GPU TECHNOLOGY CONFERENCE

Maximizing GPU Efficiency in Extreme Throughput Applications

The Fairmont San Jose | October 2, 2009, 2:00PM | Joe Stam

NVIDIA.

Motivation

- GPUs have dedicated memory which has 5–10X the bandwidth of CPU memory, this is a tremendous advantage
- New developers are sometimes discouraged by the perceived overhead of transferring data between GPU and CPU memory.

Today we'll show how to properly transfer data in high throughput applications, and reduce or eliminate the transfer burden.

GPU TECHNOLOGY CONFERENCE

AGENDA

- Asynchronous APIs
- Data Acquisition
- CUDA Streams
- “Zero-Copy”

NVIDIA.

Typical Approach

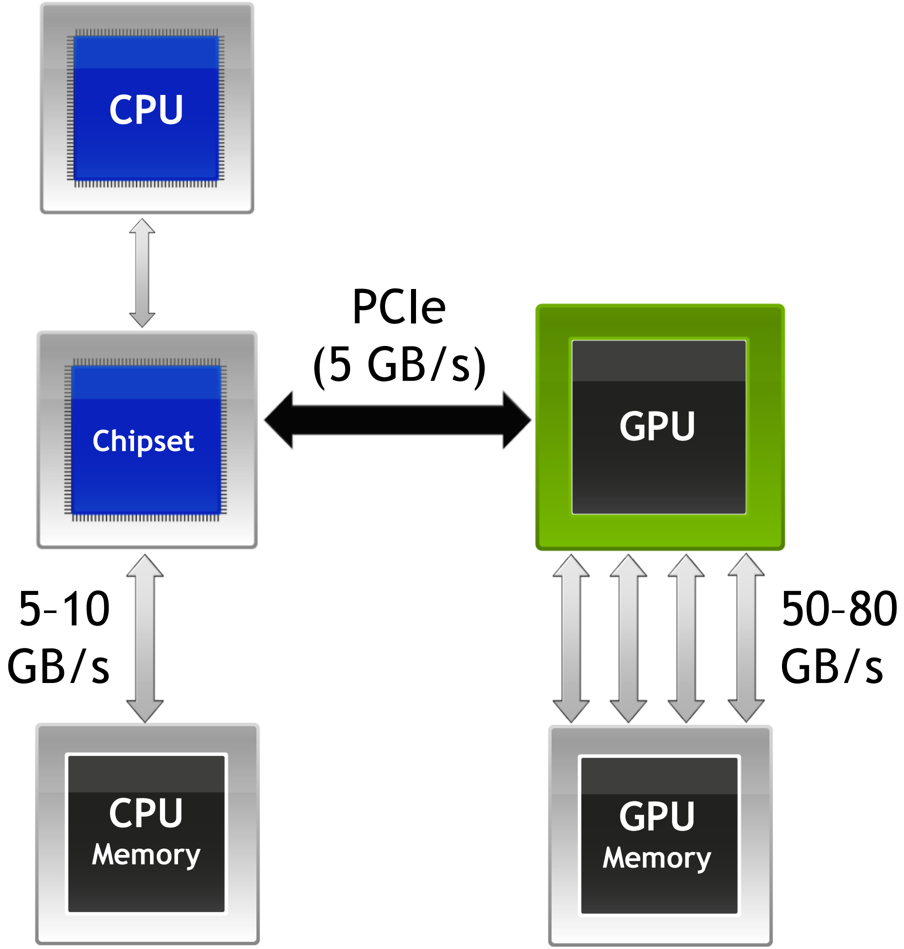
Copy Data from CPU Memory to GPU Memory

↓

Run CUDA Kernel(s)

↓

Copy Data from GPU memory to CPU Memory



*Averaged observed bandwidth

GPU TECHNOLOGY CONFERENCE

Synchronous Functions

- Standard CUDA C functions are Synchronous
- Kernel launches are:
  - Runtime API: Asynchronous
  - Driver API: `cuLaunchGrid()` or `cuLaunchGridAsync()`
- Synchronous functions block on any prior asynchronous kernel launches

NVIDIA.

Example

```
cudaMemcpy(...);                        ← Doesn't return until copy is complete
myKernel<<<grid,block>>>(...);          ← Returns immediately
cudaMemcpy(...);                        ← Waits for myKernel to complete, then starts copying. Doesn't return until copy is complete.
```

`cudaMemcpy(...);` — Doesn't return until copy is complete

`myKernel<<<grid,block>>>(...);` — Returns immediately

`cudaMemcpy(...);` — Waits for **`myKernel`** to complete, then starts copying. Doesn't return until copy is complete.

`cudaDeviceSetFlags()` function sets behavior. Tradeoff between CPU cycles and response speed

- `cudaDeviceScheduleSpin`
- `cudaDeviceScheduleYield`
- `cudaDeviceBlockingSync`

Driver API has equivalent context creation flags

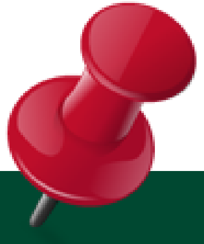
GPU TECHNOLOGY CONFERENCE

NVIDIA.

