++; bufNum %=3;
}
```

OVERLAP CAPTURE, PROCESS AND PLAY - OPENGL

Capture Thread

```
// Wait for signal to start upload
CPUWait(startUploadValid[2]);
glWaitSync(startUpload[2]);

// Bind texture object
BindTexture(capTex[2]);

// Upload
glTexSubImage(texID…);

// Signal upload complete
GLSync endUpload[2]= glFenceSync(…);
Signal(endUploadValid[2]);
```

[0] [1] [2] [3]

Render Thread

```
// Wait for download to complete
CPUWait(endDownloadValid[3]);
glWaitSync(endDownload[3]);

// Wait for upload to complete
CPUWait(endUploadValid[0];
glWaitSync(endUpload)[0]);

// Bind render target
glFramebufferTexture(playTex[3]);

// Bind video capture  source texture
BindTexture(capTex[0]);

// Draw

// Signal next upload
startUpload[0] = glFenceSync(…);
Signal(startUploadValid[0]);
// Signal next download
startDownload[3] = glFenceSync(…);
Signal(startDownloadValid[3]);
```

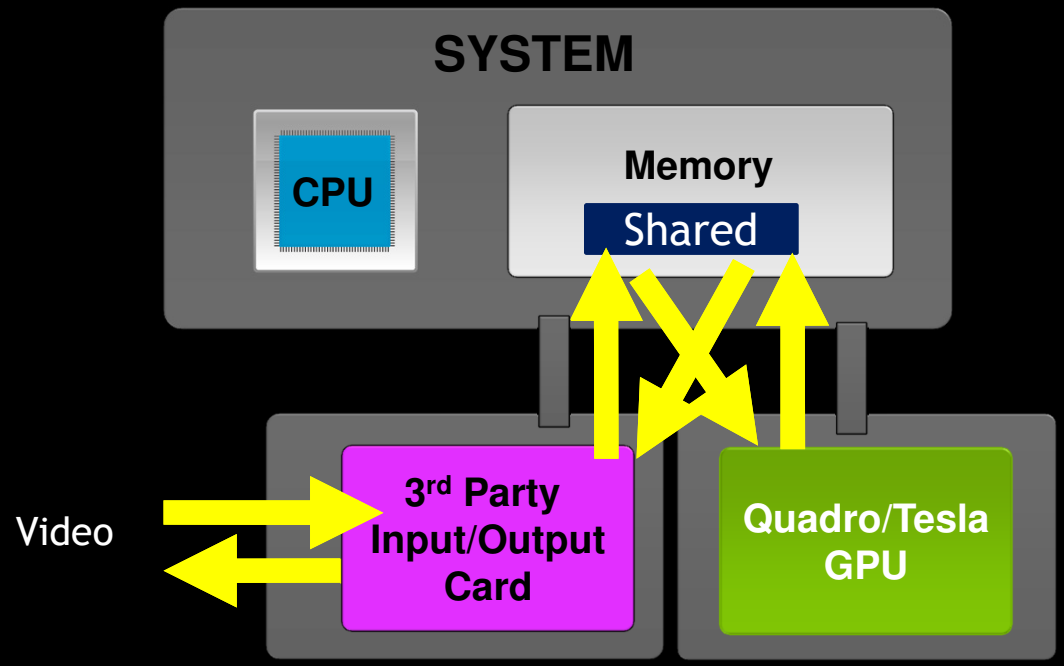
[0] [1] [2] [3]

Playout Thread

```
// Playout thread
CPUWait(startDownloadValid[2]);
glWaitSync(startDownload[2]);

// Readback
glGetTexImage(playTex[2]);

// Read pixels to PBO

// Signal download complete
endDownload[2] = glFenceSync(…);
Signal(endDownloadValid[2]);
```

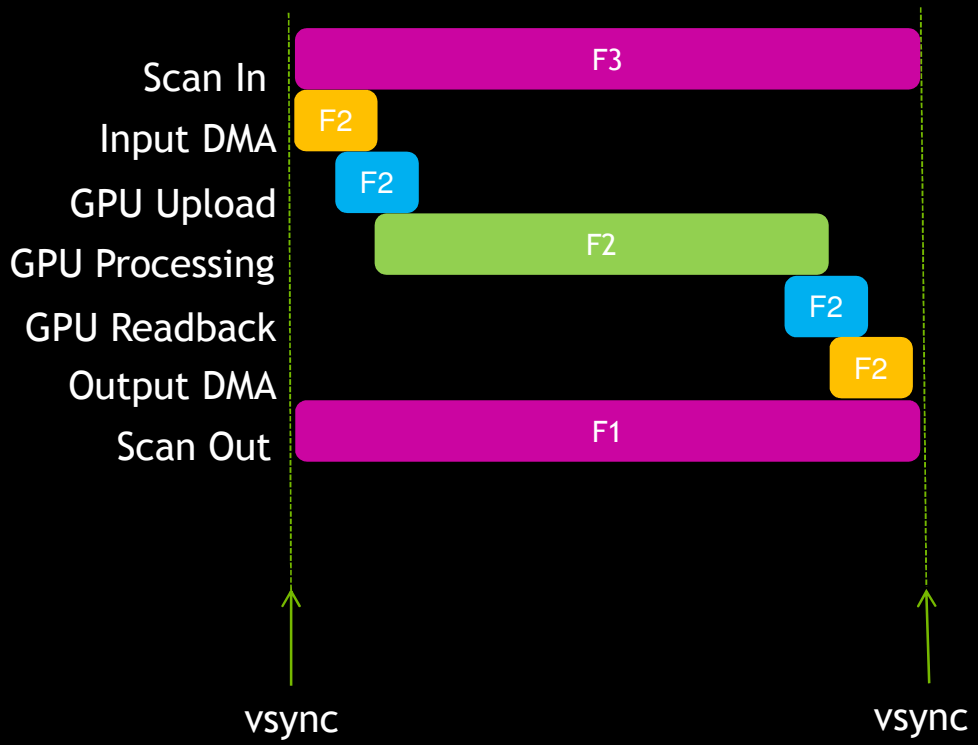
http://www.nvidia.com/docs/IO/40049/Dual_copy_engines.pdf

GPU DIRECT FOR VIDEO



Video Transfers through a Shareable System Memory Buffer!

OVERLAP COPY AND COMPUTE - GPUDIRECT FOR VIDEO

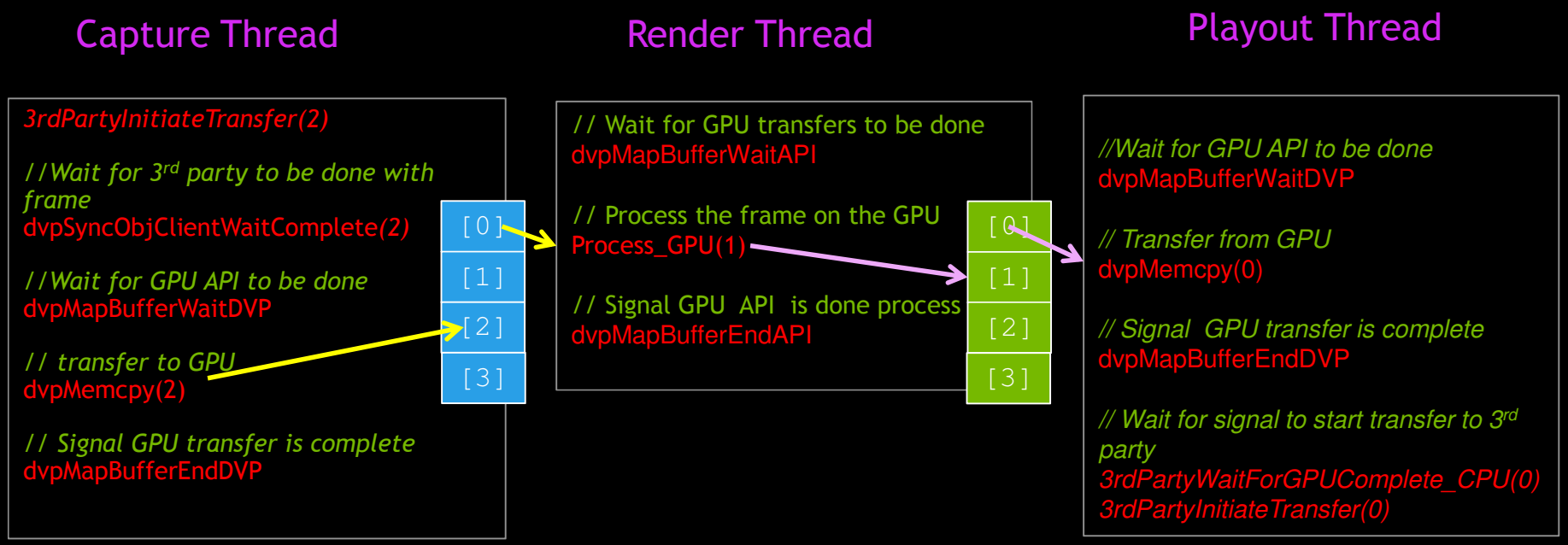


- Unified data transfer API for all Graphics and Compute APIs' objects
  - Video oriented
  - Efficient synchronization
  - GPU shareable system memory
  - Sub-field transfers
  - GPU Asynchronous transfers

Requires 3rd Party Video I/O SDK support GPUDirect for Video

GPU DIRECT FOR VIDEO APP STRUCTURE

- This structure can be used for any GPU API of your choice: CUDA, OpenGL, DirectX



GPU DIRECT FOR VIDEO

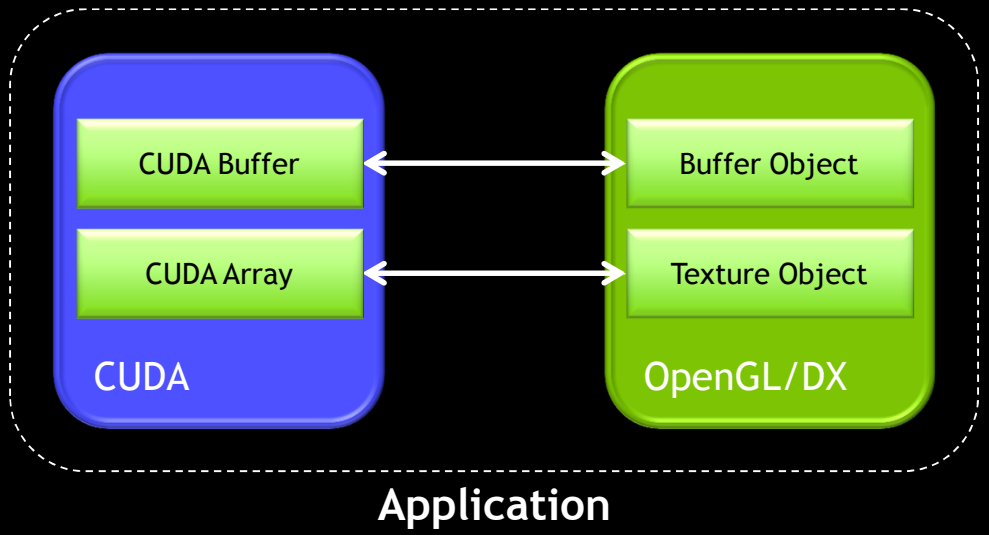
Parallel NSight demonstrates the overlap of upload, render and download.

WHY NOT PEER-TO-PEER

- Same performance and latency for video size frames
- Supportability for ISVs, 3rd party I/O vendors and NVIDIA
  - The upgrade from Fermi to Kepler GPUs required NO code changes to 3rd party I/O SDKs
- IOH-to-IOH communication issues
  - Limited PCIE slot options due to lane allocations
- Support Graphics APIs as well as CUDA
- Multi-GPU Support with R313 display driver
- Multi-OS support, Windows and Linux

COMPUTE/GRAPHICS INTEROP

- Set up the objects in the graphics context.
- Register objects with the compute context.
- Map / Unmap the objects from the compute context.



CUDA GL INTEROP SAMPLE

- Setup and register OpenGL buffer objects

