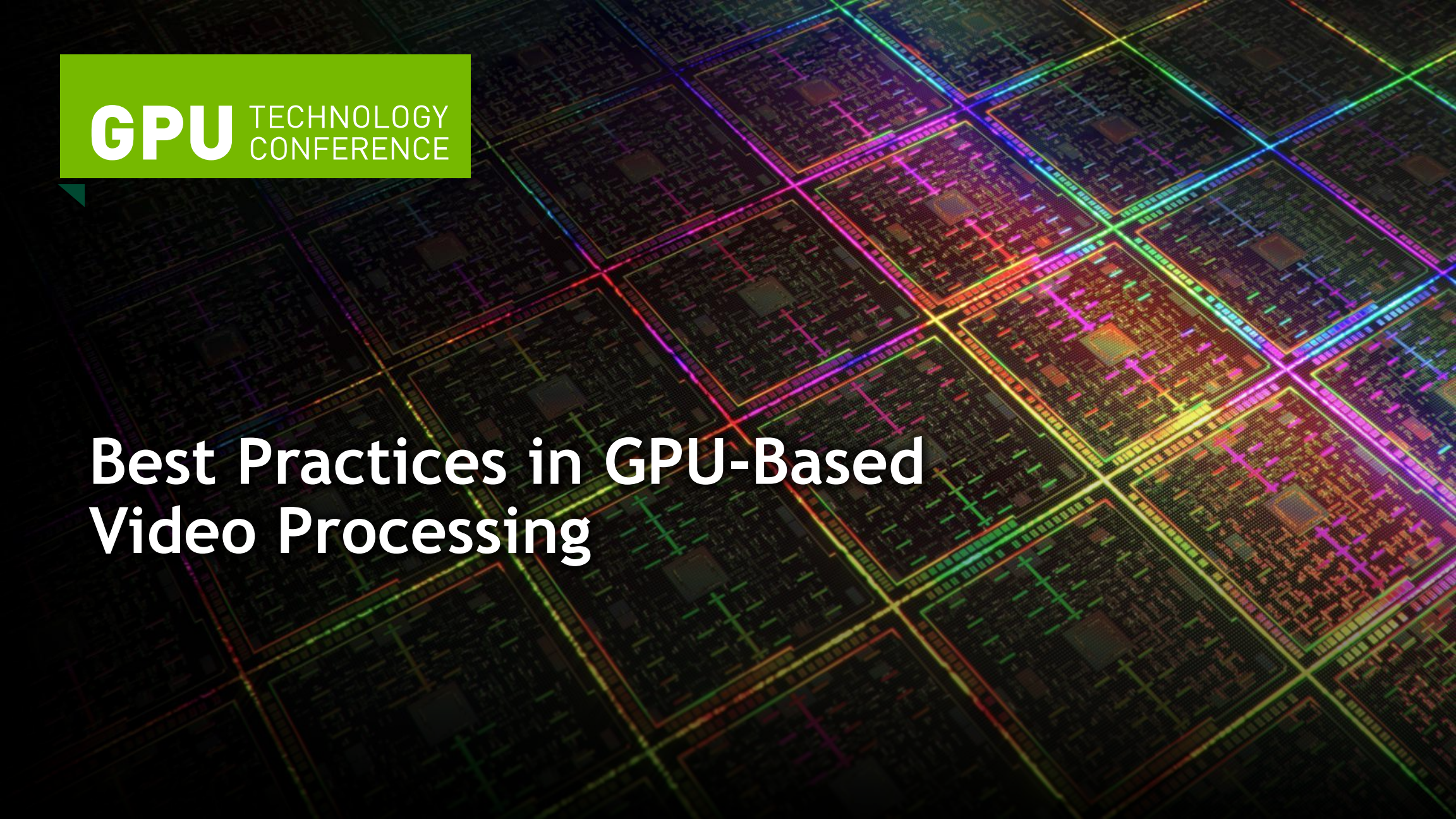
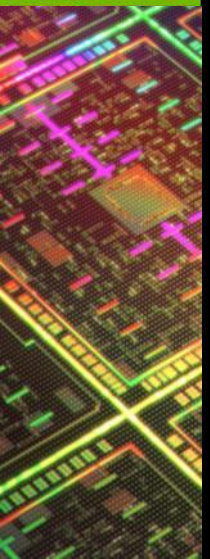


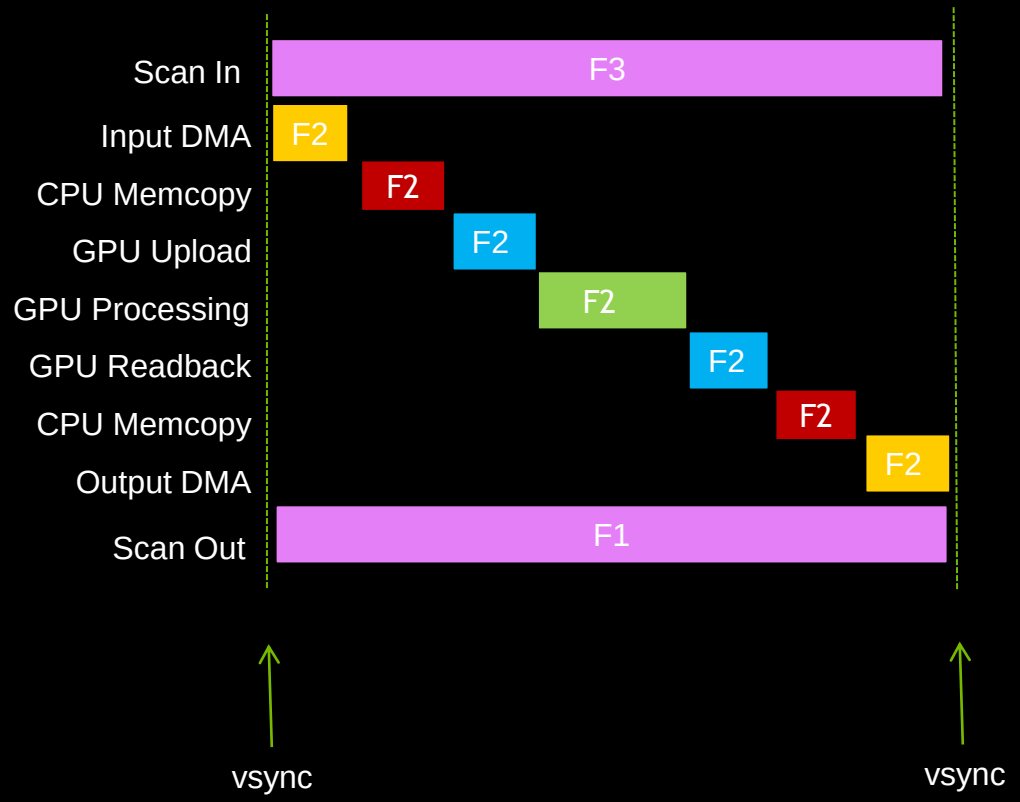
Best Practices in GPU-Based Video Processing



