



The CUDA architecture

The Art of performance optimization

wbraithwaite@nvidia.com



Overall optimization strategies



- **Maximize parallel execution**
 - Exposing data parallelism in algorithms
 - Overlap memory access with computation
 - Keep the hardware busy
- **Maximize memory bandwidth**
 - Access data efficiently
- **Maximize instruction throughput**
 - Use as few clock cycles as possible

The Art of Performance Optimization



- Exploiting parallelism
- Sending data to the device
- CUDA architecture refresher
- Execution configuration
- Speeding up memory access
- Instruction optimization

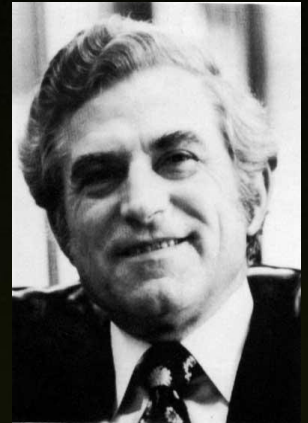


Amdahl's Law – Example

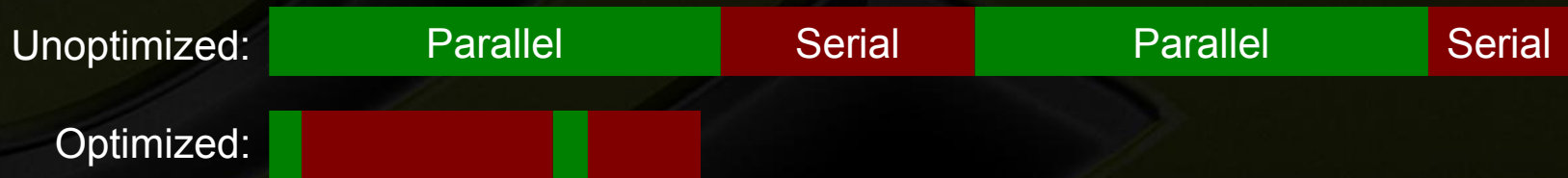


- **P = parallel proportion**
- **N = number of procs**

$$S = \frac{1}{(1 - P) + \frac{P}{N}}$$



- **Assume N → infinity**
- **Only ¾ of program can be parallelized**
- **S = 4**



- **The maximum speedup can only be 4x**

Theoretical Bandwidth



● Device Bandwidth of GTX 280

$$\bullet \underbrace{1107 * 10^6}_{\text{Memory clock (Hz)}} * \underbrace{(512 / 8)}_{\text{Memory interface (bytes)}} * \overset{\text{DDR}}{\downarrow} 2 / 10^9 = 141.6 \text{ GB/s}$$

● Some specs report 132 GB/s

- They use 1024^3 B/GB conversion rather than 10^9

Effective Bandwidth





- **Effective Bandwidth (GB/s) =**

- $((B_R + B_W) / 10^9) / \text{time}$

- **Example of copying array of N floats**

- $N * \text{sizeof}(\text{float}) / 10^9 * 2 / \text{num_seconds} = \text{GB/s}$


Array size
(bytes)


Read and
write

- **Our goal is to make effective bandwidth as close to theoretical bandwidth as possible**

Host ↔ Device – Overview



- **NVIDIA GPUs have dedicated memory**
 - nearly 10X the bandwidth of CPU memory, this is a tremendous advantage
 - 141 GB/s peak (GTX 280) vs. 6 GB/s peak (PCI-e x16 Gen2)
- **Developers may be discouraged by the overhead of transferring data between Host and Device**
- **Some ways to fix this:**
 - Avoid it...
 - Make it faster...
 - Hide it...

Host ↔ Device – Minimize transfers



- Intermediate data structures can be allocated, operated on, and deallocated without ever copying them to host memory
- Sometimes it's better to recompute than to cache
- Even low-parallelism computations can sometimes be faster than transferring back and forth
- Use graphics interoperability...

Graphics Interop – Overview



- **OpenGL buffer objects can be mapped into the device's address-space**
 - Direct3D9 Vertex objects can also be mapped (see programming guide for details)
 - Data is accessed like global memory in the kernel
 - Can remove host ↔ device transfer entirely
 - Automatic DMA from Tesla to Quadro (currently via host)
- **Image data can be displayed from PBOs**
 - using `glDrawPixels` / `glTexImage2D`
 - Requires high-speed copy in video memory

A red starburst graphic with multiple points, containing text.

See SDK!
postProcessGL
simpleGL...

Graphics Interop – Details



- Register a buffer object with CUDA-C
 - `cudaGLRegisterBufferObject(GLuint buffObj)`
- Map a registered buffer object to device memory
 - `cudaGLMapBufferObject(void** devPtr, GLuint buffObj)`
 - Returns an address in global memory
- Use returned memory address in your kernel
- Unmap the buffer object prior to use by OpenGL
 - `cudaGLUnmapBufferObject(GLuint buffObj)`
- Unregister the buffer object
 - `cudaGLUnregisterBufferObject(GLuint buffObj)`
 - Only needed if the buffer is a render-target
- Use the buffer object in your OpenGL code...

Graphics Interop – Example



- Dynamic CUDA-generated texture:

```
// setup code:
cudaGLRegisterBufferObject(pbo);

// CUDA texture generation code:
unsigned char *d_buffer;
cudaGLMapBufferObject((void**)&d_buffer, pbo);
prep_texture_kernel<<<...>>>(d_buffer);
cudaGLUnmapBufferObject(pbo);

// OpenGL rendering code:
glBindBuffer(GL_PIXEL_UNPACK_BUFFER_ARB, pbo);
glBindTexture(GL_TEXTURE_2D, tex);
glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 256, 256, GL_BGRA, GL_UNSIGNED_BYTE, 0);
```

Graphics Interop – Example



- **Frame Post-processing by CUDA:**

```
// OpenGL rendering code:  
// ...  
  
// CUDA post-processing code:  
unsigned char *d_buffer;  
cudaGLRegisterBufferObject(pbo) ;  
cudaGLMapBufferObject((void**) &d_buffer, pbo) ;  
post_process_kernel<<<...>>>(d_buffer) ;  
cudaGLUnmapBufferObject(pbo) ;  
cudaGLUnregisterBufferObject(pbo) ;
```


Graphics Interop – VBO example



```
GLuint vbo_pos;
glGenBuffers(1, &vbo_pos);
glBindBuffer(GL_ARRAY_BUFFER, &vbo_pos);
glBufferData(GL_ARRAY_BUFFER, N_POINTS*4*sizeof(float), 0, GL_DYNAMIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, 0);
cudaGLRegisterBufferObject(vbo_pos);

// LOOP:
float4 *d_pos;
cudaGLMapBufferObject((void**) &d_pos, vbo_pos);
move_points_kernel<<<N_POINTS/256, 256>>>(d_pos);
cudaGLUnmapBufferObject(vbo_pos);
glBindBuffer(GL_ARRAY_BUFFER, &vbo_pos);
glVertexPointer(4, GL_FLOAT, 0, 0);
glEnableClientState(GL_VERTEX_ARRAY);
glDrawArrays(GL_POINTS, 0, N_POINTS);
glDisableClientState(GL_VERTEX_ARRAY);
```

Host ↔ Device – Making it faster!