```
GLuint texId;
cudaGraphicsResource_t texRes;

// OpenGL texture creation...
glGenTextures(1, &texId);
glBindTexture(GL_TEXTURE_2D, texId);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8UI_EXT, texWidth, texHeight, 0,
             GL_RGBA_INTEGER_EXT, GL_UNSIGNED_BYTE, 0);
glBindTexture(GL_TEXTURE_2D, 0);

// Register texture object with CUDA for interop
cudaGraphicsGLRegisterImage(&texRes, texId, GL_TEXTURE_2D,
                            cudaGraphicsRegisterFlagsNone);
```

CUDA GL INTEROP SAMPLE

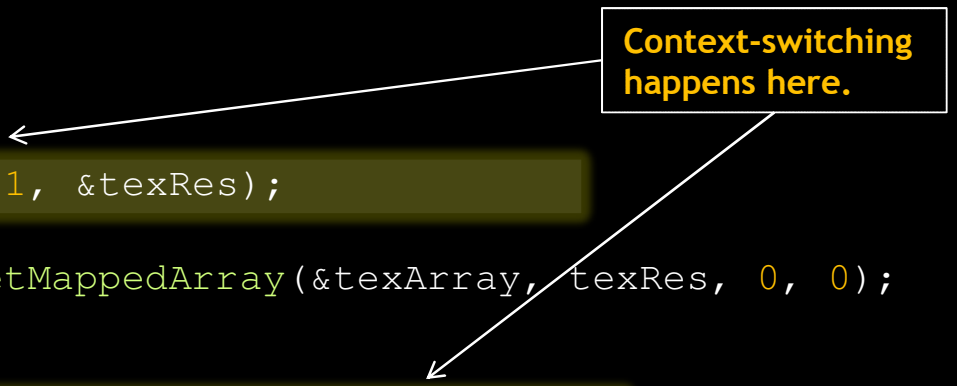
- Mapping between contexts:

```
cudaArray* texArray;

while (!done)
{
    cudaGraphicsMapResources(1, &texRes);

    cudaGraphicsSubResourceGetMappedArray(&texArray, texRes, 0, 0);

    doCUDA(texArray);

    cudaGraphicsUnmapResources(1, &texRes);

    doGL(texId);
}
```

CUDA GL INTEROP SAMPLE

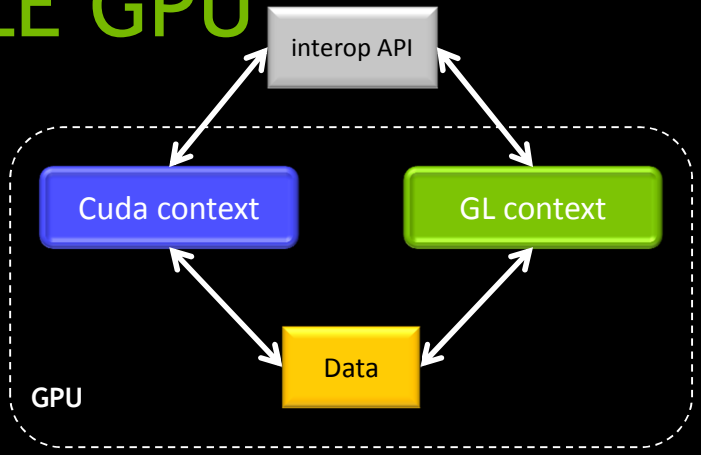
- Sync points:

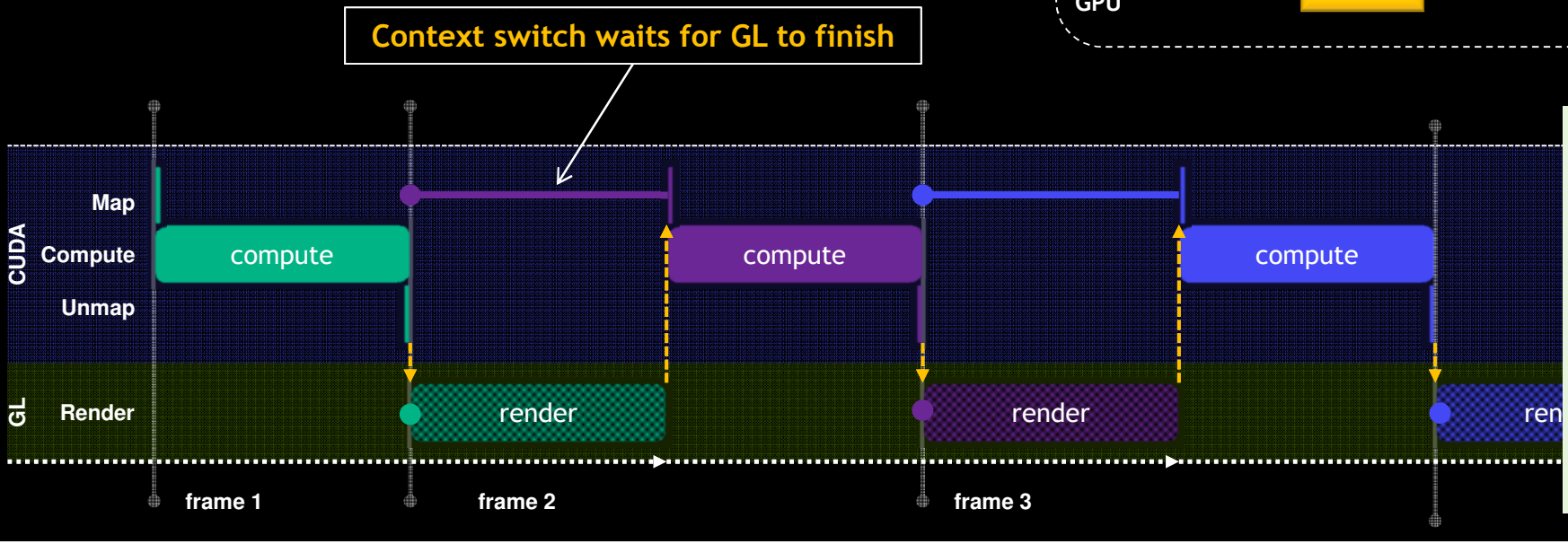
```
cudaArray* texArray;