GPU TECHNOLOGY CONFERENCE

Asynchronous APIs

- All Memory operations can also be asynchronous, and return immediately
- Memory must be allocated as ‘pinned’ using
  - `cuMemHostAlloc()`
  - `cudaHostAlloc()`
  - Older version of these functions `cuMemAllocHost()` `cudaMallocHost()` also work, but don’t have option flags



PINNED memory allows direct DMA transfers by the GPU to and from system memory. It’s locked to a physical address

NVIDIA.

Asynchronous APIs (Cont.)

- Copies & Kernels are queued up in the GPU
- Any launch overhead is overlapped

- Synchronous calls should be done outside critical sections — some of these are expensive!
  - Initialization
  - Memory allocations
  - Stream / Event creation
  - Interop resource registration

GPU TECHNOLOGY CONFERENCE

Example

```
cudaMemcpyAsync( void * dst,
                 void * src,
                 size_t count,
                 enum cudaMemcpyKind kind,
                 cudaStream_t stream)
```

More on streams soon, for now assume stream = 0

```
cudaMemcpyAsync(…);
myKernel<<<grid,block>>>(…);
cudaMemcpyAsync(…);
```

- `cudaMemcpyAsync(…);` — Returns immediately
- `myKernel<<<grid,block>>>(…);` — Returns immediately
- `cudaMemcpyAsync(…);` — Returns immediately

CPU does other stuff here

```
cudaThreadSynchronize();
```

Waits for everything on the GPU to finish, then returns

NVIDIA.

Events Can Be Used to Monitor Completion

- `cudaEvent_t / CUevent`
  - Created by **`cudaEventCreate`**`()` / **`cuEventCreate`**`()`

```
cudaEvent_t HtoDdone;
cudaEventCreate(&HtoDdone,0);
cudaMemcpyAsync(dest,source,bytes,cudaMemcpyHostToDevice,0);
cudaEventRecord(HtoDdone);
myKernel<<<grid,block>>>(…);
cudaMemcpyAsync(dest,source,bytes,cudaMemcpyDeviceToHost,0);
```

CPU can do stuff here

```
cudaEventSynchronize(HtoDdone);
```

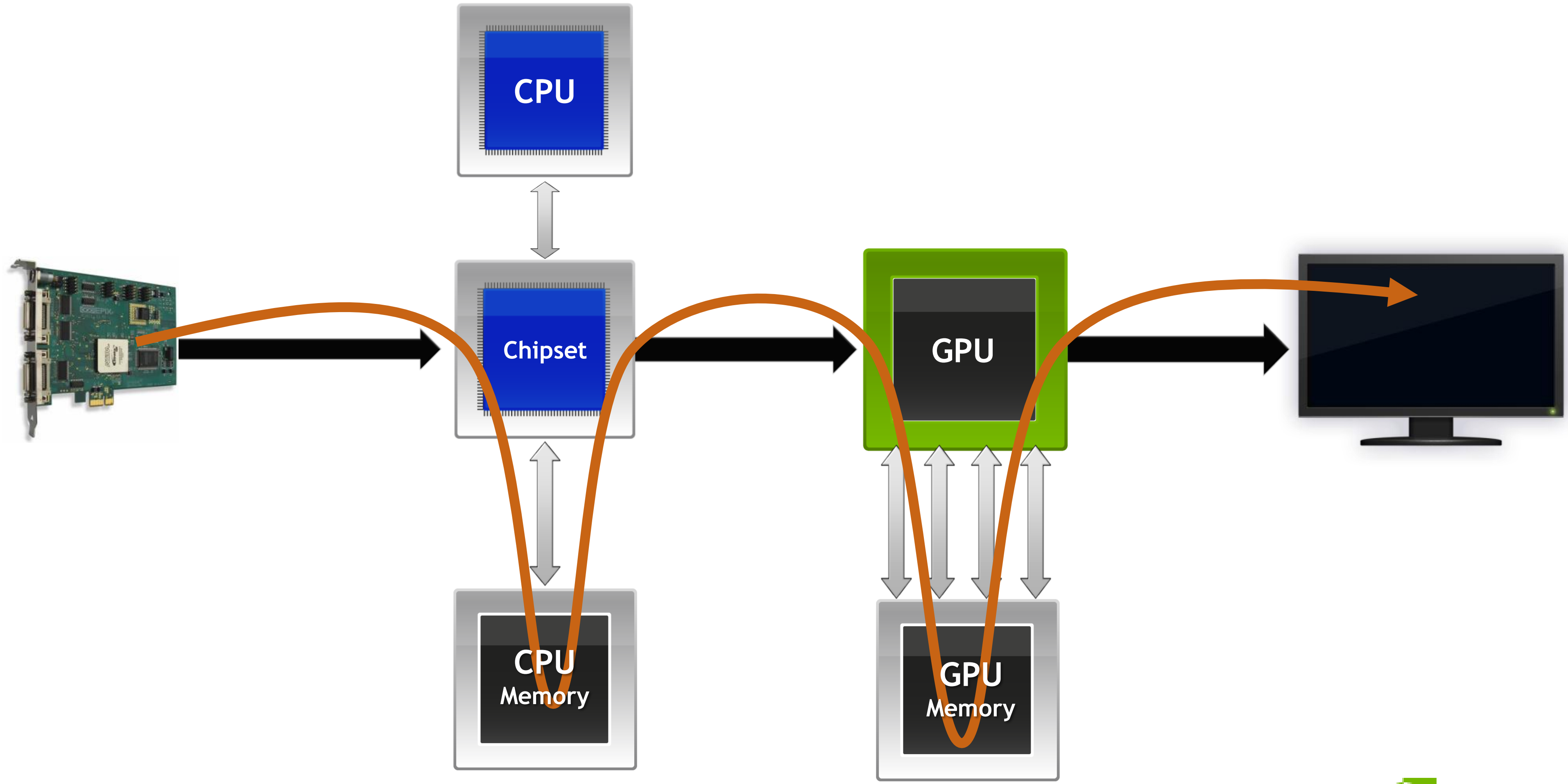
Waits just for everything before `cuEventRecord(HtoDdone)` to complete, then returns