Overview

1. *GPU-Based Video Processing Pipeline*
2. *Host Code Optimization Opportunities*
3. *Kernel Optimizations Opportunities*
4. *Questions*

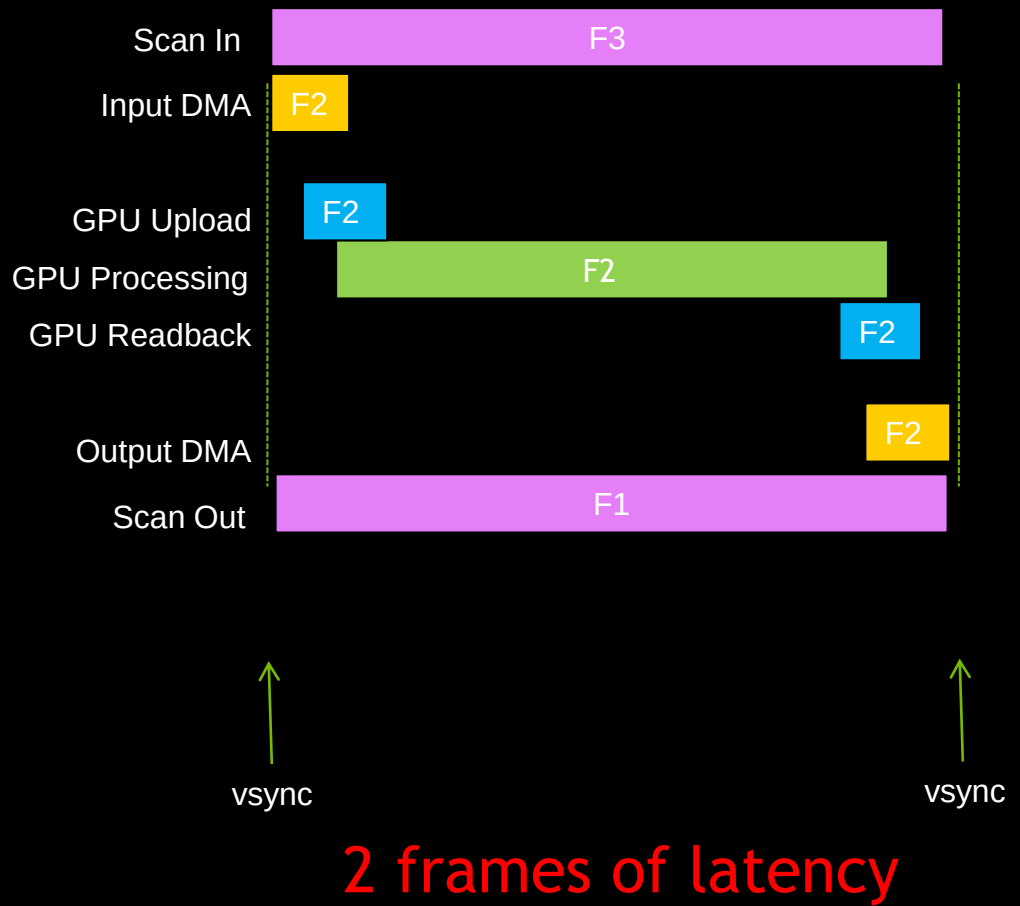
GPU-Based Video Processing Inefficiencies



2 frames of latency

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

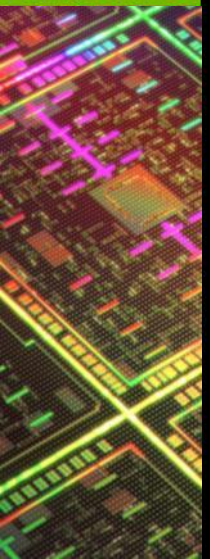
GPU-Based Video Processing Optimal



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

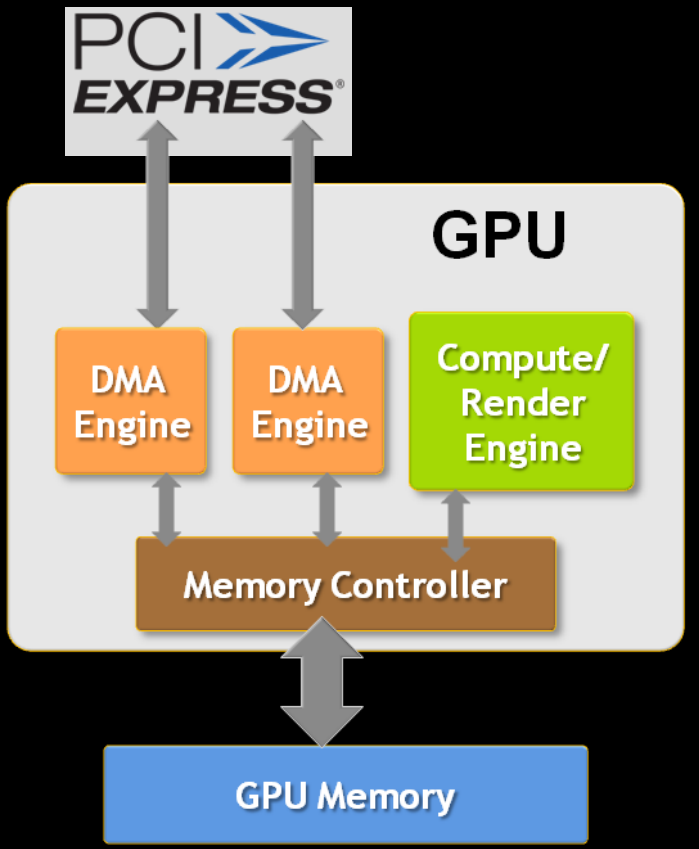
Data Movement

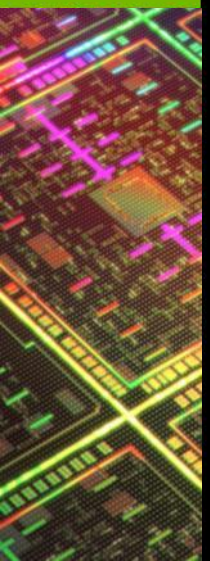
- Minimize frame time consumed by GPU upload and download
- Task overlap: overlap data transfers and data transfers with compute