while (!done)
{
    cudaGraphicsMapResources(1, &texRes);

    cudaGraphicsSubResourceGetMappedArray(&texArray, texRes, 0, 0);
    doCUDA(texArray);

    cudaGraphicsUnmapResources(1, &texRes);

    doGL(texId);
}
```



INTEROP BEHAVIOR: SINGLE GPU

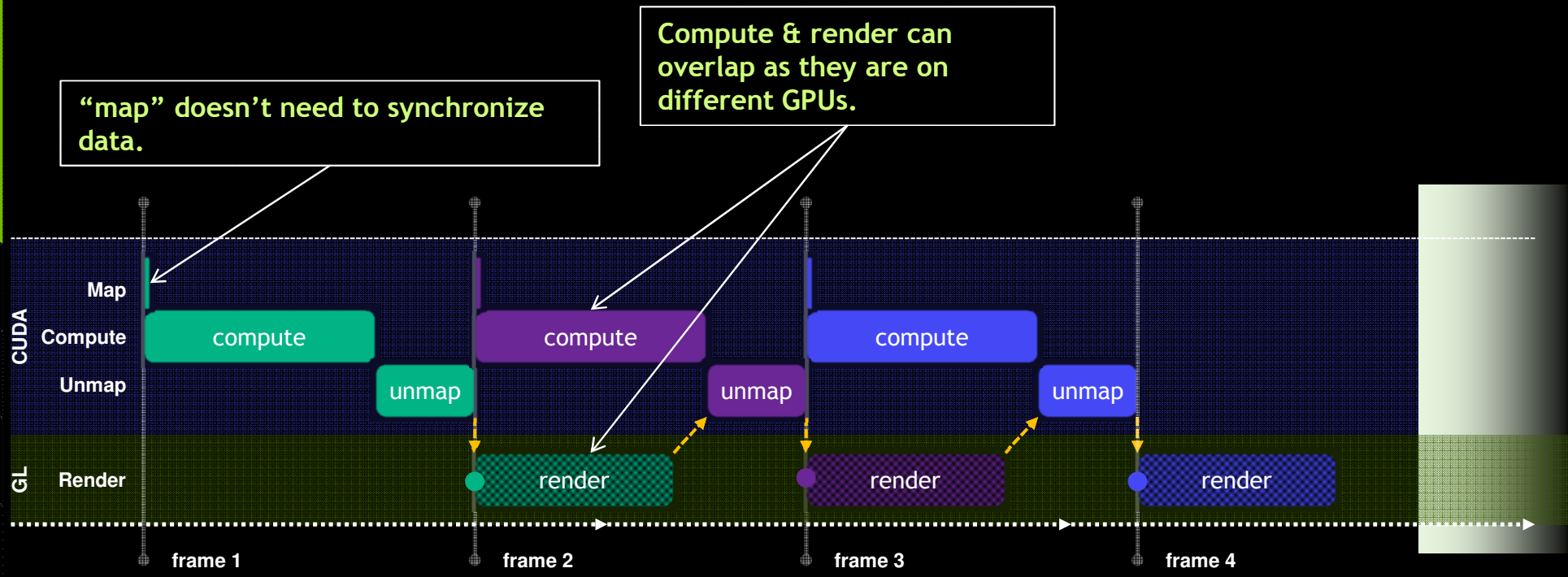
- The resource is shared.
- Context switch is fast and independent on data size.





INTEROP BEHAVIOR: MULTI-GPU

- CUDA-based Image Processing + OpenGL Display...



CODE SAMPLE – WRITE-DISCARD

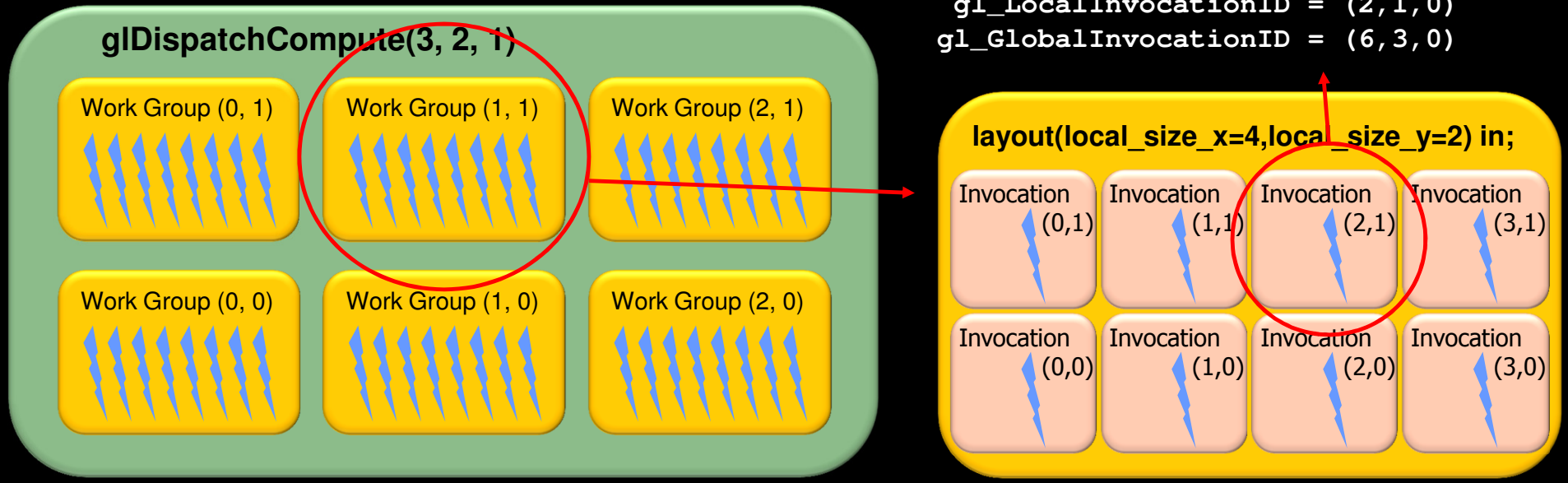
```
cudaArray* texArray;

cudaGraphicsResourceSetMapFlags(texRes, cudaGraphicsMapFlagsWriteDiscard);

while (!done)
{
    cudaGraphicsMapResources(1, &texRes);

    cudaGraphicsSubResourceGetMappedArray(&texArray, texRes, 0, 0);
    doCUDA(texArray);

    cudaGraphicsUnmapResources(1, &texRes);

    doGL(texId);
}
```

Hint that we do not care about the previous contents of buffer.

AVOID INTEROP WITH COMPUTE SHADERS

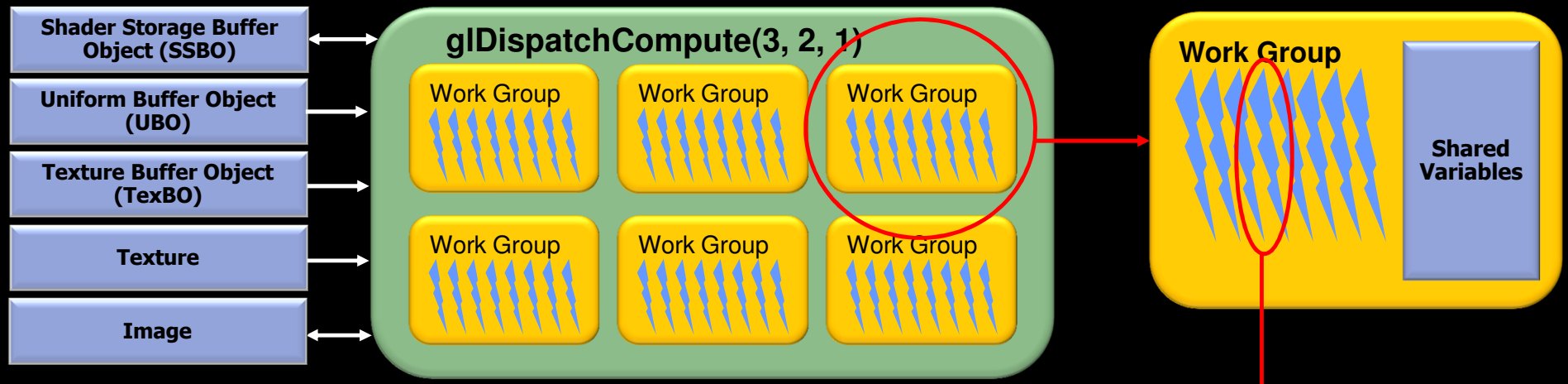
- OpenGL:
  - GL_ARB_compute_shader
  - Consistent GLSL API with familiar GLSL structures, textures, buffers, etc. (Requires OpenGL 4.3)
- DirectX:
  - DX11 DirectCompute
- No context-switching required
- No extra drivers or libraries needed.
- Less overhead on dispatching work
- No device setup code required

COMPUTE PROGRAMMING MODEL