The first memory copy is done, so the memory at source could be used again by the CPU

```
cudaThreadSynchronize();
```

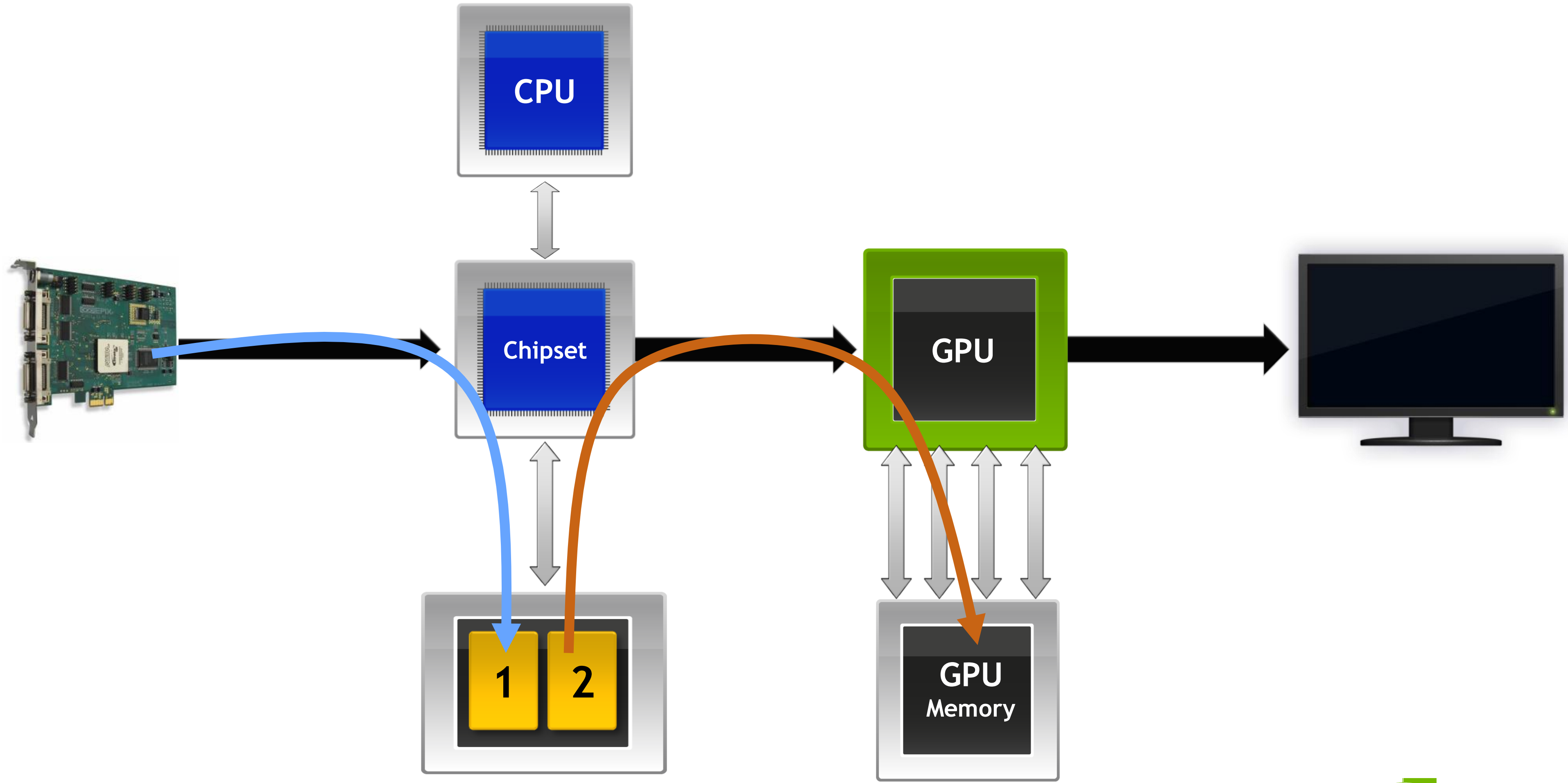
Waits for everything on the GPU to finish, then returns

Acquiring Data From an Input Device



NVIDIA.

Strategy: Overlap Acquisition With Transfer



NVIDIA.