Dual Copy Engines

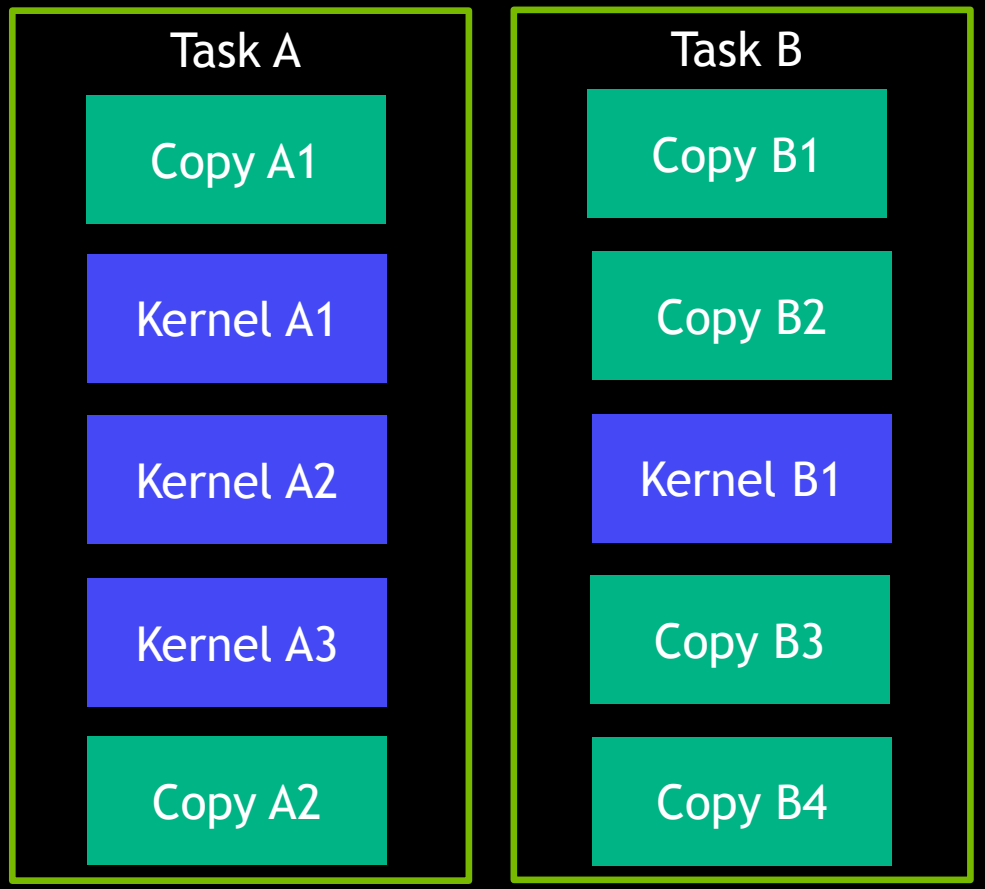
- Fermi has 2 copy engines
- Allows copy-to-host + compute + copy-to-device to overlap simultaneously
- Compute/CUDA
  - Using async API's for memcpy with CUDA streams
  - Memory is allocated as page-locked
- Graphics/OpenGL
  - Using PBO's in multiple threads