- **Group transfers**
 - One large transfer much better than many small ones
- **Use Page-locked memory...**

Host ↔ Device – Page-locked memory

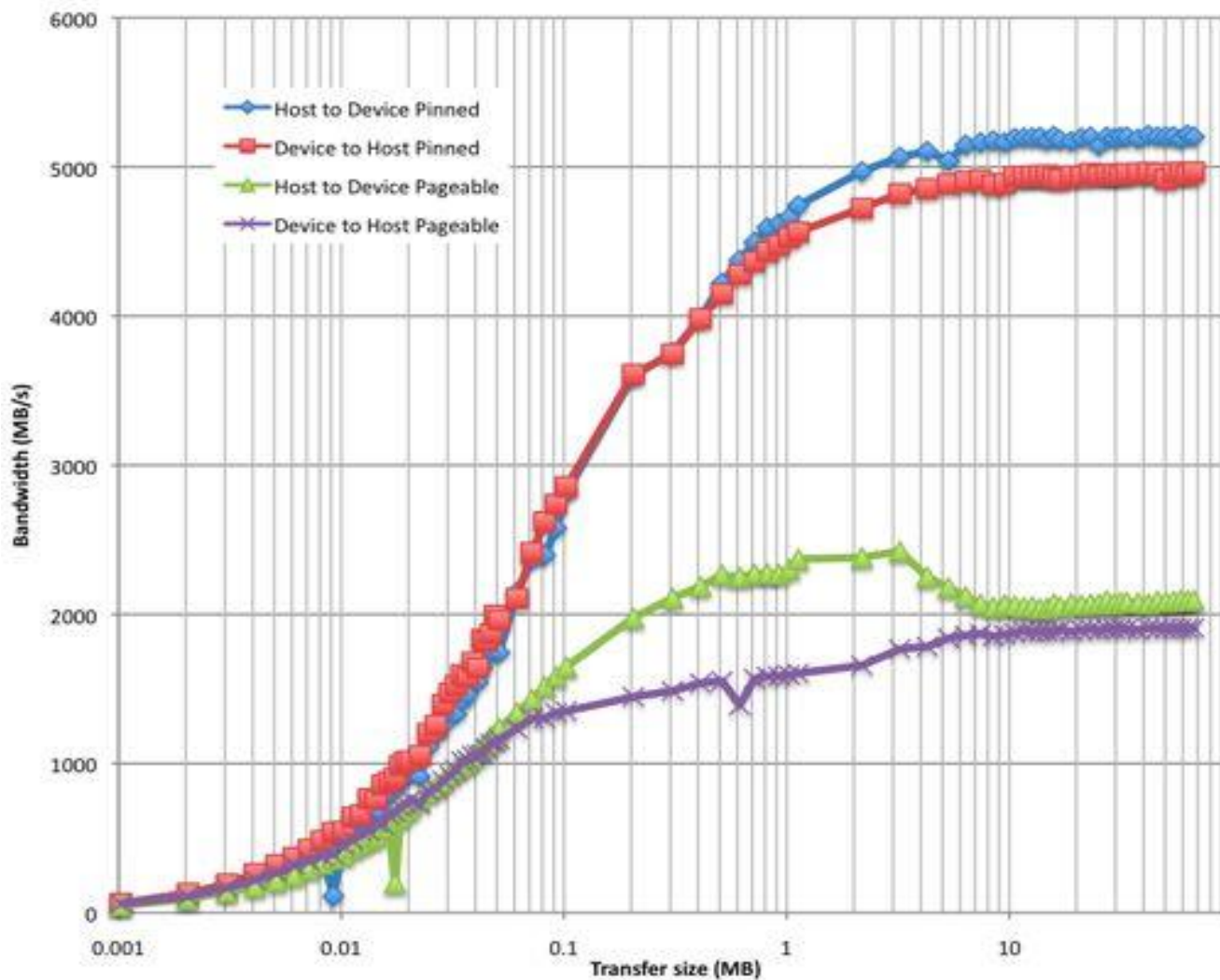


- Prevents OS from paging host memory
 - Allows PCI-e DMA to run at full speed
 - ≈ 3 GB/s (PCI-e x16 Gen1) or 6 GB/s (PCI-e x16 Gen2)
- **WARNING:**
 - Allocating too much page-locked memory can reduce system performance
- **CUDA-C:**
 - Instead of **malloc(...)**, use **cudaHostAlloc(...)**
- **OpenCL:**
 - Use **CL_MEM_ALLOC_HOST_PTR** in **clCreateBuffer**



See SDK!
bandwidthTest

Host ↔ Device – Performance



Host ↔ Device – Write-combining



- CUDA-C has option of Write-Combining
- memory is not snooped which can improve performance by up to 40%
- **WARNING:**
 - Not cached = FAST write, but SLOW host read