```
   in uvec3 gl_NumWorkGroups; // Number of workgroups dispatched
const uvec3 gl_WorkGroupSize; // Size of each work group for current shader
   in uvec3 gl_WorkGroupID;   // Index of current work group being executed
   in uvec3 gl_LocalInvocationID;  // index of current invocation in a work group
   in uvec3 gl_GlobalInvocationID; // Unique ID across all work groups
```

COMPUTE MEMORY HIERARCHY



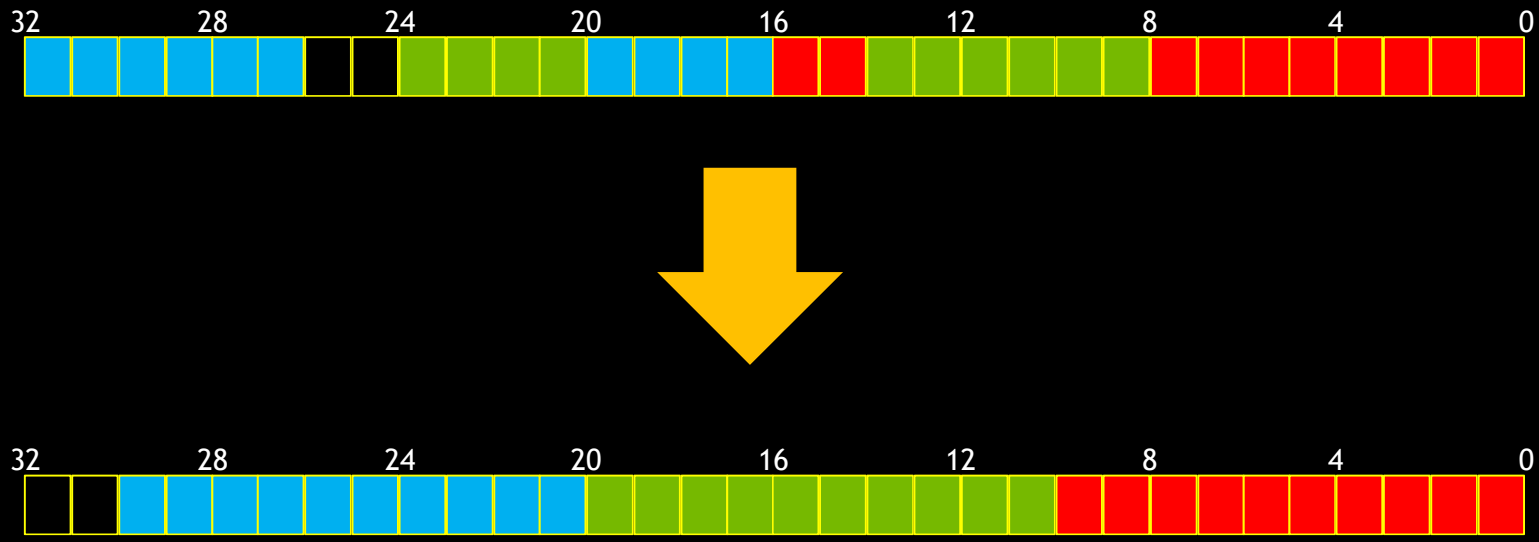
Use `void barrier()` **to synchronize invocations in a work group**

Use memory barriers to order reads/writes accessible to other invocations

```
void memoryBarrier();
void memoryBarrierAtomicCounter();
void memoryBarrierBuffer();
void memoryBarrierImage();
void memoryBarrierShared();    // Only for compute shaders
void groupMemoryBarrier();     // Only for compute shaders
```

GLCOMPUTE EXAMPLE

Problem: 10-Bit Big Endian to Little Endian Conversion



GLCOMPUTE EXAMPLE

```
#version 430

// Needed to use integer textures
#extension GL_EXT_gpu_shader4 : enable
#extension GL_ARB_compute_shader : enable

// Define local size of compute kernel
layout(local_size_x = 24,
       local_size_y = 24,
       local_size_z = 1) in;

// Input = single-component 32-bit unsigned integer texture
layout(r32ui, location = 0) uniform uimage2D inTex;

// Output = four component R10G10B10A2 unsigned integer texture
layout(rgb10_a2ui, location = 1) uniform uimage2D outTex;
```

GLCOMPUTE EXAMPLE

```
void main() {
   ivec2 outPos = ivec2(gl_GlobalInvocationID.xy);
   ivec2 inPos = ivec2(gl_GlobalInvocationID.xy);

   uint texel;
   uvec4 color2;

   // Components are in the upper 30 bits.
   texel = imageLoad(inTex, inPos);

   // Swizzle from Big Endian to Little Endian
   color2.r = ((texel & 0xFC000000)>>26) + ((texel & 0xF0000)>>10);
   color2.g = ((texel & 0x3F00)>>4) + ((texel & 0xF00000)>>20);
   color2.b = ((texel & 0xFF)<<2) + ((texel & 0xC000)>>14);
   color2.a = 0;\n" "   \n"

   // Store 4-component RGB result as uint
   imageStore(outTex, outPos, color2);
};
```

GLCOMPUTE EXAMPLE

```
// Enable the compute shader
glUseProgram(computeProgram);

// Bind input image texture
glBindImageTexture(0, inGpuTexture, 0, GL_FALSE, 0, GL_READ_ONLY, GL_R32UI);

// Bind output image texture
glBindImageTexture(1, scratchGpuTexture, 0, GL_FALSE, 0, GL_WRITE_ONLY, GL_RGB10_A2UI);

// Launch the thread groups
glDispatchCompute(width/24, height/24, 1);
```

OPENGL FRAGMENT SHADER EQUIVALENT

```
// Needed to use integer textures:
#extension GL_EXT_gpu_shader4 : enable
uniform usampler2D tex;

uint colorLookup(vec2 coord) {
//Get Texel
return(texture2D(tex, coord));
}\

void main(out vec4 color2) {
    vec2 coord = vec2(gl_TexCoord[0].s, gl_TexCoord[0].t);