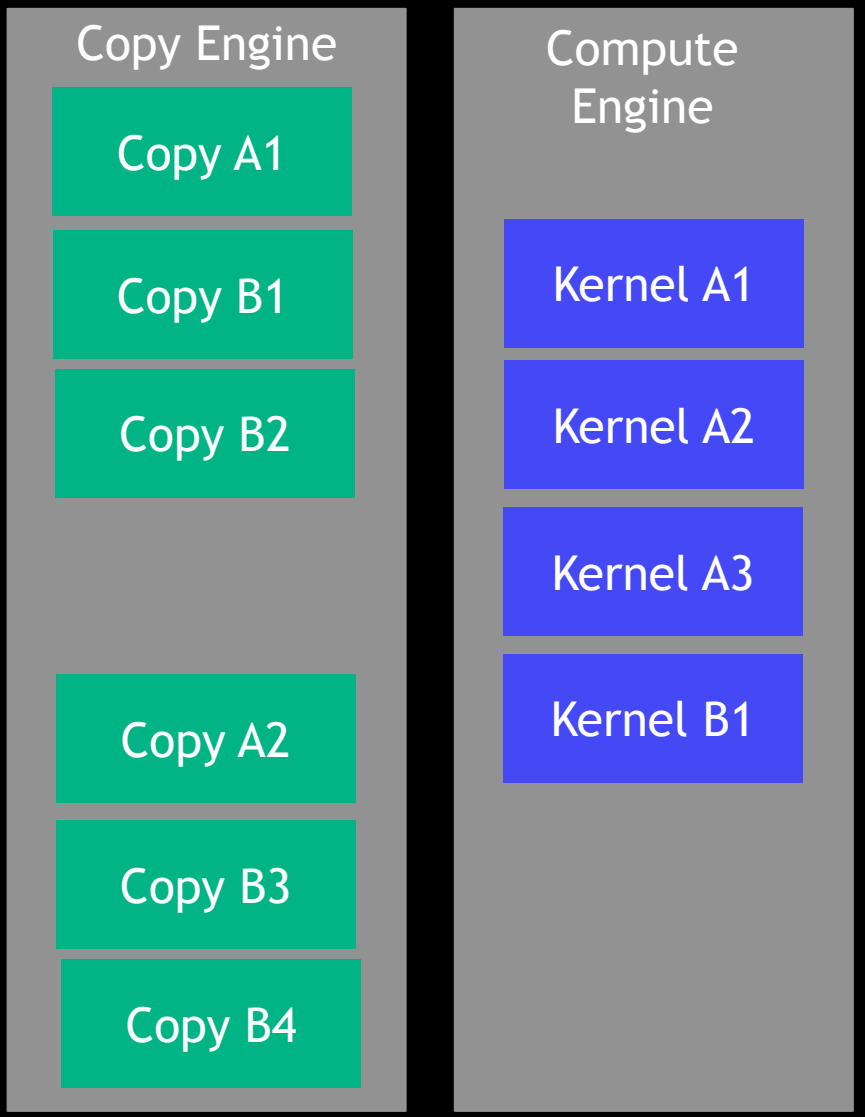


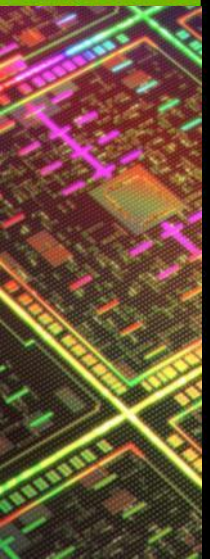
CUDA Streams

Independent Tasks



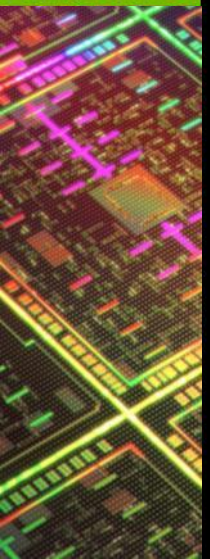
GPU Scheduling





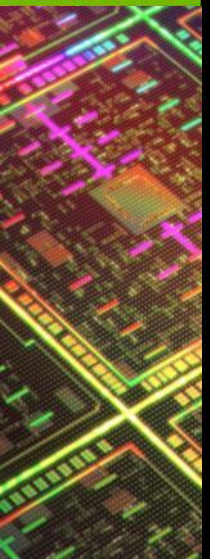
Overlap Copy and Compute - CUDA

```
\\ Allocate pinned host memory
void *pmem
cudaHostAlloc(pmem, size, cudaHostAllocDefault);

\\ Start memcpy to device
cudaMemcpy2DAsync(dst, src, cnt, cudaMemcpyHostToDevice, 1);
myKernel<<<grid, block>>>(…);

\\ Start memcpy from device
cudaMemcpy2DAsync(pmem, src, cnt, cudaMemcpyDeviceToHost, 1);


\\ Perform other CPU tasks here


\\ Wait for all GPU operations to complete
cudaThreadSynchronize();
```



Overlap Copy and Compute - CUDA

```
cudaEvent_t HtoDdone;
\\ Create CUDA event
cudaEventCreate(&HtoDdone,0);
\\ Start host to device memcpy
cudaMemcpyAsync(dst, src, cnt, cudaMemcpyHostToDevice, 1);
\\ Record event
cudaEventRecord(HtoDdone);

myKernel<<<grid,block>>>(…);

\\ Start device to host memcpy
cudaMemcpyAsync(dest,source,bytes,cudaMemcpyDeviceToHost,1);

\\ Perform other CPU tasks here


\\ Wait for host to device memcpy to complete
cudaEventSynchronize(HtoDdone);
\\ Source memory buffer can now be reused.
\\ Wait for all GPU operations to complete
cudaThreadSynchronize();
```

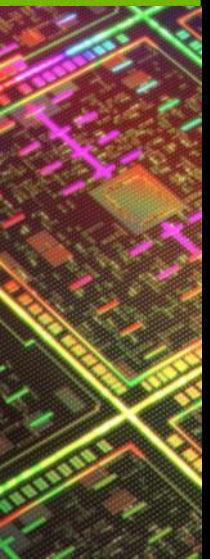


Overlap Video Capture with GPU compute

```
in bufNum = 0;
void * pCPUbuf[2];
\\ Allocate two pinned CPU buffers and ping pong between them.
memAllocHost(…);

while (!done) {

        cudaMemcpyAsync(pGPUbuf,pCPUbuf[(bufNum+1)%2],size,cudaMemcpyHostToDevice,1);

        myKernel<<<…>>>(GPUbuf…);

        \\ Execute other asynchronous GPU stuff

        \\ Grab next frame
        GrabFrame(pCPUbuf[bufNum]);

        \\ Execute other CPU stuff


        \\ Wait for all GPU processing to complete
        cudaThreadSynchronize();


        \\ Swap buffers
        bufNum++; bufNum %=2;
}
```

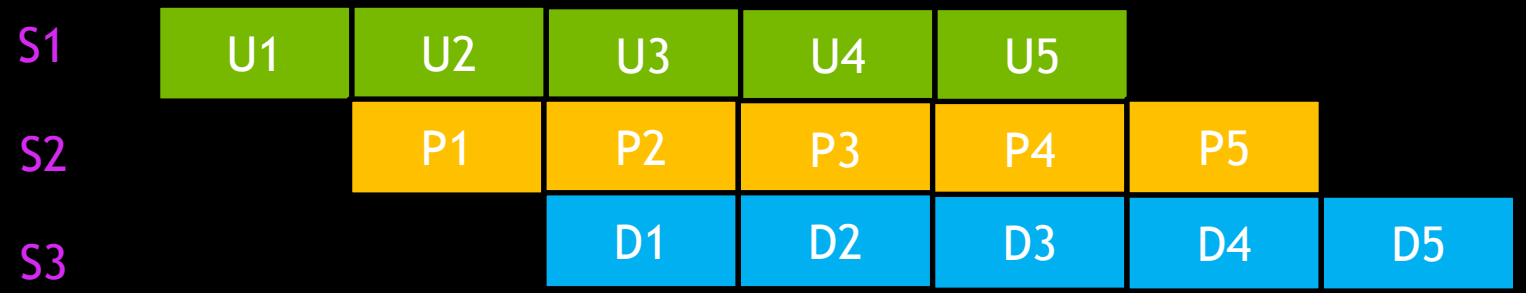
Video Capture, Process and Play - CUDA

```
while (!done){
    \\ Start upload of captured frame
    cudaMemcpyAsync(pGPUbuf[bufNum],pCPUbuf[(bufNum+1)%3],
                            size, cudaMemcpyHostToDevice, uploadStream);

    \\ Start download of processed frame
    cudaMemcpyAsync(pGPUbuf[bufNum+2],pCPUbuf[(bufNum+2)%3],
                           size, cudaMemcpyDeviceToHost, downloadStream);

    \\ 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;
}
```