Strategy: Overlap Acquisition With Transfer

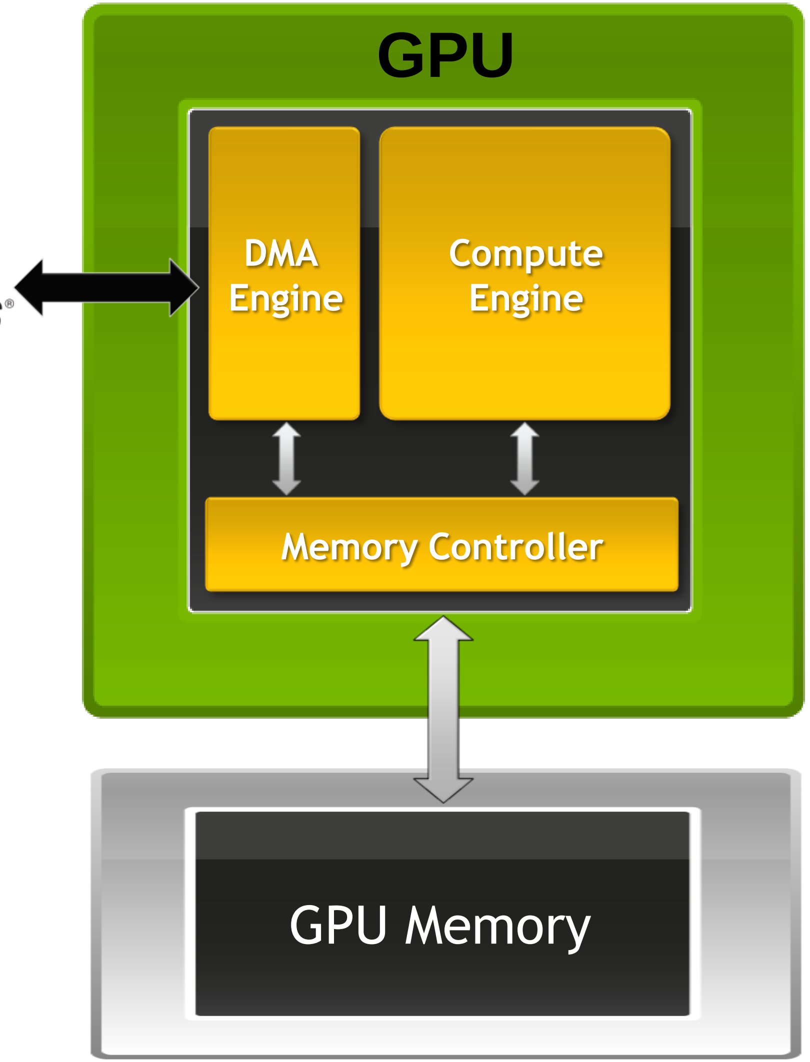
- Allocate 2 pinned CPU buffers, ping-pong between them

```
int bufNum = 0;
void * pCPUbuf[2];
... Allocate buffers
while (!done)
{
    cudaMemcpyAsync(pGPUbuf,pCPUbuf[(bufNum+1)%2],size,
                    cudaMemcpyHostToDevice,0);
    myKernel1<<<…>>>(GPUbuf…);
    myKernel2<<<…>>>(GPUbuf…);
    … other GPU stuff, all asynchronous

    GrabMyFrame(pCPUbuf[bufNum]);
    … other CPU stuff

    cudaThreadSynchronize();
    bufNum++; bufNum %=2;
}
```

Concurrent

CUDA Streams

- NVIDIA GPUs with Compute Capability >= 1.1 have a dedicated DMA engine
- DMA transfers over PCIe can be concurrent with CUDA kernel execution*
- **Streams** allows independent concurrent in-order queues of execution
  - `cudaStream_t, CUstream`
  - `cudaStreamCreate(), cuStreamCreate()`
- Multiple streams exist within a single context, they share memory and other resources



*1D Copies only! `cudaMemcpy2DAsync` cannot overlap.

Stream Parameter

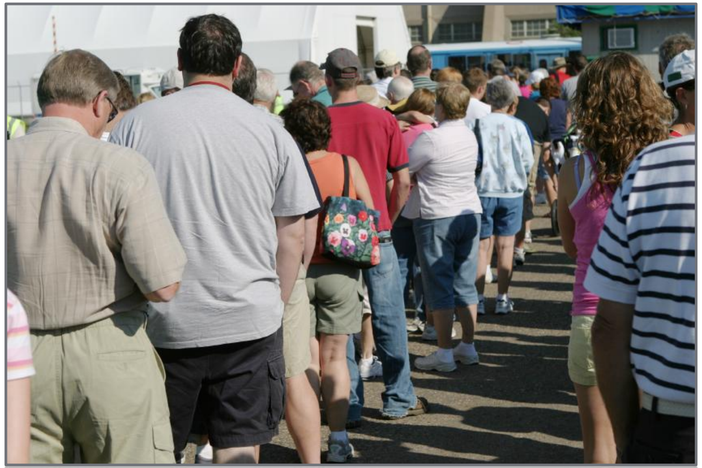
- All Async function varieties have a **stream** parameter
- Runtime Kernel Launch

```
<<< GridSize, BlockSize,SMEM Size, Stream>>>
```

- Driver API

```
cuLaunchGridAsync(function, width, height,
  stream)
```

- Copies & Kernel launches with the **same** stream parameter execute **in-order**



GPU TECHNOLOGY CONFERENCE

CUDA Streams

Independent Tasks

TASK A	TASK B
COPY A1	COPY B1
KERNEL A1	COPY B2
KERNEL A2	KERNEL B1
KERNEL A3	COPY B3
COPY A2	COPY B4

Scheduling on GPU

Copy Engine	Compute Engine
COPY A1	
COPY B1	KERNEL A1
COPY B2	KERNEL A2
	KERNEL A3
COPY A2	KERNEL B1
COPY B3	
COPY B4	

NVIDIA.