    uint color1 = colorLookup(coord);
    float scale = 1.0f / 1023.0f;
    float scalea = 1.0f / 3.0f;\n"
    color2.r = float(((color1 & 0xFF)<<2) + ((color1 & 0xC000)>>14)) * scale;
    color2.g = float(((color1 & 0x3F00)>>4) + ((color1 & 0xF00000)>>20)) * scale;
    color2.b = float(((color1 & 0xFC000000)>>26)+ ((color1 & 0xF0000)>>10)) * scale;
    color2.a = 0.0f;
}
```

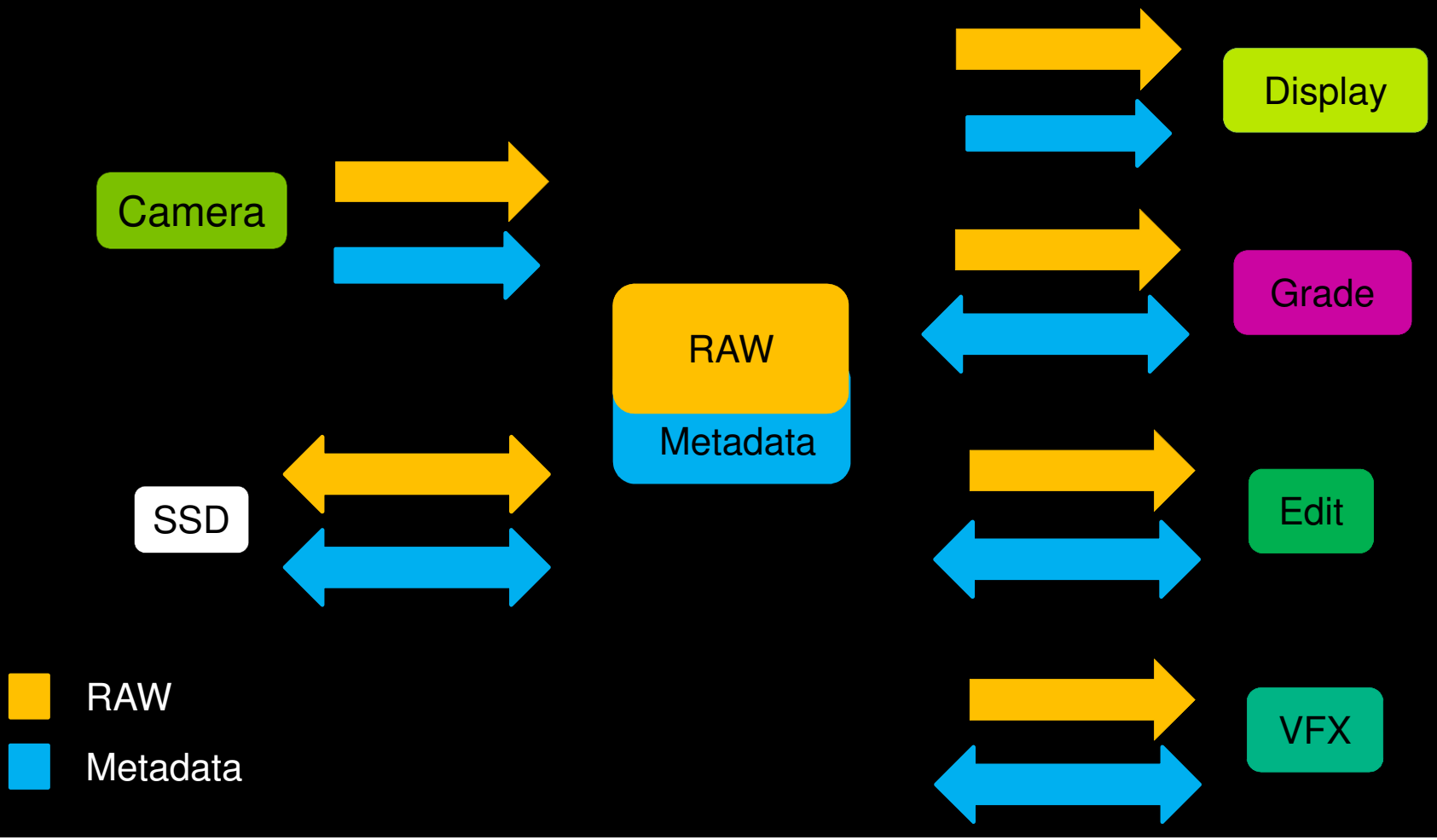
COMPUTE OR FRAGMENT SHADER?

- Fragment Shader
  - Granularity determined by the rasterizer
- Compute Shader
  - Granularity determined by the specified work group size

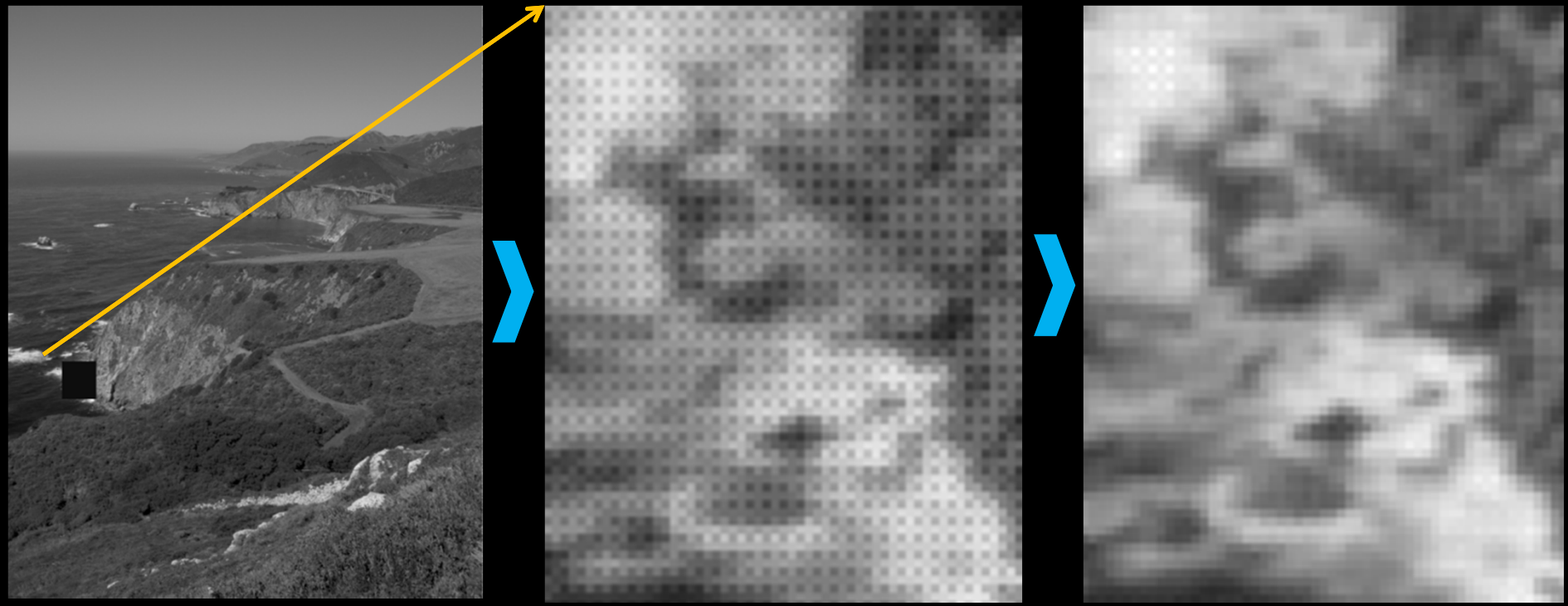
```
glDispatchCompute(GetWidth()/36, GetHeight(), 1);
```

  - Provide more explicit control
  - Each work group will execute on the same compute unit
  - Shared memory

Example: 4K Debayering



Final 4K Image Rendering



Original Image

Raw sensor data

Linearization
Black Level Adjustment

Images Courtesy of Jack Holm

Final 4K Image Rendering

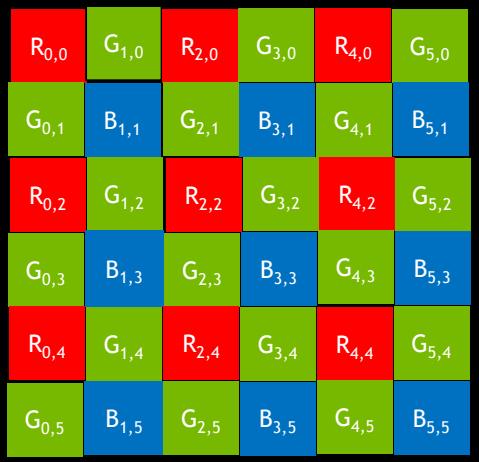
De-Bayer and create raw device RGB image

White Balance and **Color Space Conversion** to working color space

Tone Mapping Gamma Correction

Images Courtesy of Jack Holm

DEBAYER USING BILINEAR INTERPOLATION



Red Pixel

$$R_{2,2} = R_{2,2}$$

$$G_{2,2} = \frac{G_{2,1} + G_{1,2} + G_{3,2} + G_{2,3}}{4}$$

$$B_{2,2} = \frac{B_{1,1} + B_{3,2} + B_{1,3} + B_{3,3}}{4}$$

Blue Pixel

$$R_{1,1} = \frac{R_{0,0} + R_{2,0} + R_{0,2} + R_{2,2}}{4}$$

$$G_{1,1} = \frac{G_{1,0} + G_{0,1} + G_{2,1} + G_{1,2}}{4}$$

$$B_{1,1} = B_{1,1}$$

Green Pixel

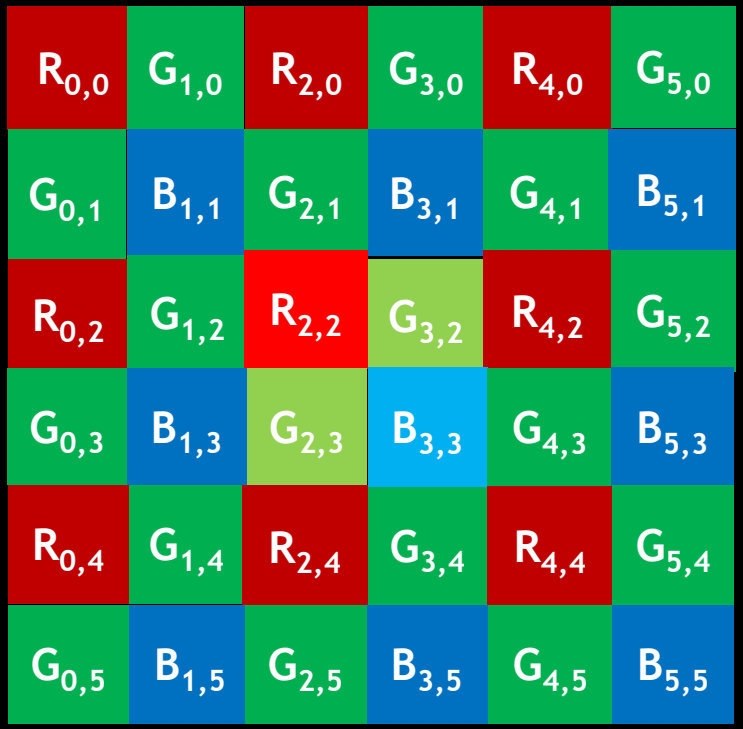
$$R_{2,1} = \frac{R_{2,0} + R_{2,2}}{2}$$

$$G_{2,1} = G_{2,1}$$

$$B_{2,1} = \frac{B_{1,1} + B_{3,1}}{2}$$

DETERMINE UNIT OF WORK TO AVOID THREAD DIVERGENCE

Each thread computes color components for 2x2 quad:

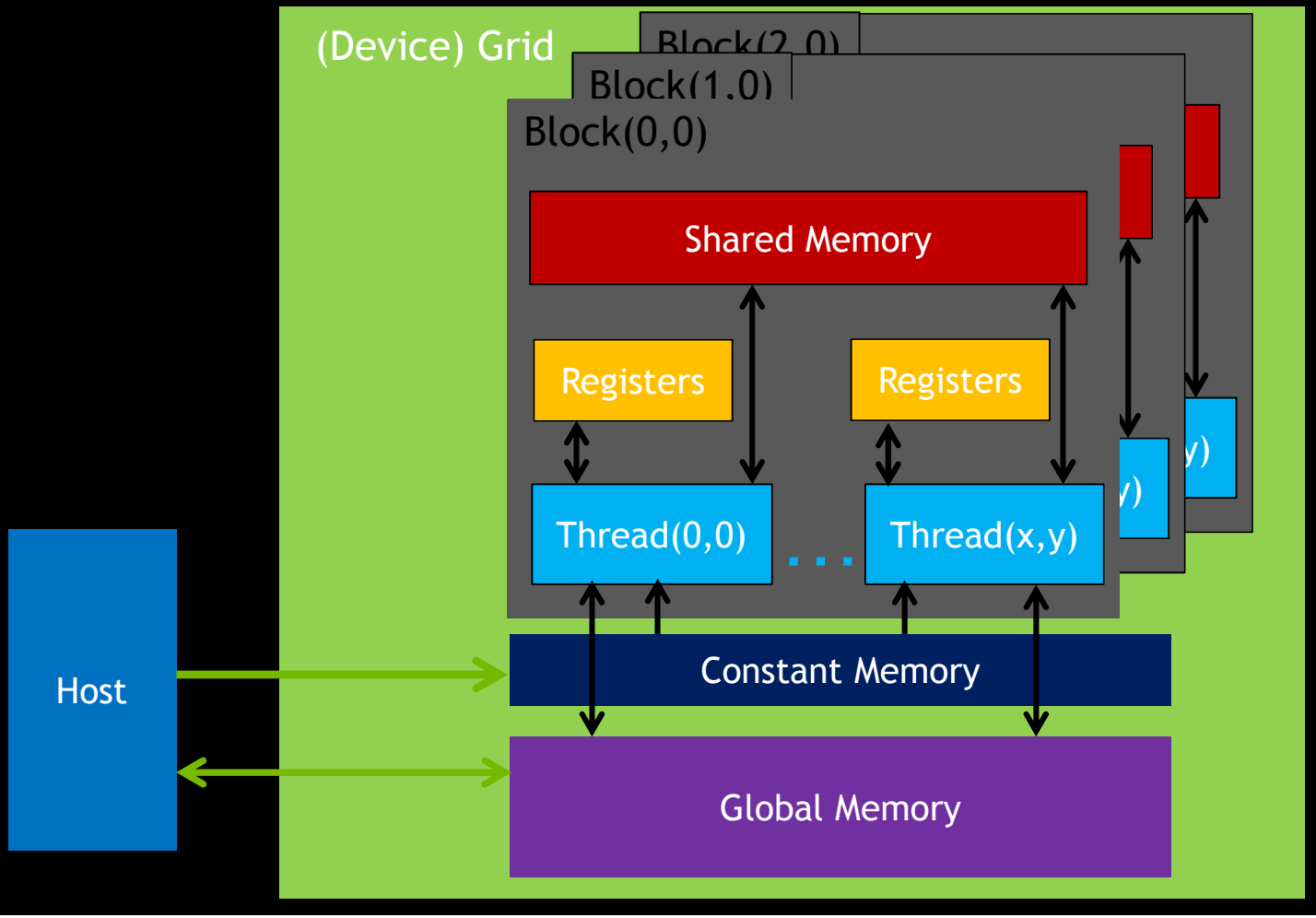


$$R_{2,2} = R_{2,2}$$

$$G_{2,2} = \frac{G_{2,1} + G_{1,2} + G_{3,2} + G_{2,3}}{4}$$

$$B_{2,2} = \frac{B_{1,1} + B_{3,2} + B_{1,3} + B_{3,3}}{4}$$

OPTIMIZE / COALESCE MEMORY ACCESS



TEXTURE MEMORY

- Separate dedicated texture cache
  - Only 1 device memory read on a miss
- Out-of-bounds index handling (clamp or wrap)
- Interpolation (linear, bilear, trilinear or none)
- Format conversion ({char, short, int} -> float)

Free

Memory Coalescing

Use memory access patterns that enable the GPU to coalesce groups of reads or writes of multiple data items into one operation.

TEXTURE MEMORY