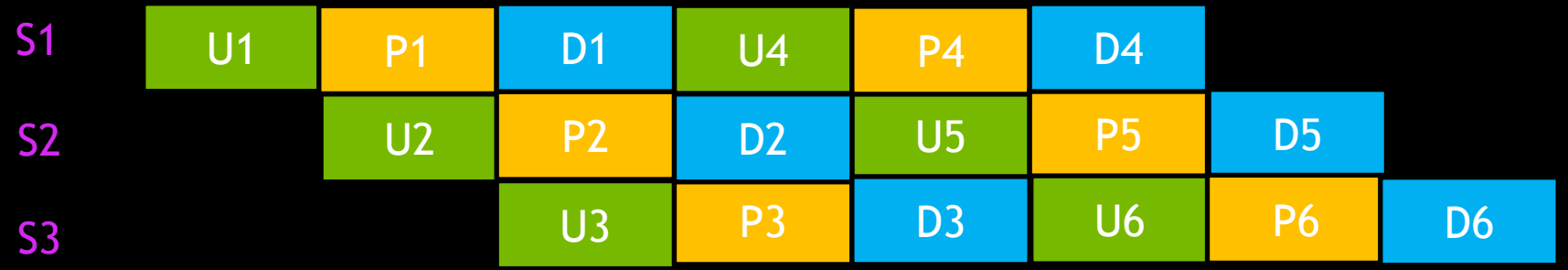


Video Capture, Process and Play - CUDA

Task Oriented Streams



Frame Oriented Streams



Overlap Capture, Process and Play - OpenGL

Capture Thread

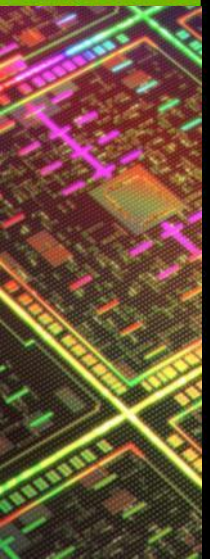
```
// Wait for signal to start upload
CPUWait(startUploadValid[2]);
glWaitSync(startUpload[2]);

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

// Upload
glTexSubImage(texID…);

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

Render Thread

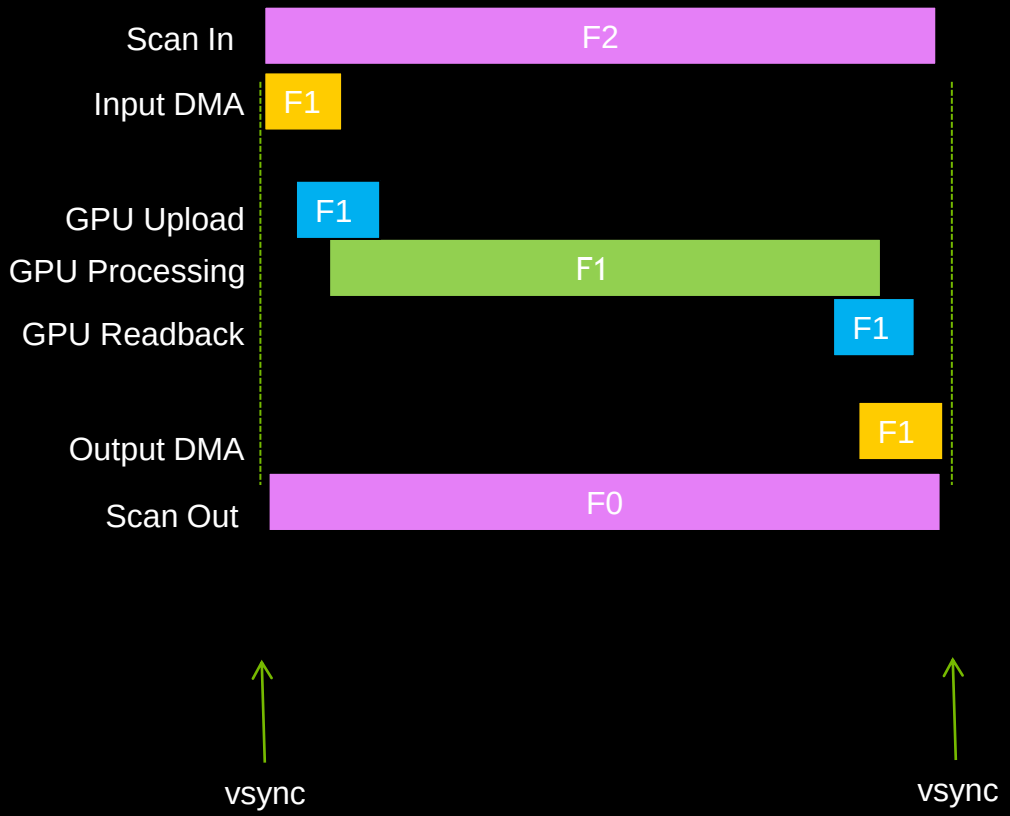
```
// Wait for download to complete
CPUWait(endDownloadValid[3]);
glWaitSync(endDownload[3]);

// 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]);
```

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]);
```



Overlap Copy and Compute - GPUDirect for Video

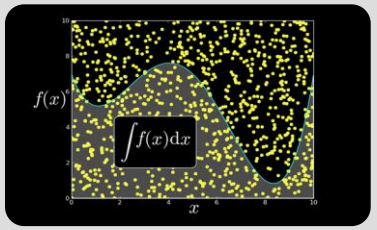


- Unified data transfer API for all Graphics and Compute APIs' objects
  - Video oriented
  - Efficient synchronization
  - GPU shareable system memory
  - Sub-field transfers
  - GPU Asynchronous transfers

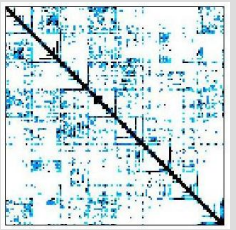
Requires 3rd Party Video I/O SDK support GPUDirect for Video

GPU Compute Libraries

NVIDIA cuBLAS



NVIDIA cuRAND



NVIDIA cuSPARSE

NVIDIA NPP

GPU VSIPL

Vector Signal Image Processing

CULA tools

GPU Accelerated Linear Algebra

MAGMA

ICL UT

Matrix Algebra on GPU and Multicore

NVIDIA cuFFT

ROGUE WAVE SOFTWARE

IMSL Library

CUSP

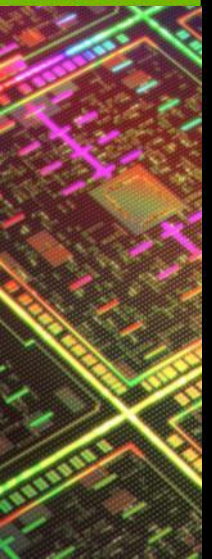
Sparse Linear Algebra

libjacket

Building-block Algorithms for CUDA

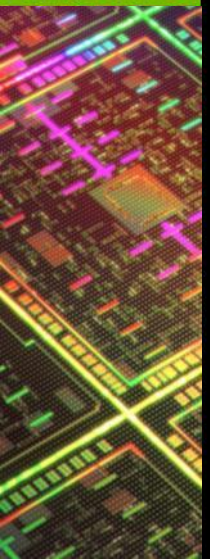
Thrust

C++ STL Features for CUDA



Minimize Interop

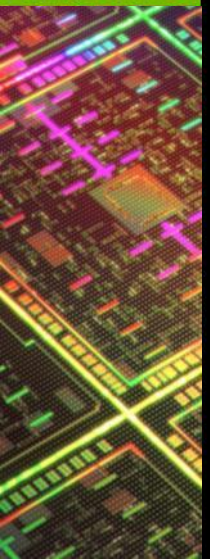
- CUDA/OpenGL and CUDA/DX interop operations introduce a sync point into normally asynchronous GPU operations.
- Single GPU: semaphore acquire operation
- Multi-GPU: semaphore operation and device-to-device memory copy
- Don’t register/unregister OpenGL/DX buffers with CUDA inside the compute/render loop.



devQuery.exe

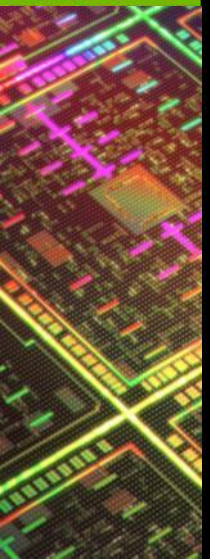
Device 1: "Quadro 5000"