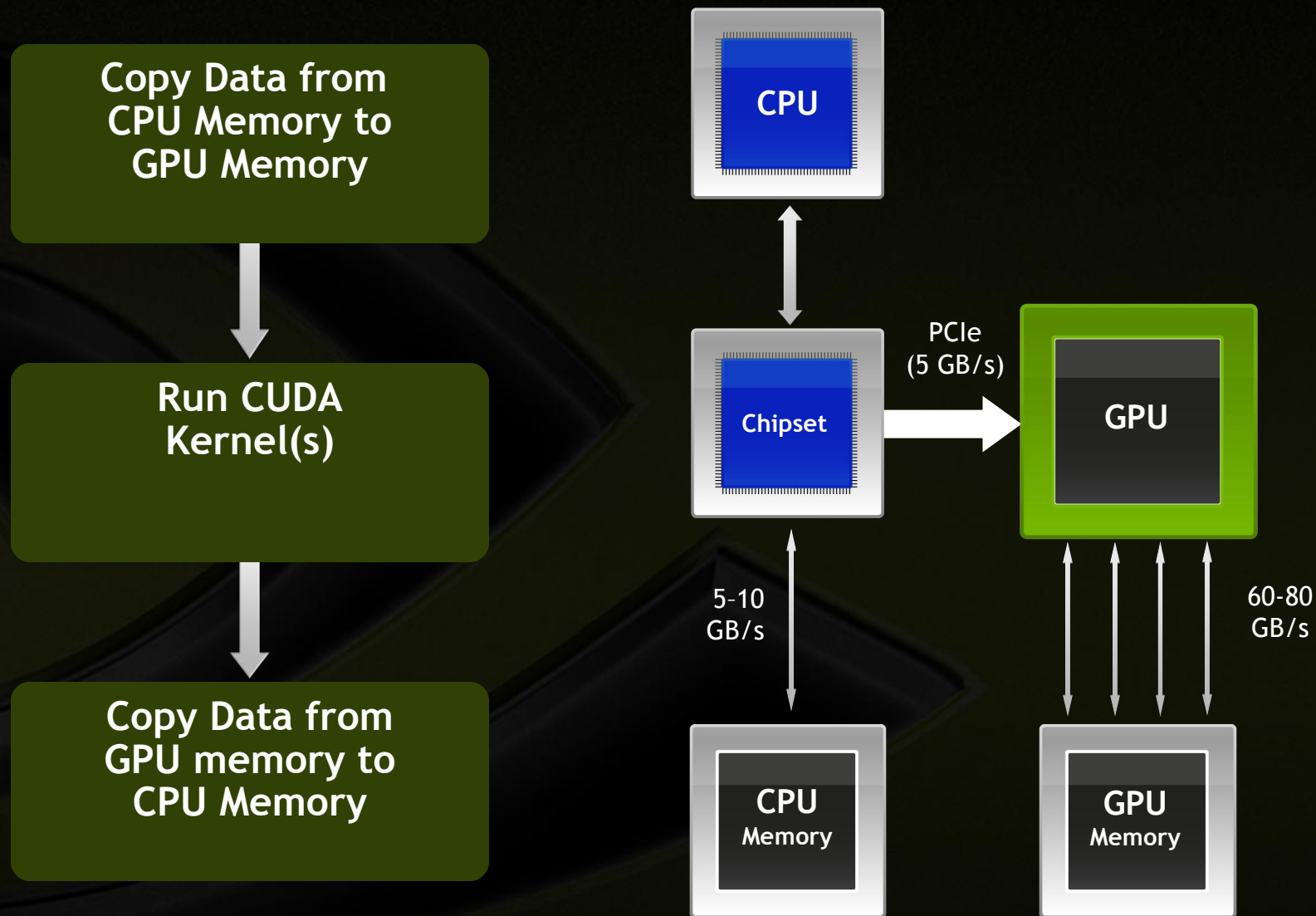
```
cudaHostAlloc((void**) &h_data,  
              num_bytes,  
              cudaHostAllocWriteCombined  
              );
```

Host ↔ Device – Hiding it



- **Asynchronous API**
- **Data Acquisition example**
- **CUDA Streams**
- **Zero Copy**

Typical Approach



Synchronous Functions



- Standard CUDA C functions are **Synchronous**
 - Trade-off between CPU cycles and response speed
 - `cudaDeviceSetFlags(...)`
 - `cudaDeviceScheduleSpin,`
`cudaDeviceScheduleYield,`
`cudaDeviceBlockingSync`
- Runtime API: Kernel launches are **Asynchronous**
- Synchronous functions block on any prior asynchronous kernel launches

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

Asynchronous API




- **All memory operations can also be asynchronous, and return immediately**
- **Copies & Kernels are queued up in the GPU**
- **Any launch overhead is overlapped**

Asynchronous API – Caveats



- **Memory must be allocated as page-locked using `cudaHostAlloc()`**
- **Synchronous calls should be done outside critical sections — some of these are expensive!**
 - Initialization
 - Memory allocations
 - Stream / Event creation
 - Interop resource registration



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

Asynchronous API – 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(...) ; ←----- Returns immediately  
myKernel<<<grid,block>>>(...) ; ←----- Returns immediately  
cudaMemcpyAsync(...) ; ←----- Returns immediately
```

```
// cpu does work here...
```

```
cudaThreadSynchronize() ; ←----- Waits for everything on the  
GPU to finish, then returns
```

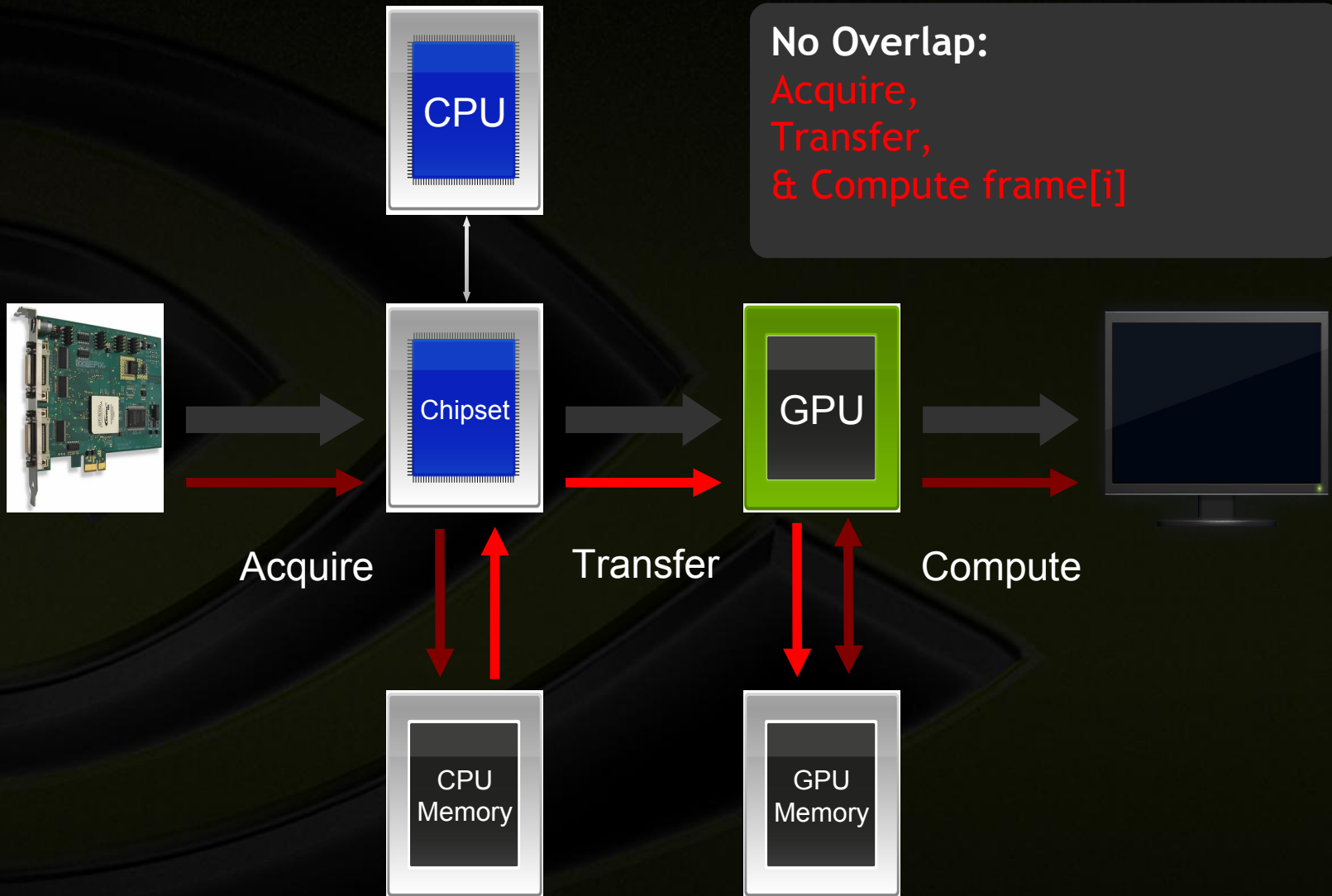
Using Events – Example