```
// Allocate CUDA device memory for the source image on the GPU
unsigned short * pGPU_Source_Image;
size_t SourcePitch;
err = cudaMallocPitch((void**)&pGPU_Source_Image, &SourcePitch,
                      Image_Width*sizeof(unsigned short),
                      Image_Height);

// Copy the source image to the GPU
err = cudaMemcpy2DAsync(pGPU_Source_Image, SourcePitch,
                        pCPU_Source_Image,
                        Image_Width*sizeof(unsigned short),
                        Image_Width*sizeof(unsigned short),
                        Image_Height, cudaMemcpyHostToDevice, 1);
```

TEXTURE MEMORY

HSV Table	HSV mapping table
Exposure Curve	1D array specifying the exposure curve
Tone Map	1D array specifying the tone/look map
Gamma Table	1D array specifying the gamma ramp

CONSTANT MEMORY

- 64KB/device
- Cached
- As fast as reading from a register

```
// Store the color space conversion matrix on the GPU,
// in CUDA constant memory.  Avoid the overhead of passing it
// as an argument as it is unlikely to change per frame.
err = cudaMemcpyToSymbolAsync(ColorMatrix,
                              ColorMatrix_CPU,
                              9*sizeof(float), 0,
                              cudaMemcpyHostToDevice, 1);
```

CONSTANT MEMORY

Coefficient Table	Weight multipliers that represent the contribution of each of the 2 x 2 single color source pixels to each color channel component in the 2 x 2 RGB output pixels.
Offset Table	Precomputed offsets to each of the source pixels within the 2 x 2 source pixel quad.
White Point	Vector containing the white point
Camera CSC	3x3 matrix specifying the camera RGB colorspace conversion
Final CSC	3x3 matrix specifying the final output colorspace conversion

OPTIMIZE CGMA – COMPUTE TO GMEM ACCESS

$$R_{2,2} = R_{2,2} \qquad G_{2,2} = \frac{G_{2,1} + G_{1,2} + G_{3,2} + G_{2,3}}{4} \qquad B_{2,2} = \frac{B_{1,1} + B_{3,2} + B_{1,3} + B_{3,3}}{4}$$

No Shared Memory

3 values calculated for each 9 GMEM reads

.33:1

Quadro K5000: 173 GB/sec GMEM Bandwdith

173 * 0.33 = 57 GB/sec = 19 gigapixels

Shared Memory

Block Size: 32 x 21
SMEM Size: 34 x 22 (includes apron pixels)
672 values : 748 GMEM reads

.90:1

Quadro K5000: 173 GB/sec GMEM Bandwdith

173 * 0.90 = 156 GB/sec = 51 gigapixels

Use SMEM to improve CGMA!

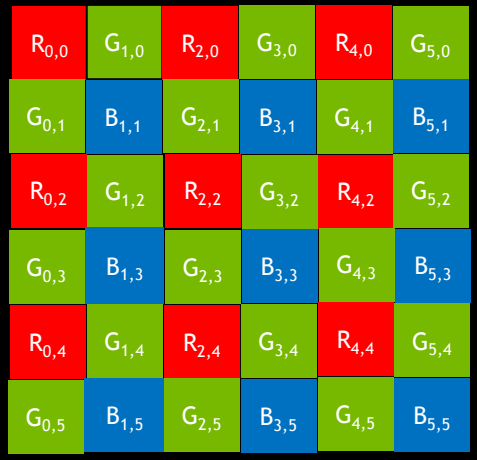
USE SHARED MEMORY

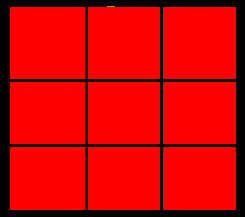
- Faster than global device memory.
- Structure usage to avoid bank conflicts