CUDA Driver Version / Runtime Version:	4.2 / 4.2
CUDA Capability Major/Minor version number:	2.0
Total amount of global memory:	2560 MBytes (2684026880 bytes)
(11) Multiprocessors x (32) CUDA Cores/MP:	352 CUDA Cores
GPU Clock rate:	1026 MHz (1.03 GHz)
Memory Clock rate:	1500 Mhz
Memory Bus Width:	320-bit
L2 Cache Size:	655360 bytes
Max Texture Dimension Size (x,y,z)	1D=(65536), 2D=(65536,65535), 3D=(2048,2048,2048)
Max Layered Texture Size (dim) x layers	1D=(16384) x 2048, 2D=(16384,16384) x 2048
Total amount of constant memory:	65536 bytes
Total amount of shared memory per block:	49152 bytes
Total number of registers available per block:	32768
Warp size:	32
Maximum number of threads per multiprocessor:	1536
Maximum number of threads per block:	1024
Maximum sizes of each dimension of a block:	1024 x 1024 x 64
Maximum sizes of each dimension of a grid:	65535 x 65535 x 65535
Maximum memory pitch:	2147483647 bytes
Texture alignment:	512 bytes



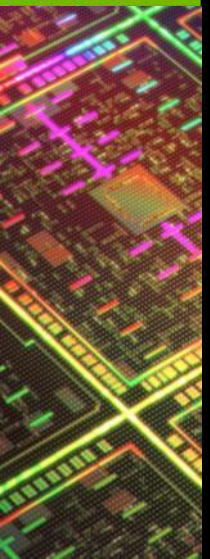
devQuery.exe

```
Concurrent copy and 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
  Concurrent kernel execution:                 Yes
  Alignment requirement for Surfaces:          Yes
  Device has ECC support enabled:              No
  Device is using TCC driver mode:             No
  Device supports Unified Addressing (UVA):    No
  Device PCI Bus ID / PCI location ID:         15 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simu
ltaneously) >
```



bandwidthTest.exe

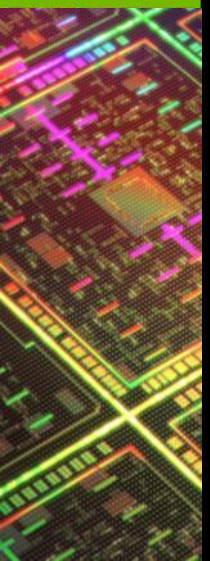
```
Device 1: Quadro 5000
Quick Mode

Host to Device Bandwidth, 1 Device(s), Paged memory
  Transfer Size (Bytes)        Bandwidth(MB/s)
  33554432                     2535.5

Device to Host Bandwidth, 1 Device(s), Paged memory
  Transfer Size (Bytes)        Bandwidth(MB/s)
  33554432                     2012.5

Device to Device Bandwidth, 1 Device(s)
  Transfer Size (Bytes)        Bandwidth(MB/s)
  33554432                     97777.5

[bandwidthTest] test results...
PASSED
```