GPU TECHNOLOGY CONFERENCE

Avoid Serialization!

STREAM A

- COPY A1
- KERNEL A1
- KERNEL A2
- KERNEL A3
- COPY A2

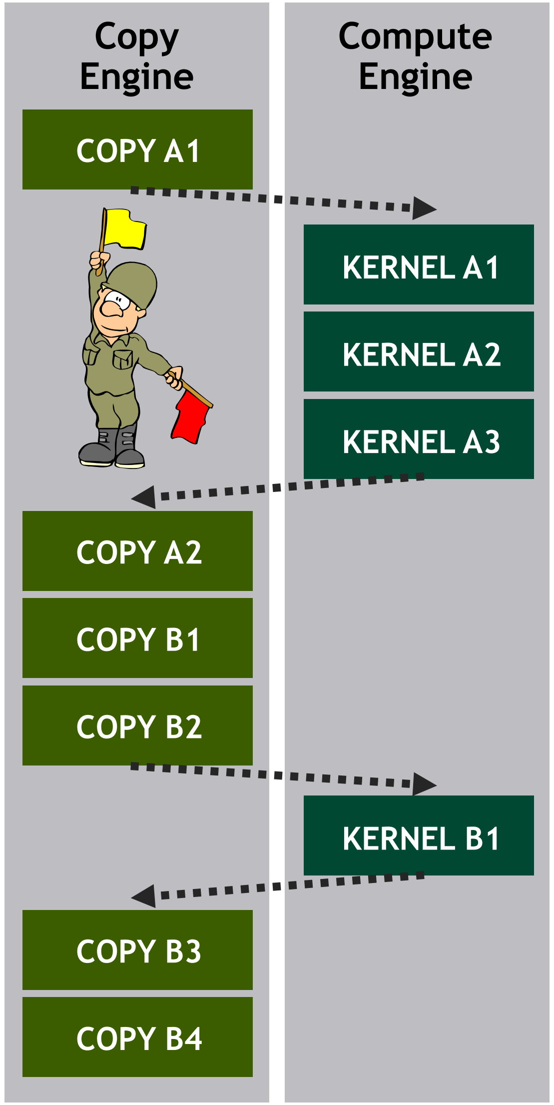
STREAM B

- COPY B1
- COPY B2
- KERNEL B1
- COPY B3
- COPY B4

WRONG WAY!

```
CudaMemcpyAsync(A1…,StreamA);
KernelA1<<<…,StreamA>>>();
KernelA2<<<…,StreamA>>>();
KernelA3<<<…,StreamA>>>();
CudaMemcpyAsync(A2…,StreamA);

CudaMemcpyAsync(B1…,StreamB);
CudaMemcpyAsync(B2…,StreamB);
KernelB1<<<…,StreamB>>>();
CudaMemcpyAsync(B2…,StreamB);
CudaMemcpyAsync(B2…,StreamB);
```

- Engine queues are filled in the order code is executed



Stream Code Order

STREAM A

- COPY A1
- KERNEL A1
- KERNEL A2
- KERNEL A3
- COPY A2

STREAM B

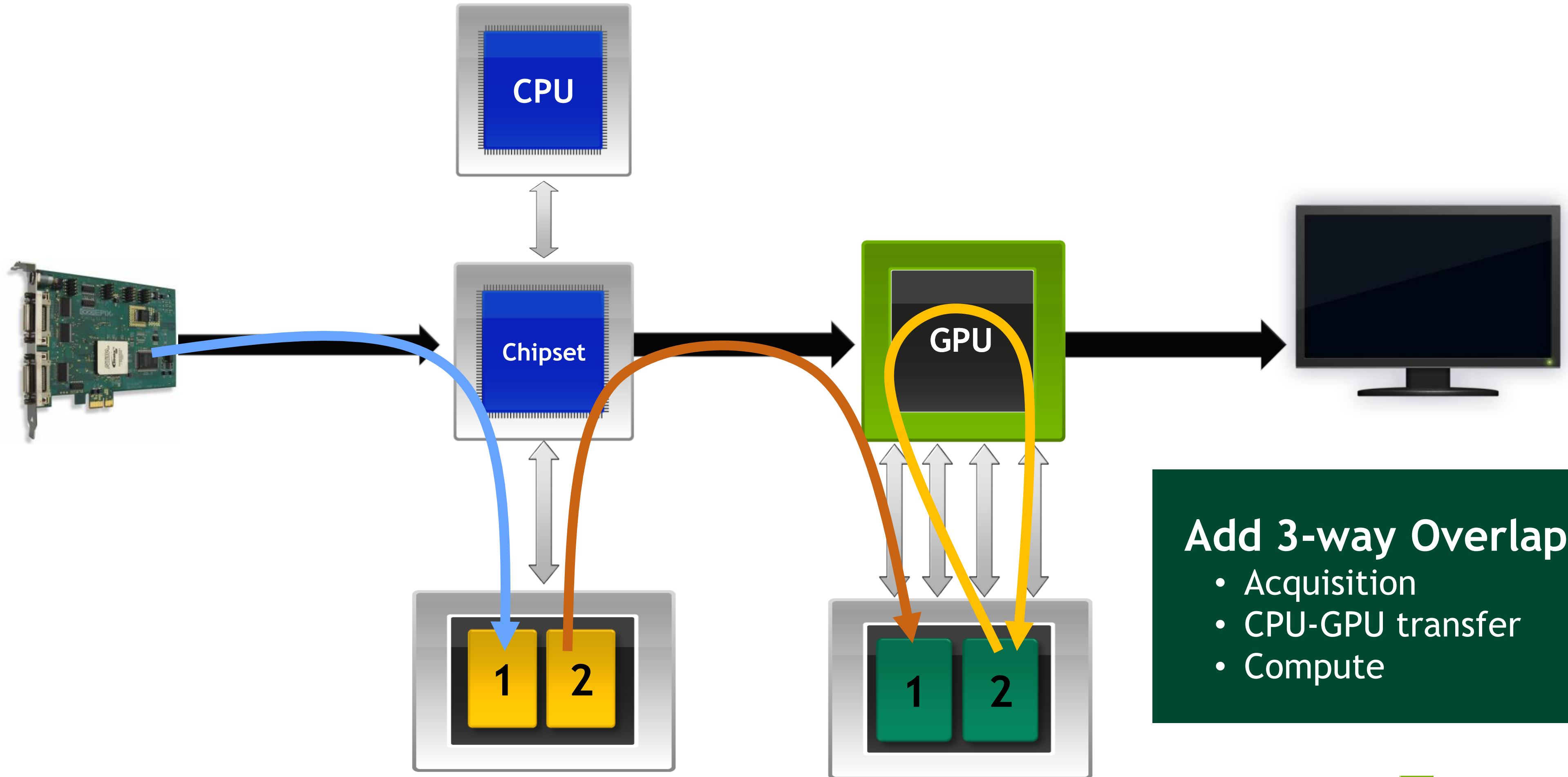
- COPY B1
- COPY B2
- KERNEL B1
- COPY B3
- COPY B4