```
// Shared memory to hold the image tile we will read in from the raw image.
__shared__ float tile_R[BILINEAR_TILE_H][BILINEAR_TILE_W];
__shared__ float tile_G1[BILINEAR_TILE_H][BILINEAR_TILE_W];
__shared__ float tile_B[BILINEAR_TILE_H][BILINEAR_TILE_W];
__shared__ float tile_G2[BILINEAR_TILE_H][BILINEAR_TILE_W];
```

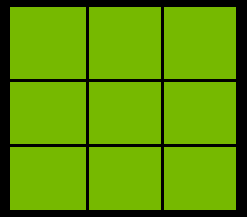
AVOID SMEM BANK CONFLICTS

Use four different color planes to store raw source pixels:

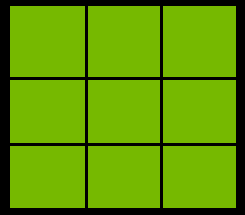




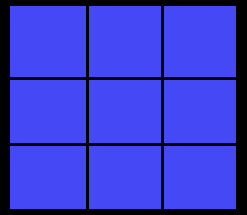
Red SMEM Tile



Green 1 SMEM Tile



Green 2 SMEM Tile



Blue SMEM Tile

MAXIMIZE OCCUPANCY

$$occupancy = \frac{\#\ of\ executing\ threads}{\#of\ possible\ executing\ threads}$$

- Function of:
  - Block size
  - Shared memory
  - Registers

MAXIMIZE OCCUPANCY

CUDA_Occupancy_Calculator (version 1) [Autosaved] [Compatibility Mode]

CUDA GPU Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	3.0	(Help)
1.b) Select Shared Memory Size Config (bytes)	49152	
2.) Enter your resource usage:		
Threads Per Block	672	(Help)
Registers Per Thread	11	
Shared Memory Per Block (bytes)	11968	

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:		(Help)
Active Threads per Multiprocessor	2016	
Active Warps per Multiprocessor	63	
Active Thread Blocks per Multiprocessor	3	
Occupancy of each Multiprocessor	98%	

Physical Limits for GPU Compute Capability:	3.0
Threads per Warp	32
Warps per Multiprocessor	64
Threads per Multiprocessor	2048
Thread Blocks per Multiprocessor	16
Total # of 32-bit registers per Multiprocessor	65536
Register allocation unit size	256
Register allocation granularity	warp
Registers per Thread	63
Shared Memory per Multiprocessor (bytes)	49152
Shared Memory Allocation unit size	256
Warp allocation granularity	4
Maximum Thread Block Size	1024

Allocated Resources		Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warps	(Threads Per Block / Threads Per Warp)	21	64	3
Registers	(Warp limit per SM due to per-warp reg count)	21	128	6
Shared Memory (Bytes)		12032	49152	4

Note: SM is an abbreviation for (Streaming) Multiprocessor

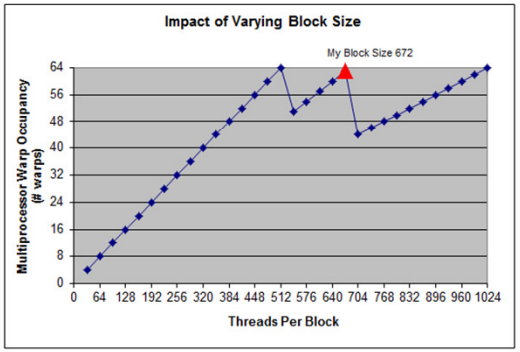
Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	3	21	63
Limited by Registers per Multiprocessor	6		
Limited by Shared Memory per Multiprocessor	4		

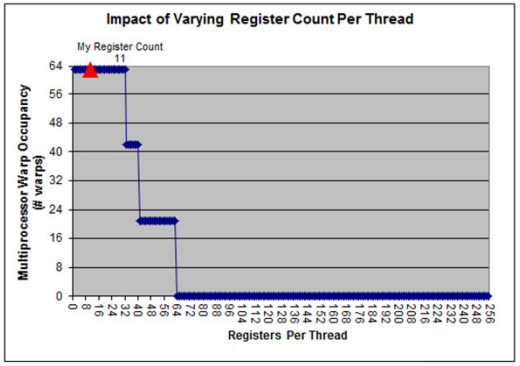
Note: Occupancy limiter is shown in orange

Physical Max Warps/SM = 64
Occupancy = 63 / 64 = 98%

Click Here for detailed instructions on how to use this occupancy calculator.
For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.





Calculator | Help | GPU Data | Copyright & License

KNOW YOUR SYSTEM: DEVICEQUERY.EXE

```
Device 0: "Quadro K5100M"
  CUDA Driver Version / Runtime Version          6.0 / 5.5
  CUDA Capability Major/Minor version number:    3.0
  Total amount of global memory:                 8192 MBytes (8589934592 bytes)
  ( 8) Multiprocessors, (192) CUDA Cores/MP:     1536 CUDA Cores
  GPU Clock rate:                                771 MHz (0.77 GHz)
  Memory Clock rate:                             1800 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 524288 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536),
                                                 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  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
```

C:\NVIDIA\CUDA\CUDA Samples\Bin\win64\Release

KNOW YOUR SYSTEM: DEVICEQUERY.EXE

```
  Concurrent copy and kernel execution:          Yes with 2 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 PCI Bus ID / PCI location ID:            1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 6.0, CUDA Runtime Version = 5.5, NumDevs =
1, Device0 = Quadro K5100M
Result = PASS
```

C:\NVIDIA\CUDA\CUDA Samples\Bin\win64\Release

KNOW YOUR SYSTEM: BANDWIDTHTEST.EXE

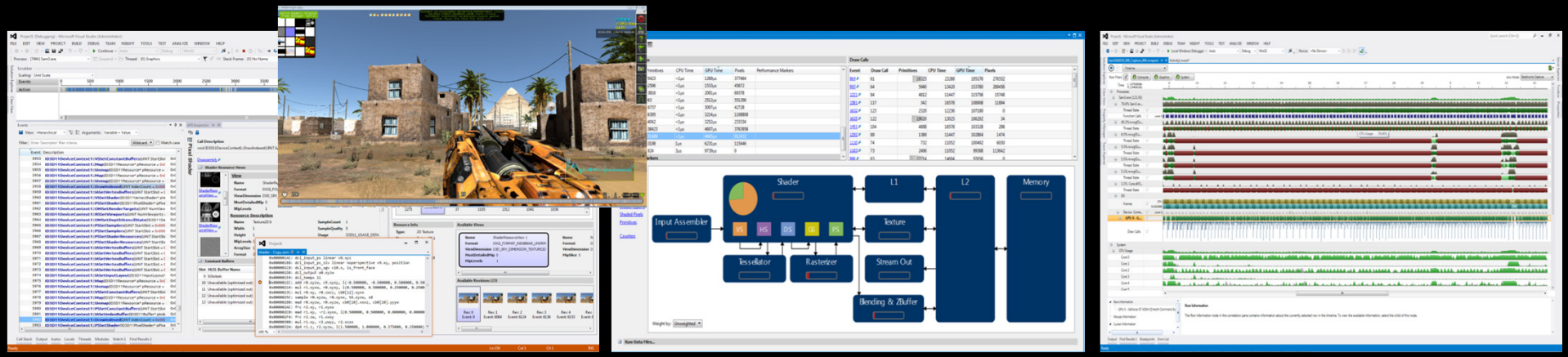
```
Device 0: Quadro K5000
Quick Mode

Host to Device Bandwidth, 1 Device(s)
PINNED Memory Transfers
  Transfer Size (Bytes)        Bandwidth(MB/s)
  33554432                     5937.4

Device to Host Bandwidth, 1 Device(s)
PINNED Memory Transfers
  Transfer Size (Bytes)        Bandwidth(MB/s)
  33554432                     3409.9