```
cudaEvent_t HtoDdone;  
cudaEventCreate (&HtoDdone, 0);  
cudaMemcpyAsync(d_dest, h_source, bytes, cudaMemcpyHostToDevice, 0);  
cudaEventRecord (HtoDdone); ← .....  
  
myKernel<<<grid, block>>> (...);  
  
cudaMemcpyAsync(d_dest, h_source, bytes, cudaMemcpyDeviceToHost, 0);  
  
// cpu can do stuff here  
  
cudaEventSynchronize (HtoDdone); ← .....  
  
// The first memory copy is done,  
// so the memory at source could be  
// used again by the CPU  
  
cudaThreadSynchronize (); ← .....  
.....
```

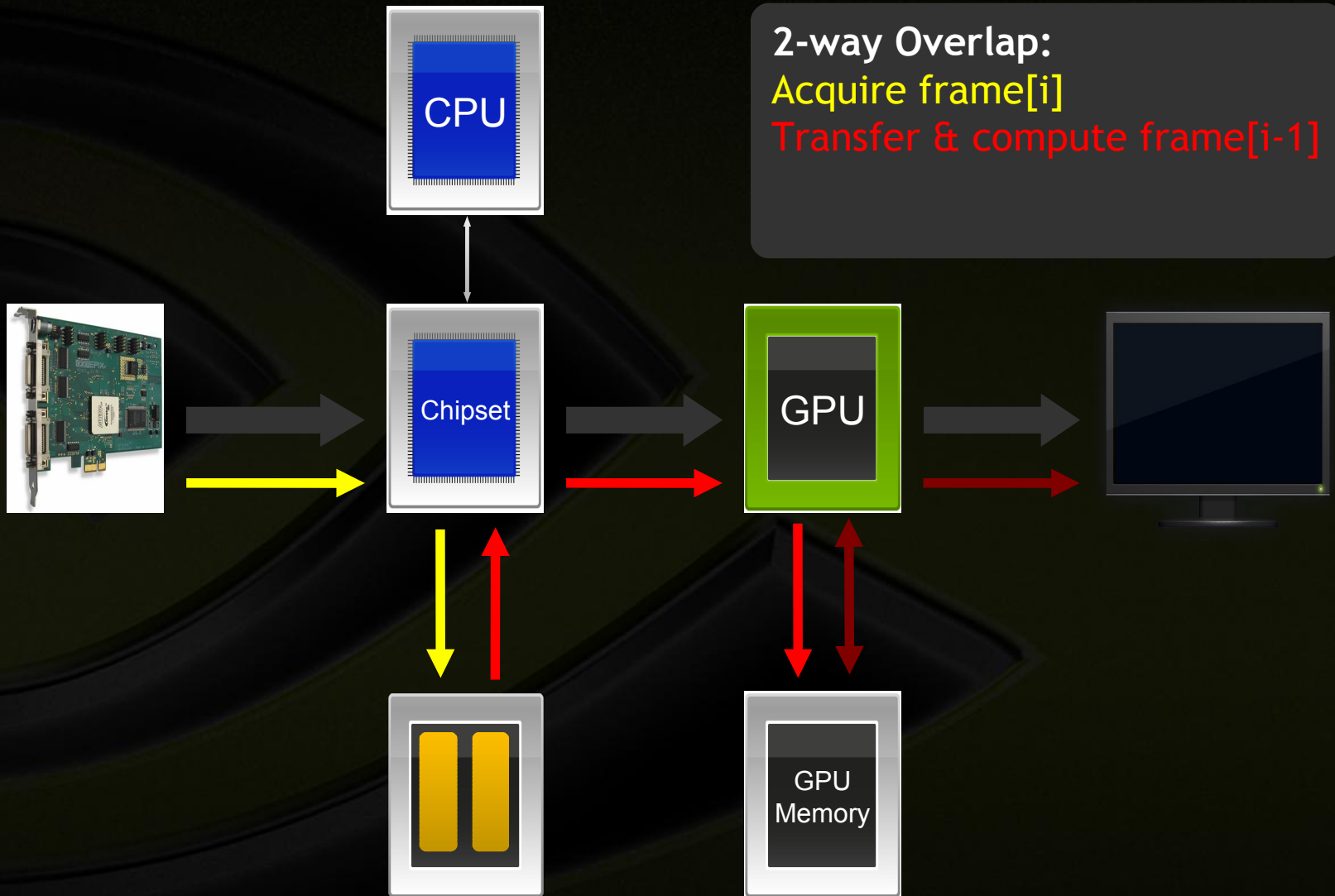
Waits just for everything before
`cudaEventRecord (HtoDdone)`
to complete, then returns

Waits for everything on the
GPU to finish, then returns

Acquiring Data From an Input Device



Overlap Acquisition With Transfer



Overlap Acquisition With Transfer



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

```
int buf = 0;
void* d_framebuf;
void* h_framebuf[2];
// Allocate buffers...