CORRECT WAY!

```
CudaMemcpyAsync(A1…,StreamA);
KernelA1<<<…,StreamA>>>();
KernelA2<<<…,StreamaA>>>();
KernelA3<<<…,StreamA>>>();

CudaMemcpyAsync(B1…,StreamB);
CudaMemcpyAsync(B2…,StreamB);
KernelB1<<<…,StreamB>>>();

CudaMemcpyAsync(A2…,StreamA);

CudaMemcpyAsync(B2…,StreamB);
CudaMemcpyAsync(B2…,StreamB);
```

Copy Engine	Compute Engine
COPY A1	
COPY B1	KERNEL A1
COPY B2	KERNEL A2
	KERNEL A3
COPY A2	KERNEL B1
COPY B3	
COPY B4	

Revisit Our Data I/O Example



Add 3-way Overlap:
- Acquisition
- CPU-GPU transfer
- Compute

GPU TECHNOLOGY CONFERENCE

3-Way Overlap

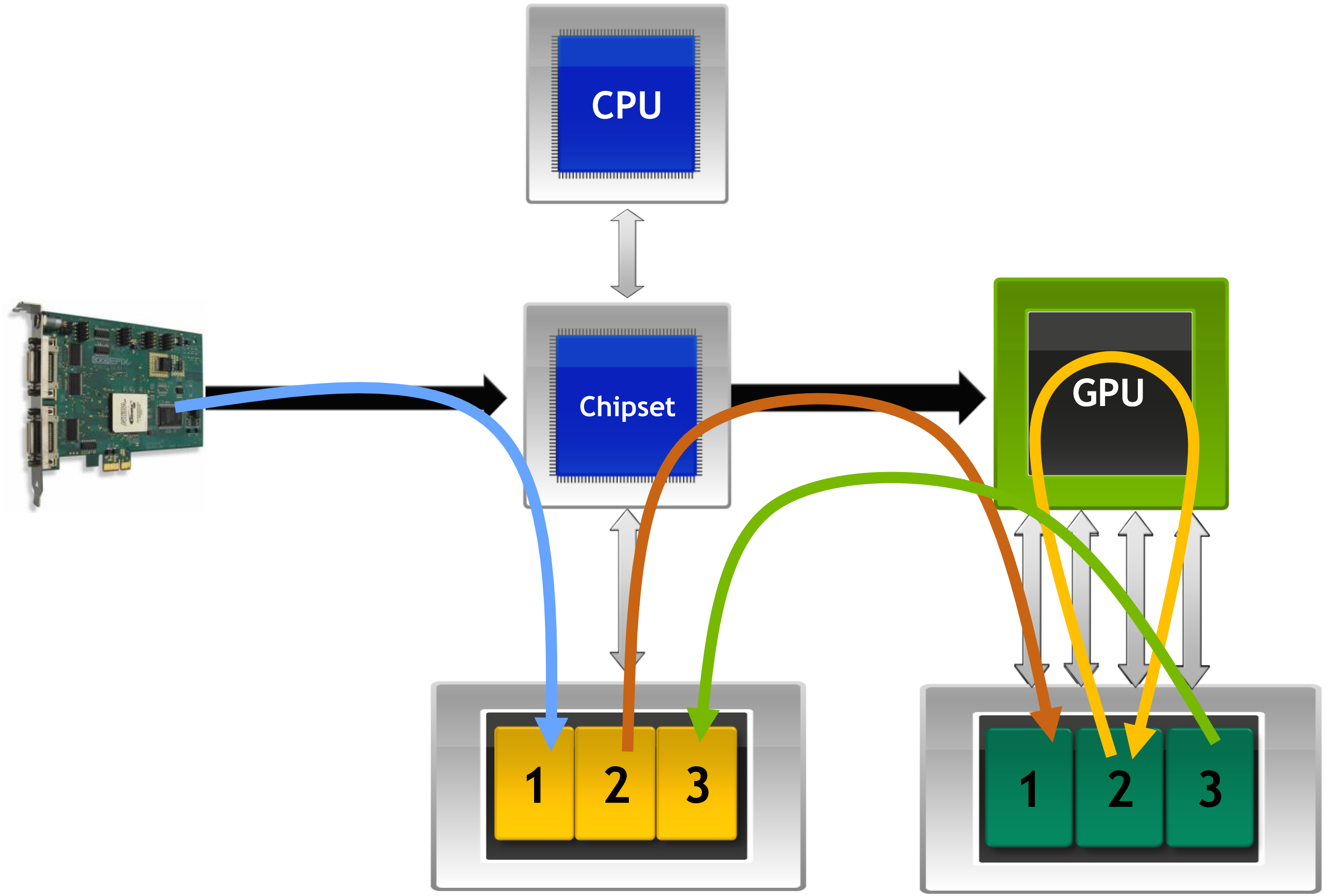
- As before, allocate two CPU buffers
- Also allocate two GPU buffers

```
int bufNum = 0;
void * pCPUbuf[2];
void * pGPUbuf[2];
cudaStream_t copyStream;
cudaStream_t computeStream;

// Allocate Buffers
cudaHostAlloc(&(pCPUbuf[0]),size,0);
cudaHostAlloc(&(pCPUbuf[1]),size,0);
cudaMalloc(&(pGPUbuf[0]),size,0);
cudaMalloc(&(pGPUbuf[1]),size,0);

// Create Streams
cudaStreamCreate(&copyStream,0);
cudaStreamCreate(&computeStream,0);
```