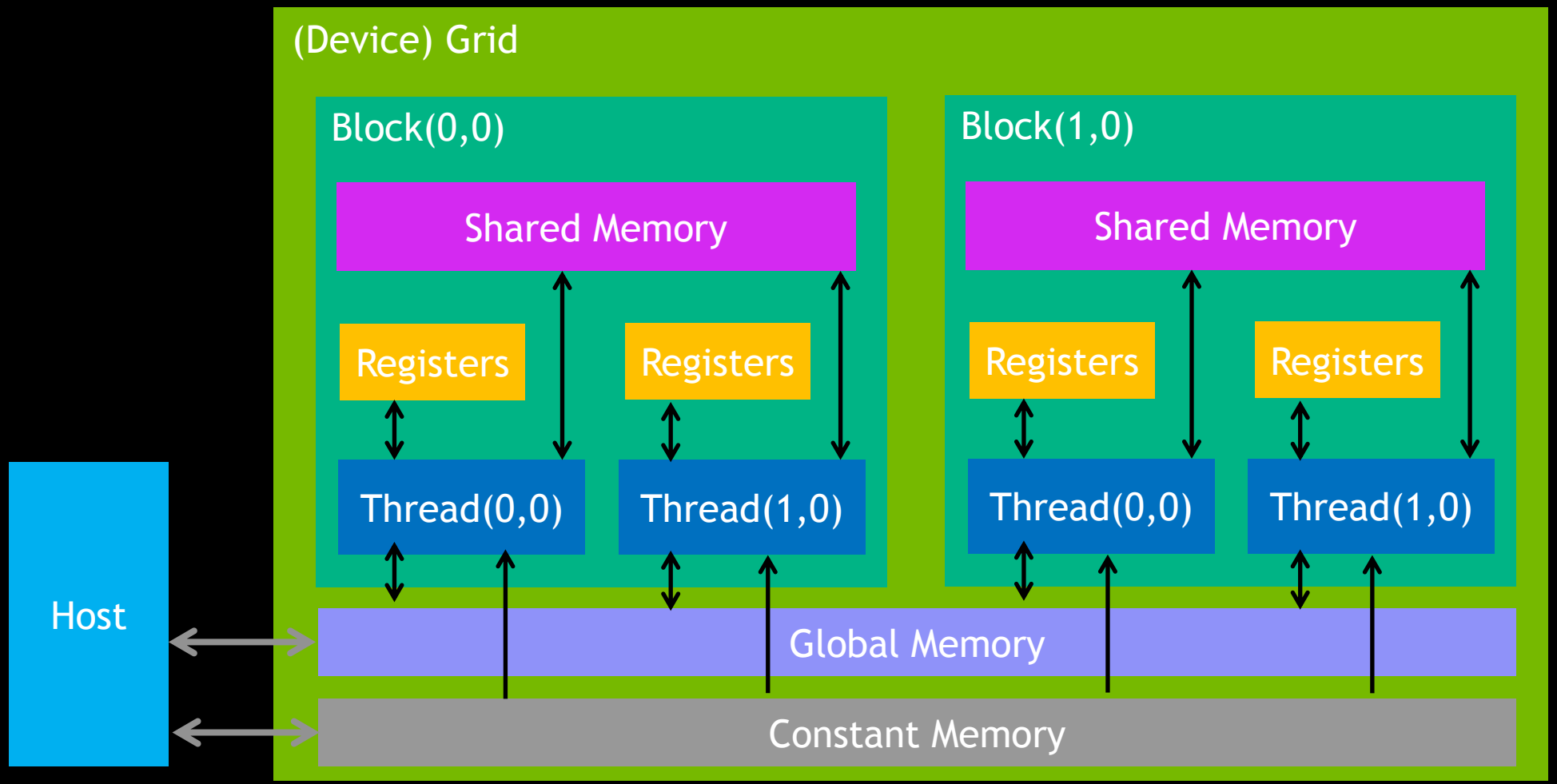


CUDA GPU Timers

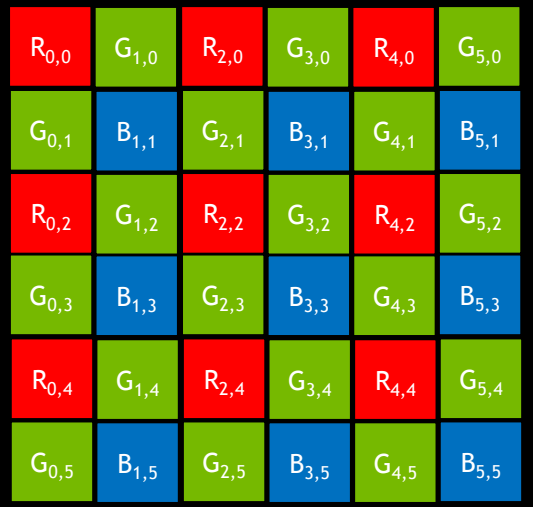
```
\\ Create CUDA events
cudaEventCreate(&start);
cudaEventCreate(&stop);
\\ Record start event
cudaEventRecord( start, 0 );
\\ Execute CUDA kernel
kernel<<<grid,threads>>> ( d_odata, d_idata, size_x, size_y, NUM_REPS
\\ Record stop event
cudaEventRecord( stop, 0 );
\\ Synchronize
cudaEventSynchronize( stop );
\\ Calculate elapsed time
cudaEventElapsedTime( &time, start, stop );
\\ Cleanup
cudaEventDestroy( start );
cudaEventDestroy( stop );
```

GPU Device Memory Hierarchy





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

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

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

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

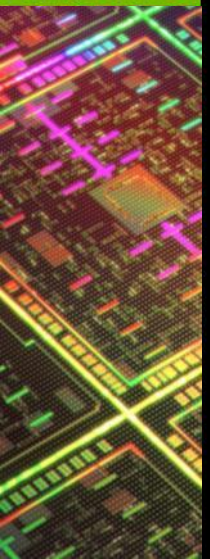
Texture Memory

- 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.



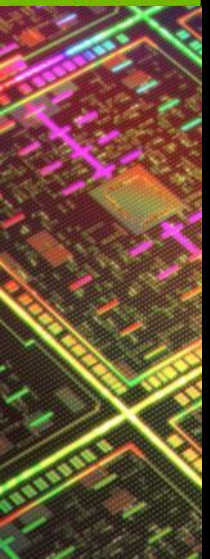
Texture Memory

```
// 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);
```

Constant Memory

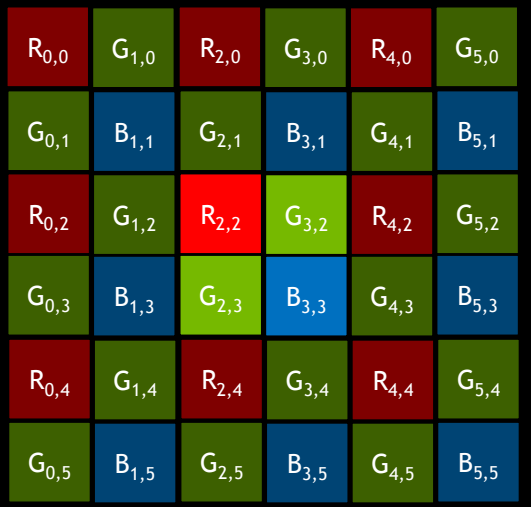
- 64KB/device
- Cached
- As fast as reading from a register

```
// 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);
```



Avoid Thread Divergence

Each thread computes color components for 2x2 quad:



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

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

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

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 5000: 120 GB/sec GMEM Bandwdith

120 * 0.33 = 40 GB/sec = 13 gigapixels

Shared Memory

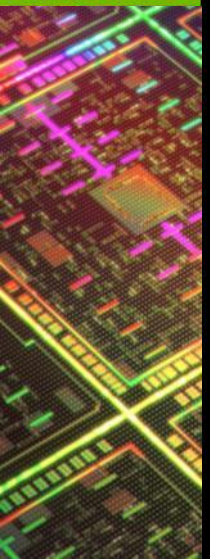
Block Size: 32 x 24
SMEM Size: 34 x 25 (includes apron pixels)
768 values : 850 GMEM reads

.90:1

Quadro 5000: 120 GB/sec GMEM Bandwdith

120 * 0.90 = 108 GB/sec = 36 gigapixels

Use SMEM to improve CGMA!



Shared Memory

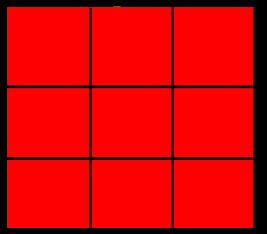
- Faster than global device memory.
- Structure usage to avoid bank conflicts

```
// 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];
```

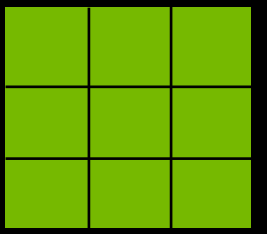
Avoid SMEM Bank Conflicts

Use four different color planes to store raw source pixels:

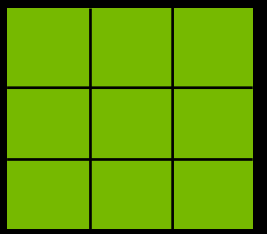
$R_{0,0}$	$G_{1,0}$	$R_{2,0}$	$G_{3,0}$	$R_{4,0}$	$G_{5,0}$
$G_{0,1}$	$B_{1,1}$	$G_{2,1}$	$B_{3,1}$	$G_{4,1}$	$B_{5,1}$
$R_{0,2}$	$G_{1,2}$	$R_{2,2}$	$G_{3,2}$	$R_{4,2}$	$G_{5,2}$
$G_{0,3}$	$B_{1,3}$	$G_{2,3}$	$B_{3,3}$	$G_{4,3}$	$B_{5,3}$
$R_{0,4}$	$G_{1,4}$	$R_{2,4}$	$G_{3,4}$	$R_{4,4}$	$G_{5,4}$
$G_{0,5}$	$B_{1,5}$	$G_{2,5}$	$B_{3,5}$	$G_{4,5}$	$B_{5,5}$