while (!done)
{
    cudaMemcpyAsync(d_framebuf, h_framebuf[(buf+1)%2], size,
                   cudaMemcpyHostToDevice, 0);

    myKernel1<<<...>>>(d_framebuf);
    // ... other GPU stuff, all asynchronous

    AcquireFrame(h_framebuf[buf]);
    // ... other CPU stuff

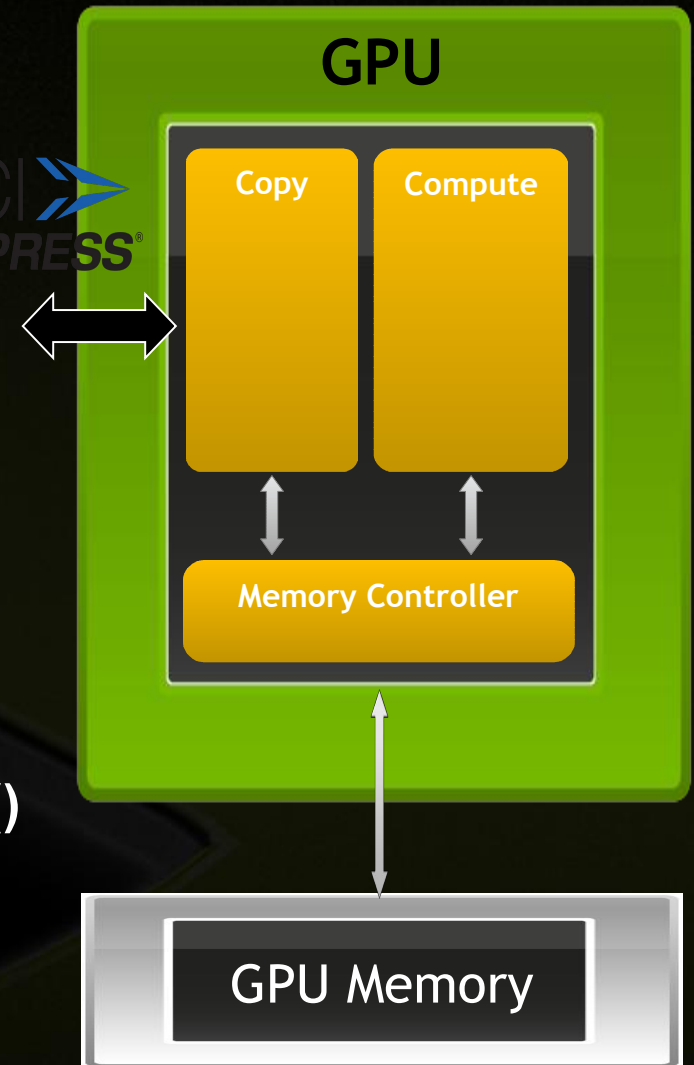
    cudaThreadSynchronize();
    buf++; buf%=2;
}
```

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, cudaStreamCreate()`
- **Multiple streams exist within a single context, they share memory and other resources**

PCI EXPRESS®



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

Stream Parameter



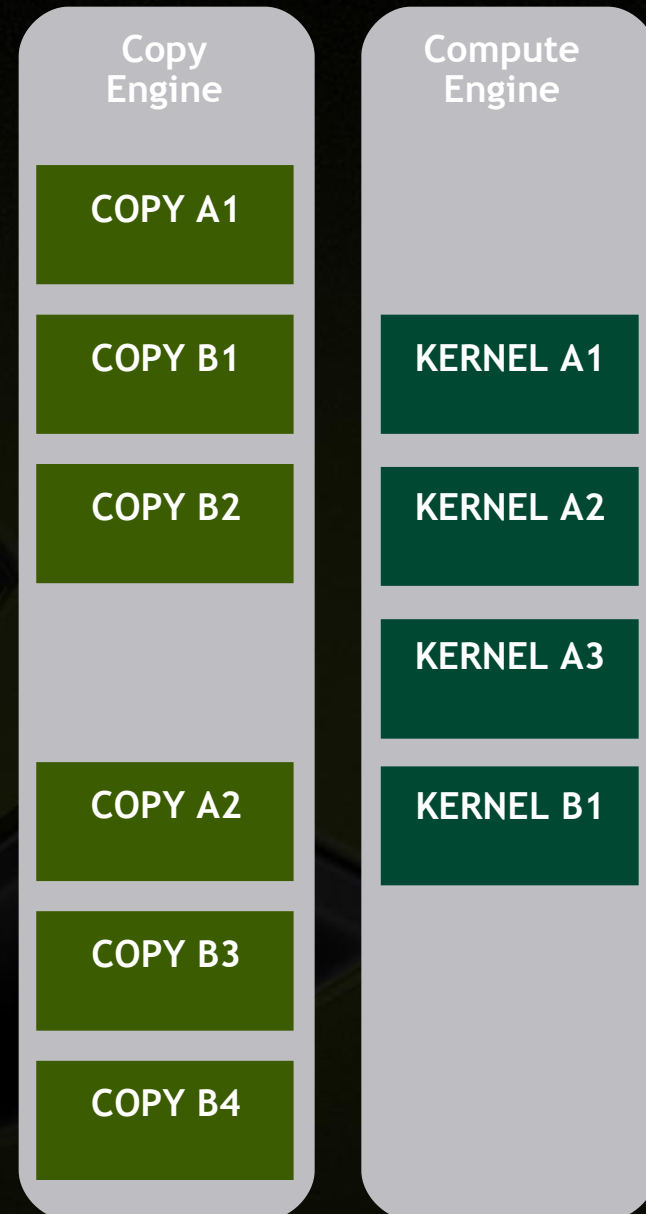
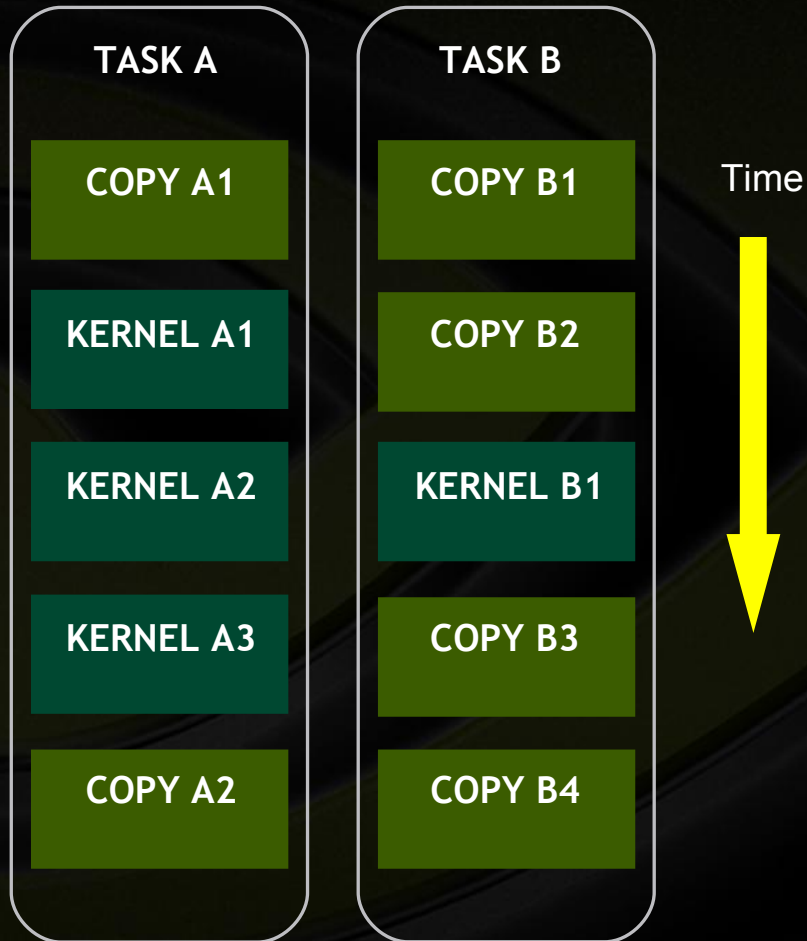
- All Async function varieties have a stream parameter
- Runtime Kernel Launch
 - <<<GridSize, BlockSize, SMEM Size, Stream>>>
- Copies & Kernel launches with the same stream parameter execute in-order

CUDA Streams

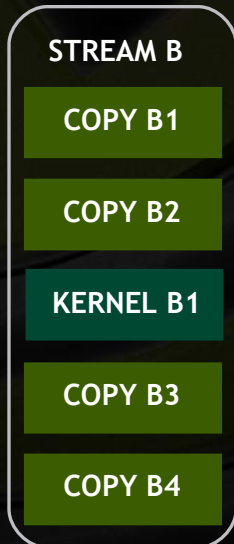
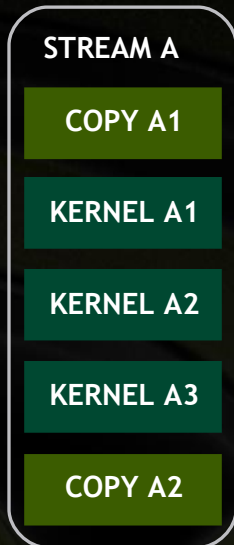
Scheduling on GPU:



Independent Tasks:



Avoid Serialization!



WRONG WAY!