3-Way Overlap (Cont.)

```
while (!done)
{
        cudaMemcpyAsync(pGPUbuf[bufNum],pCPUbuf[(bufNum+1)%2],size,
                        cudaMemcpyHostToDevice,copyStream);

        myKernel1<<<gridSz,BlockSz,0,computeStream>>>(pGPUbuf[(bufNum+1)%2]…);
        myKernel2<<<gridSz,BlockSz,0,computeStream>>>(pGPUbuf[(bufNum+1)%2]…);
        … other GPU stuff, all asynchronous

        GrabMyFrame(pCPUbuf[bufNum]);
        … other CPU stuff

        cudaThreadSynchronize();
        bufNum++; bufNum %=2;
}
```

What About Readback?



Readback

```
while (!done)
{
        cudaMemcpyAsync(pGPUbuf[bufNum],pCPUbuf[(bufNum+1)%3],size,
                        cudaMemcpyHostToDevice,copyStream);

        cudaMemcpyAsync(pGPUbuf[bufNum+2],pCPUbuf[(bufNum+2)%3],size,
                        cudaMemcpyDeviceToHost,copyStream);

        myKernel1<<<gridSz,BlockSz,0,computeStream>>>(pGPUbuf[(bufNum+1)%3]…);
        myKernel2<<<gridSz,BlockSz,0,computeStream>>>(pGPUbuf[(bufNum+1)%3]…);
        … other GPU stuff, all asynchronous

        GrabMyFrame(pCPUbuf[bufNum]);
        … other CPU stuff

        cudaThreadSynchronize();
        bufNum++; bufNum %=3;
}
```

GPU TECHNOLOGY CONFERENCE

4-Way Overlap?

- **NEW** hardware adds a 2nd copy engine!
- Simultaneous upload and downloading
- So just add a new stream! (still works with prior hardware, just serialized)

```
while (!done)
{
        cudaMemcpyAsync(pGPUbuf[bufNum],pCPUbuf[(bufNum+1)%3],size,
                        cudaMemcpyHostToDevice,uploadStream);

        cudaMemcpyAsync(pGPUbuf[bufNum+2],pCPUbuf[(bufNum+2)%3],size,
                        cudaMemcpyDeviceToHost,downloadStream);

        myKernel1<<<gridSz,BlockSz,0,computeStream>>>(pGPUbuf[(bufNum+1)%3]…);
        myKernel2<<<gridSz,BlockSz,0,computeStream>>>(pGPUbuf[(bufNum+1)%3]…);
        … other GPU stuff, all asynchronous

        GrabMyFrame(pCPUbuf[bufNum]);
        … other CPU stuff

        cudaThreadSynchronize();
        bufNum++; bufNum %=3;
}
```

Host Memory Mapping, a.k.a "Zero-Copy"

The easy way to achieve copy/compute overlap!