Device to Device Bandwidth, 1 Device(s)
PINNED Memory Transfers
  Transfer Size (Bytes)        Bandwidth(MB/s)
  33554432                     129900.5
```

C:\ProgramData\NVIDIA Corporation\CUDA Samples\v5.0\bin\

GPU TECHNOLOGY CONFERENCE

NVIDIA NSIGHT FOR GRAPHICS DEVELOPERS

for OpenGL ~4.3+, Direct3D 9/11/11.1

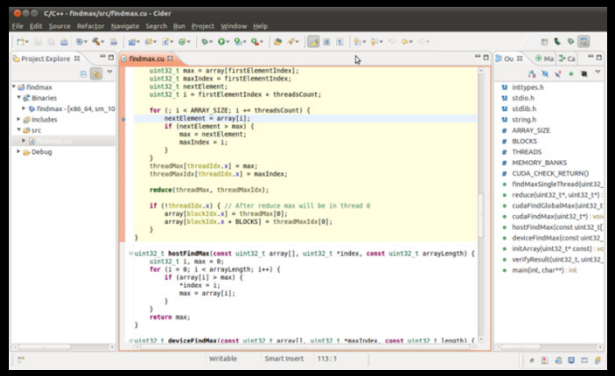


- Frame debugger and profiler
- HLSL and GLSL shader debugger
- Application and system trace

GPU TECHNOLOGY CONFERENCE

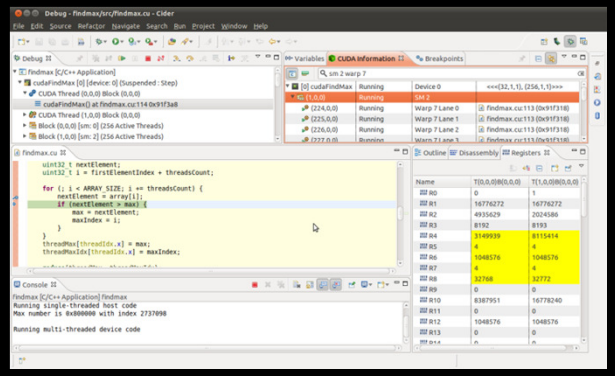
NVIDIA® NSIGHT™ eclipse

Full featured IDE for developing CUDA-C apps on X86, ARM & POWER8.



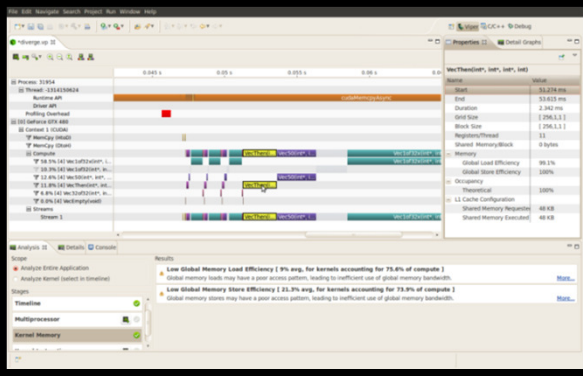
CUDA-Aware Editor

- Automated CPU to GPU code refactoring
- Semantic highlighting of CUDA code
- Integrated code samples & docs
- Cross-compilation for Linux target



Nsight Debugger

- Simultaneously debug CPU and GPU code
- Inspect variables across CUDA threads
- Use breakpoints & single-step debugging
- Integrated CUDA memory checker



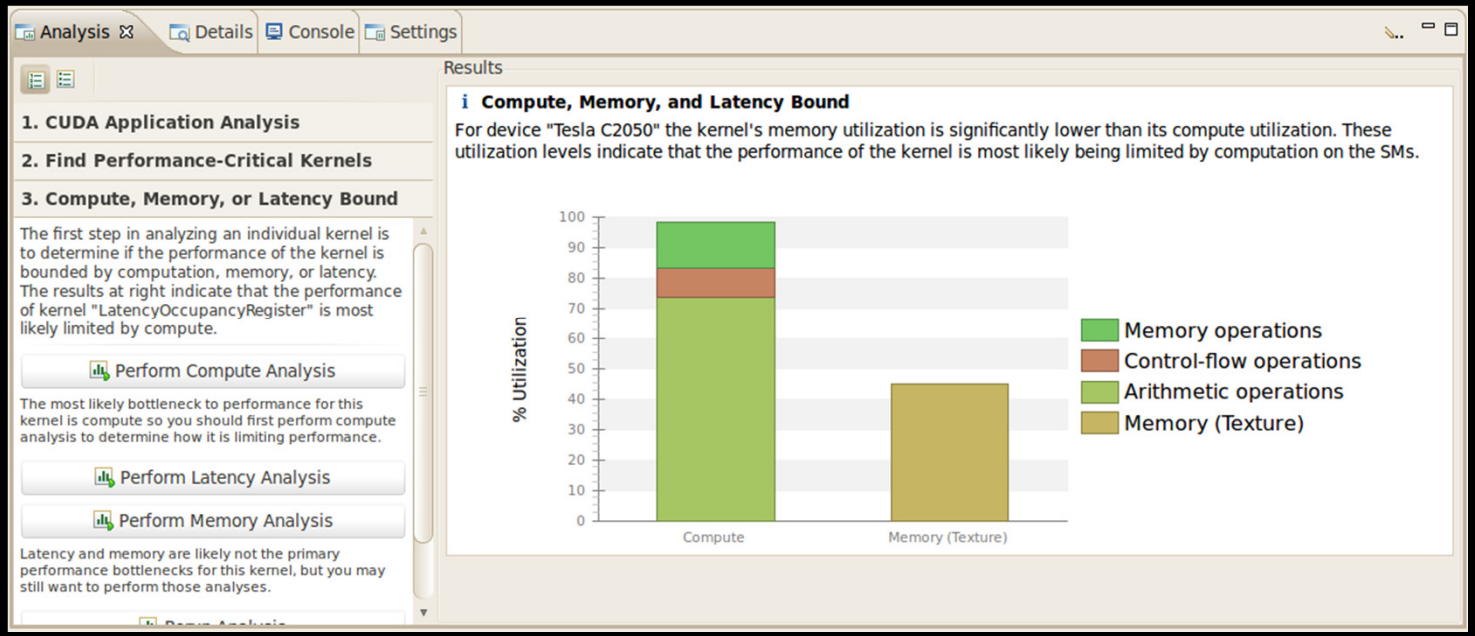
Nsight Profiler

- Quickly identifies performance issues
- Guided expert analysis
- Source line correlation

CUDA STANDALONE TOOLS

Visual Profiler

- Trace CUDA activities
- Performance instrumentation with source code correlation
- Guided Expert Analysis



NVPROF

- Generates execution summary
- Gather Performance events

CUDA-MEMCHECK

- Out of bounds memory access detection
- Detects Race Condition

CUDA-GDB

- Command line CUDA debugging
- Debug CPU and GPU code

* Android new in CUDA 6.0

HIGHER BIT DEPTH CONSIDERATIONS

Bit Depth	API	Scanout Surface	HDMI	DP	Full Screen	Window
10	DirectX	R10G10B10A2_UNORM R10G10B10_XR_BIAS_A2_UNORM	✓	✓	✓	X
10	OpenGL	A2RGB10	✓	✓	✓	✓ +
12	DirectX	R16G16B16A16_FLOAT	✓	✓ *	✓	X
12	OpenGL	X	X	X	X	X

*12bpc supported but not currently enabled due to unavailability of 12bpc DP devices on which to test it.

+Quadro only.

HIGHER RESOLUTION CONSIDERATIONS

- 4K, UltraHD, Cinema 4K -- many names for a display >= 3,840 x 2,160 pixels
  - Ultra HD: 3,840 x 2,160
    - Quad 1080p
  - Cinema 4K: 4,096 x 2,160
    - closer to std 35mm film aspect
- Need to pay attention to the frame-rate!
- Different panels/projectors have different connectors

HIGHER RESOLUTION CONSIDERATIONS

Number of Display Device Inputs:

Single Input	Multiple Inputs
HDMI 1.4 (DVI-DL or DP1.1) 3840x2160@30, 4096x2160@24 DisplayPort 1.2 3,840x2,160@60 4,096x2,160@60 (K6000 only)	2x DP1.1 or DVI-DL 4,096x2,400@60 4x HDMI (3,840x2,400@60) 4x DP1.1 or DVI-DL (3,840x2,400@120)

GOT QUESTIONS? COME HANGOUT WITH US

GPU-Based Image and Video Processing Hangout
GTC Hangouts - Concourse Pod C
Wednesday 9-11AM

GPU TECHNOLOGY CONFERENCE

NVIDIA

QUESTIONS?