```
CudaMemcpyAsync (A1..., StreamA) ;
```

```
KernelA1<<<..., StreamA>>> () ;
```

```
KernelA2<<<..., StreamA>>> () ;
```

```
KernelA3<<<..., StreamA>>> () ;
```

```
CudaMemcpyAsync (A2..., StreamA) ;
```

```
CudaMemcpyAsync (B1..., StreamB) ;
```

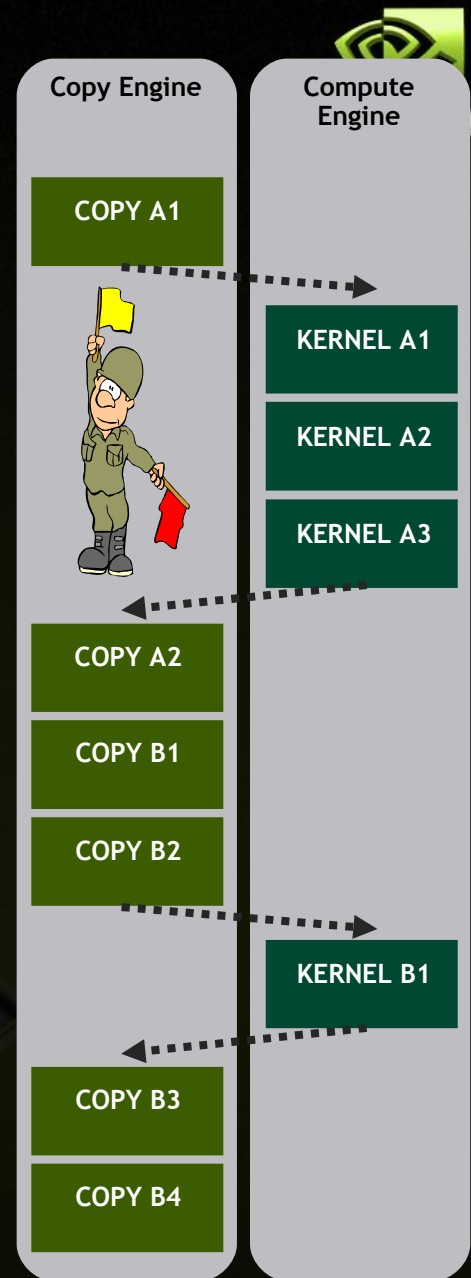
```
CudaMemcpyAsync (B2..., StreamB) ;
```

```
KernelB1<<<..., StreamB>>> () ;
```

```
CudaMemcpyAsync (B3..., StreamB) ;
```

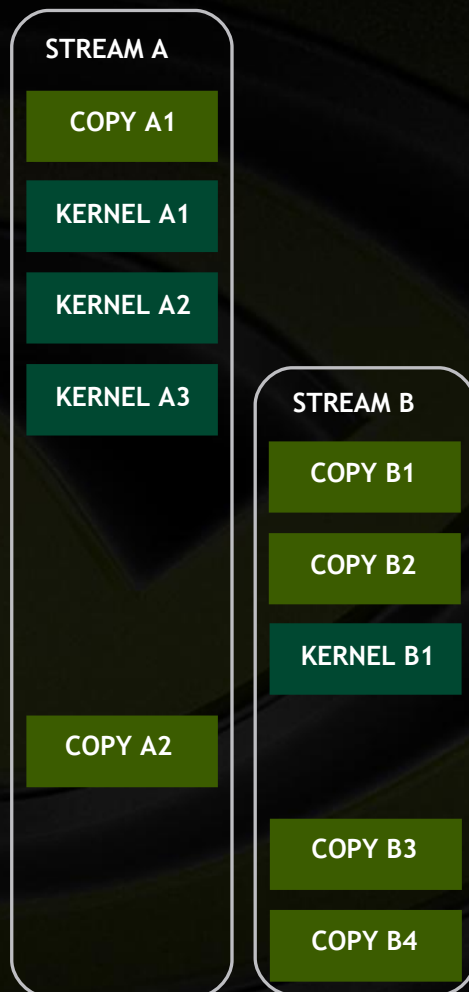
```
CudaMemcpyAsync (B4..., StreamB) ;
```

- Engine queues are filled in the order code is executed



Stream Code Order

CORRECT WAY!