1. **Enable Host Mapping***

   Runtime: `cudaSetDeviceFlags()` with `cudaDeviceMapHost` flag

   Driver: `cuCtxCreate()` with `CU_CTX_MAP_HOST`

2. **Allocate pinned CPU memory**

   Runtime: `cudaHostAlloc()`, use `cudaHostAllocMapped` flag

   Driver: `cuMemHostAlloc()` use `CUDA_MEMHOSTALLOC_DEVICEMAP`

3. **Get a CUDA device pointer to this memory**

   Runtime: `cudaHostGetDevicePointer()`

   Driver: `cuMemHostGetDevicePointer()`

4. **Just use that pointer in your kernels!**

Note: For Ion™ and other Unified Memory Architecture (UMA) GPUs zero-copy eliminates data transfer altogether!

*Check the **`canMapHostMemory`** / **`CU_DEVICE_ATTRIBUTE_CAN_MAP_HOST_MEMORY`** device property flag to see if Zero-Copy is available.

Zero-Copy Guidelines

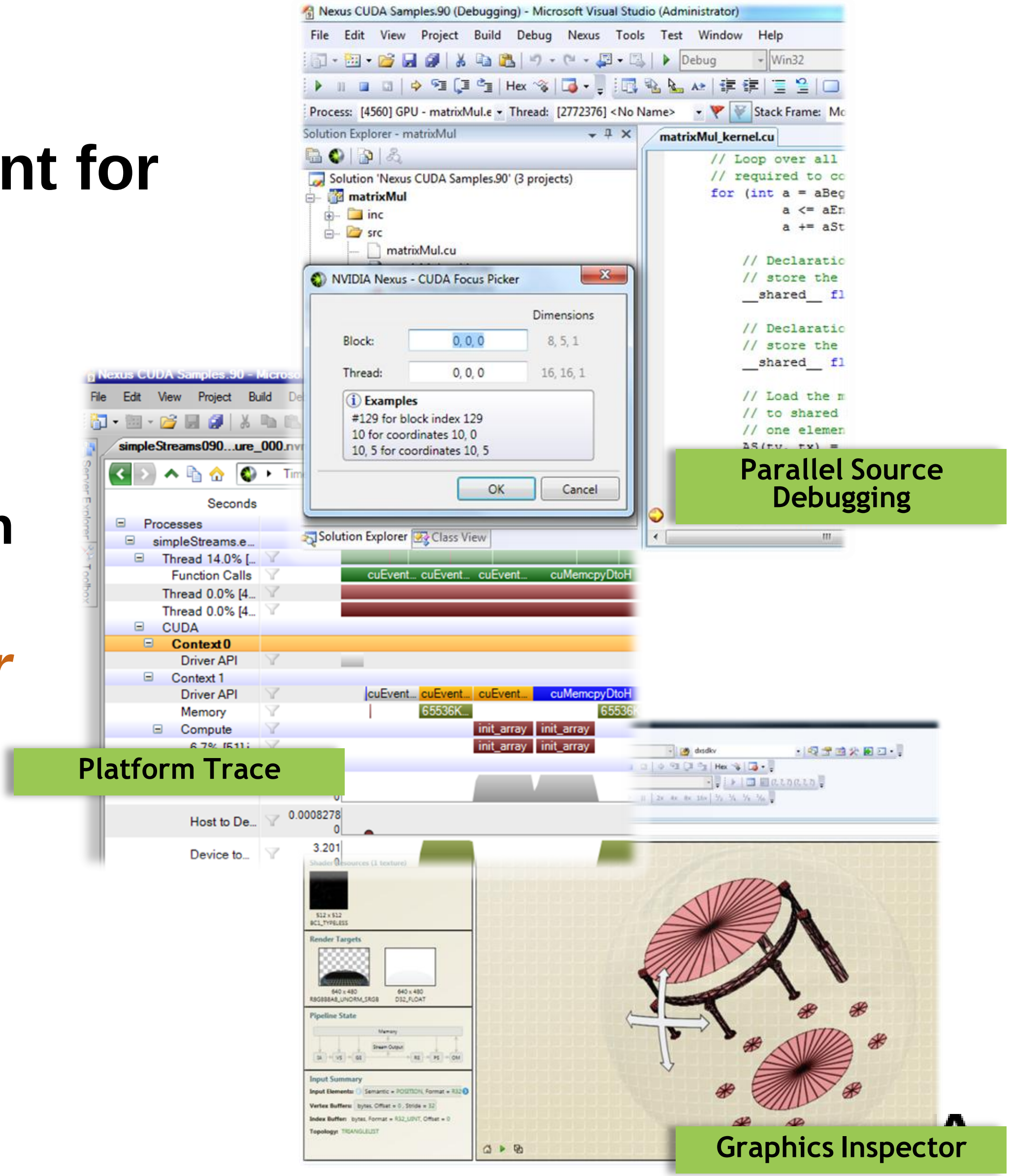
- Data is transferred over the PCIe bus automatically, but it's slow
- Use when data is only read/written once
- Use for very small amounts of data (new variables, CPU/GPU communication)
- Use when compute/memory ratio is very high and occupancy is high, so latency over PCIe is hidden
- Coalescing is *critically* important!

GPU TECHNOLOGY CONFERENCE

NVIDIA NEXUS

The first development environment for massively parallel applications.

- **Hardware GPU Source Debugging**
- **Platform-wide Analysis**
- **Complete Visual Studio integration**



Timeline trace is excellent for analyzing streams!

Register for the Beta here at GTC!
http://developer.nvidia.com/object/nexus.html

Beta available October 2009

Releasing in Q1 2010

GPU TECHNOLOGY CONFERENCE

Questions?

NVIDIA.