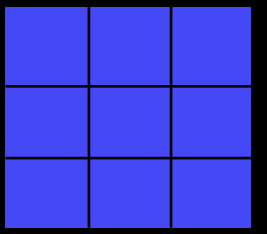
Red SMEM Tile



Green 1 SMEM Tile



Green 2 SMEM Tile

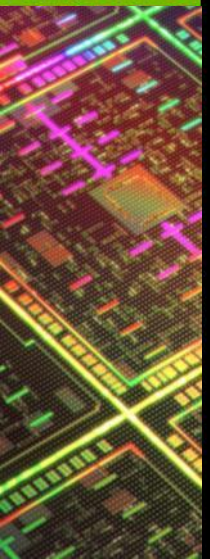


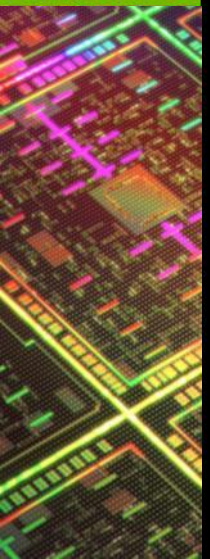
Blue SMEM Tile

Occupancy

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

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





Occupancy

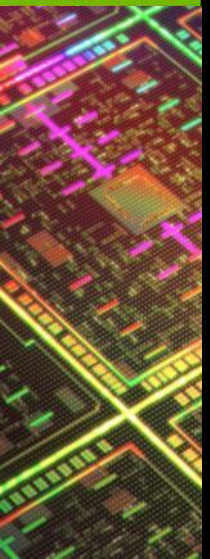
Shared Memory

- Block Size: 32 x 24
- Pixels: 34 x 25 x 4 = 13,056 Bytes
- SMEM/Block (Fermi): 49,152 Bytes - OK!

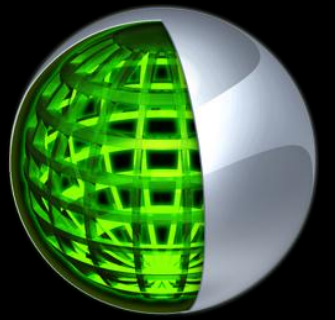
Threads

- Block Size: 32 x 24 = 768 Threads
- Threads/Block (Fermi): 1024 - OK!
- Threads/SM (Fermi): 1536

Occupancy = 2 Blocks / SM = 1536 Threads

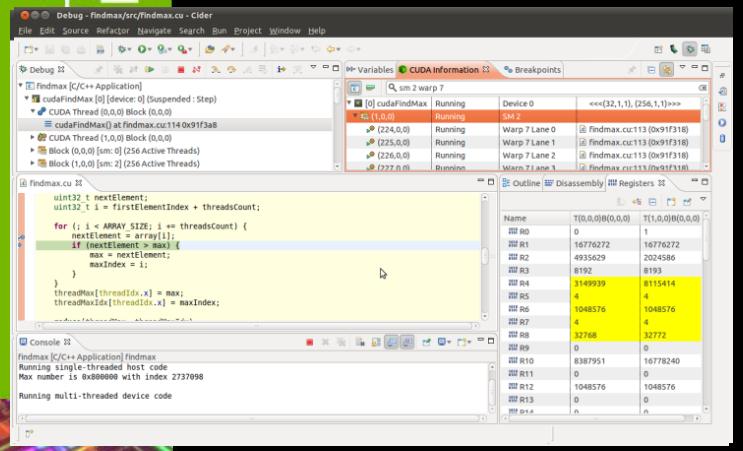


Nsight Visual Studio Edition@GTC'12



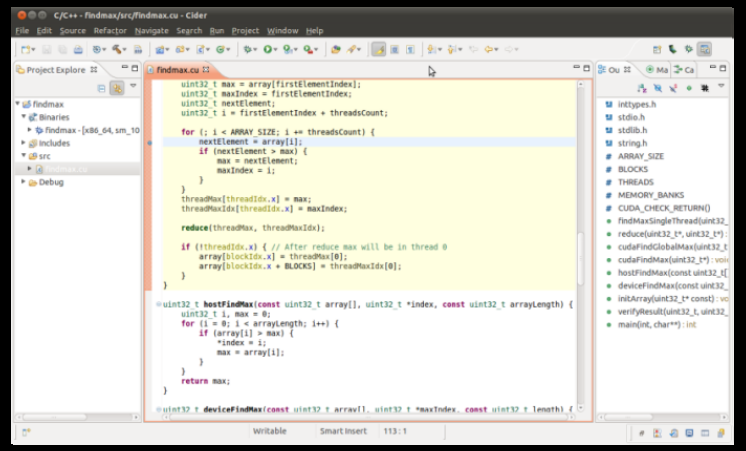
- NVIDIA Nsight Visual Studio Edition Trainings
  - Debugging: Tue:2-3pm, 5-6pm - Wed: 2-3pm - Thu: 9-10am, 4-5pm
  - Profiling: Tue:3-4pm – Wed: 9-10am, 4-5pm – Thu: 2-3pm
- Nsight Lab
  - Tue: 4-5pm – Wed: 10-11am, 3-4pm, 5-6pm - Thu: 10-11am, 3-4pm
- Nsight Visual Studio Edition@NVIDIA Booth/Exhibition Hall
  - Tue,Wed: 12-2pm, 6-8pm – Thu: 12-2pm

NVIDIA® Nsight™ Eclipse Edition



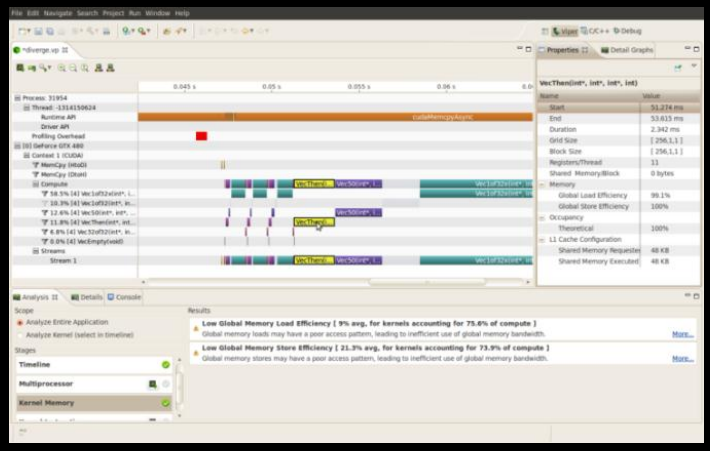
CUDA-Aware Editor

- Automated CPU to GPU code refactoring
- Semantic highlighting of CUDA code
- Integrated code samples & docs



Nsight Debugger

- Simultaneously debug of CPU and GPU
- Inspect variables across CUDA threads
- Use breakpoints & single-step debugging



Nsight Profiler

- Quickly identifies performance issues
- Integrated expert system
- Source line correlation

Available for Linux and Mac OS


GPU TECHNOLOGY CONFERENCE
Questions?