```
CudaMemcpyAsync (A1..., StreamA) ;
```

```
KernelA1<<<..., StreamA>>>() ;
```

```
KernelA2<<<..., StreamA>>>() ;
```

```
KernelA3<<<..., StreamA>>>() ;
```

```
CudaMemcpyAsync (B1..., StreamB) ;
```

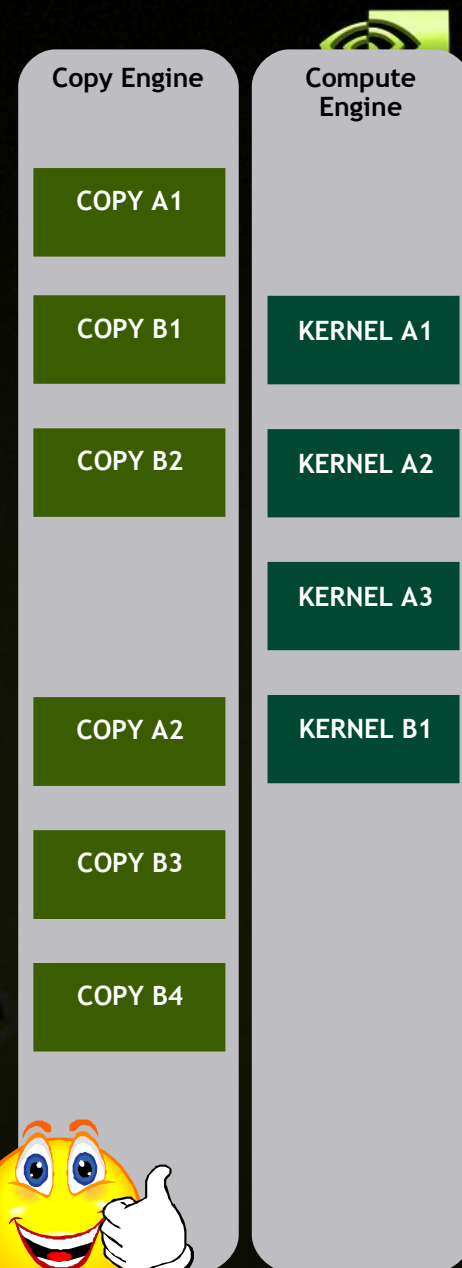
```
CudaMemcpyAsync (B2..., StreamB) ;
```

```
KernelB1<<<..., StreamB>>>() ;
```

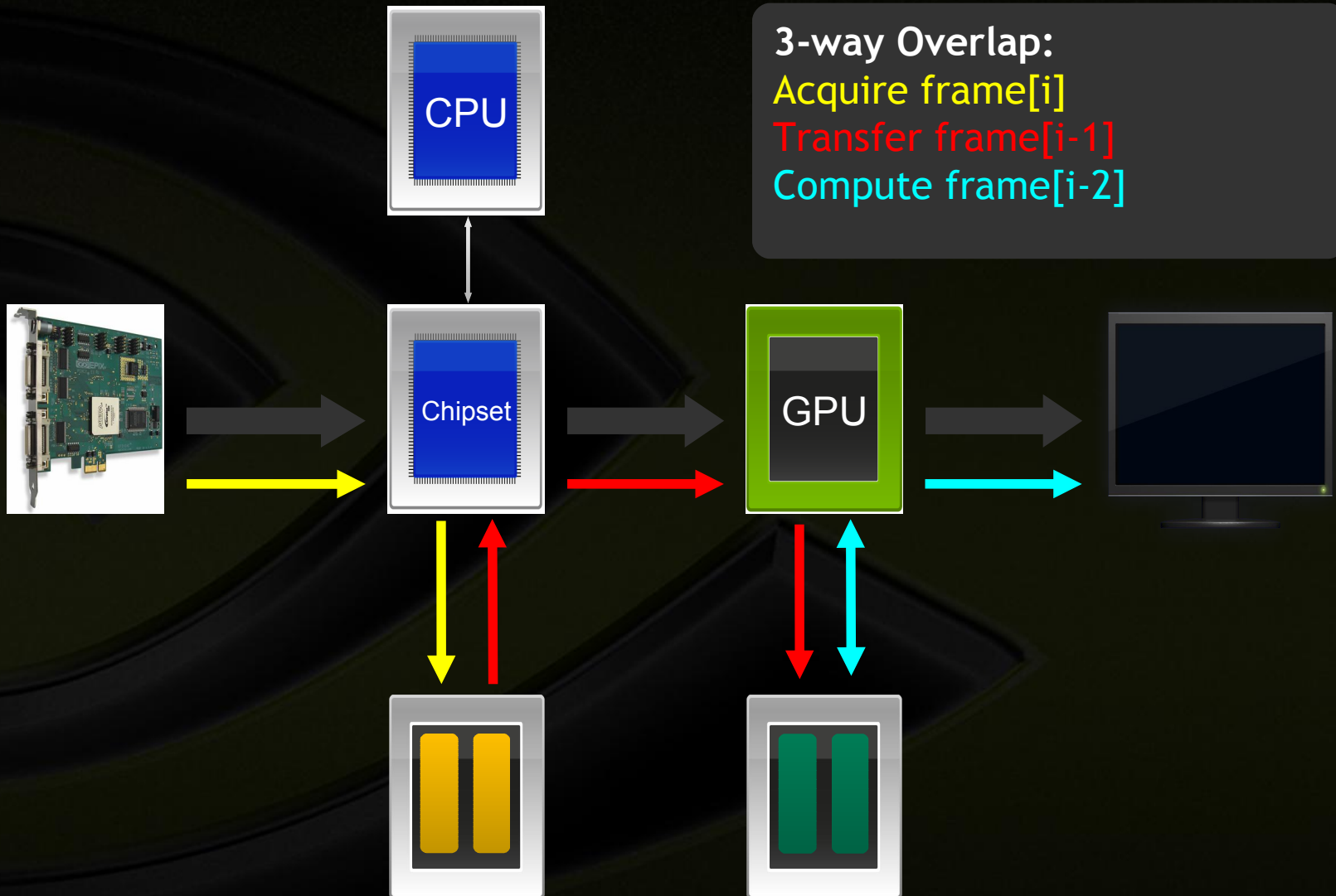
```
CudaMemcpyAsync (A2..., StreamA) ;
```

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

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



Revisit Our Data I/O Example



3-Way Overlap



- As before, allocate two host buffers
- Also allocate two device buffers

```
int buf = 0; // current buffer
void* h_framebuf[2];
void* d_framebuf[2];
cudaStream_t copyStream; // stream for copy
cudaStream_t compStream; // stream for compute

// Allocate Buffers
cudaHostAlloc(&(h_framebuf[0]), size, 0);
cudaHostAlloc(&(h_framebuf[1]), size, 0);

cudaMalloc(&(d_framebuf[0]), size, 0);
cudaMalloc(&(d_framebuf[1]), size, 0);

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

3-Way Overlap (Cont.)



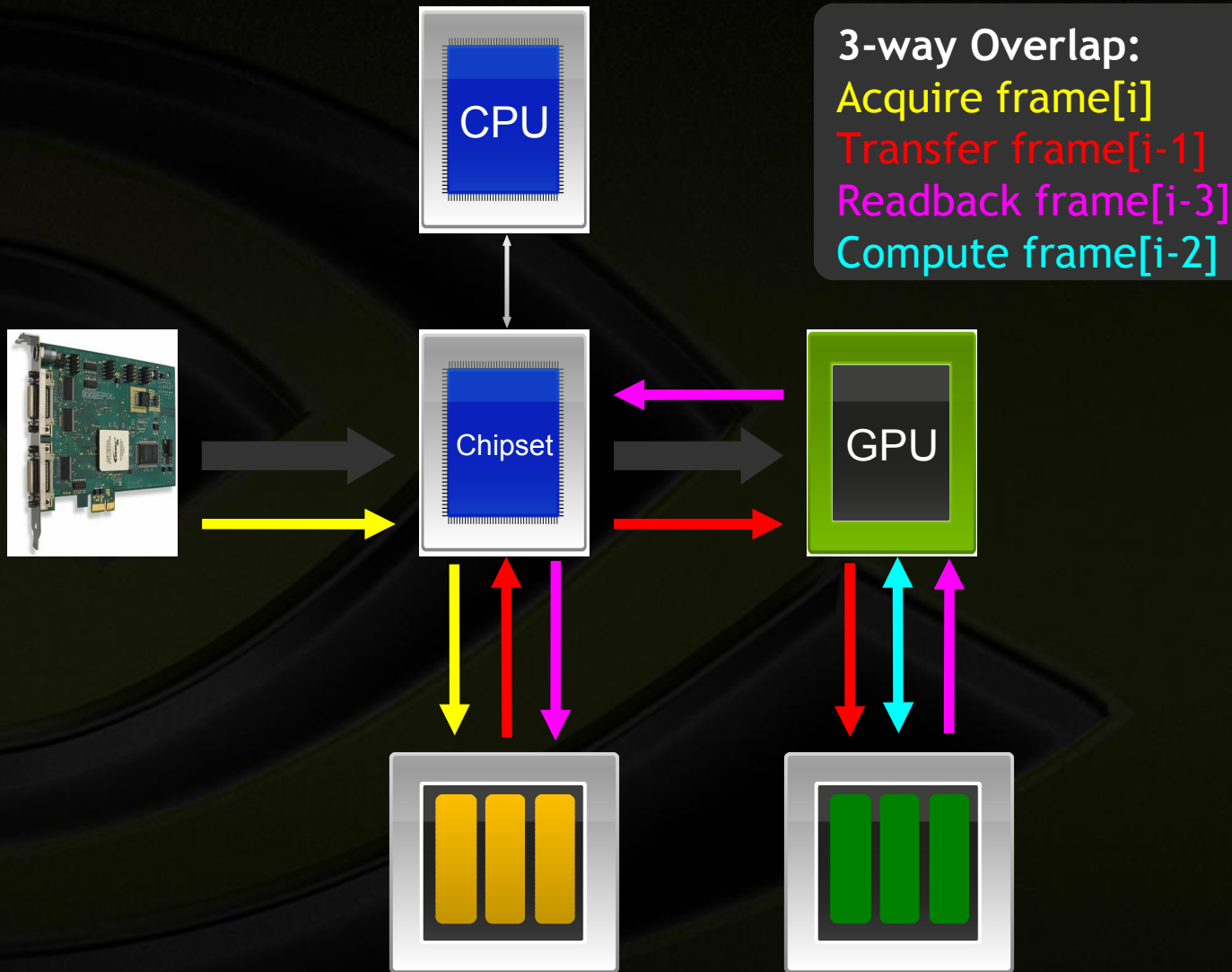
```
while (!done)
{
    cudaMemcpyAsync(d_framebuf[buf],
                   h_framebuf[(buf+1)%2], size,
                   cudaMemcpyHostToDevice,
                   copyStream);

    myKernel1<<<grid,block,0,compStream>>>(d_framebuf[(buf+1)%2]);
    myKernel2<<<grid,block,0,compStream>>>(d_framebuf[(buf+1)%2]);
    // ... other GPU stuff, all asynchronous

    AcquireFrame(h_framebuf[buf]);
    // ... other CPU stuff

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

What About Readback?



Readback



```
while (!done)
{
    cudaMemcpyAsync(d_framebuf[buf], h_framebuf[(buf+1)%3], size,
                   cudaMemcpyHostToDevice, copyStream);

    cudaMemcpyAsync(d_framebuf[buf+2], h_framebuf[(buf+2)%3], size,
                   cudaMemcpyDeviceToHost, copyStream);

    kernel1<<<grid,block,0,compStream>>>(d_framebuf[(buf+1)%3]...);
    kernel2<<<grid,block,0,compStream>>>(d_framebuf[(buf+1)%3]...);
    // ... other GPU stuff, all asynchronous

    AcquireFrame(h_framebuf[buf]);
    // ... other CPU stuff

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

4-Way Overlap?



- **FUTURE hardware adds a 2nd copy engine!**
 - Simultaneous upload and downloading
 - Simply add another stream
 - still works with prior hardware, just serialized

```
...  
cudaMemcpyAsync(d_framebuf[buf], h_framebuf[(buf+1)%3], size,  
                cudaMemcpyHostToDevice, uploadStream);
```

```
cudaMemcpyAsync(d_framebuf[buf+2], h_framebuf[(buf+2)%3], size,  
                cudaMemcpyDeviceToHost, downloadStream);  
...
```

Host Memory Mapping – Zero-Copy



- The easy way to achieve copy/compute overlap!
- Access host memory directly from device code
 - Transfers implicitly performed as needed by device code
 - Introduced in CUDA 2.2
 - Check **canMapHostMemory** field of **cudaDeviceProp** variable
- All setup is done on host using mapped memory

```
cudaSetDeviceFlags(cudaDeviceMapHost);  
...  
cudaHostAlloc((void**) &a_h, nBytes, cudaHostAllocMapped);  
cudaHostGetDevicePointer((void**) &a_d, (void *)a_h, 0);  
for (i=0; i<N; i++)  
    a_h[i] = i;  
increment<<<grid, block>>>>(a_d, N);
```



See SDK
samples!

Zero Copy guidelines



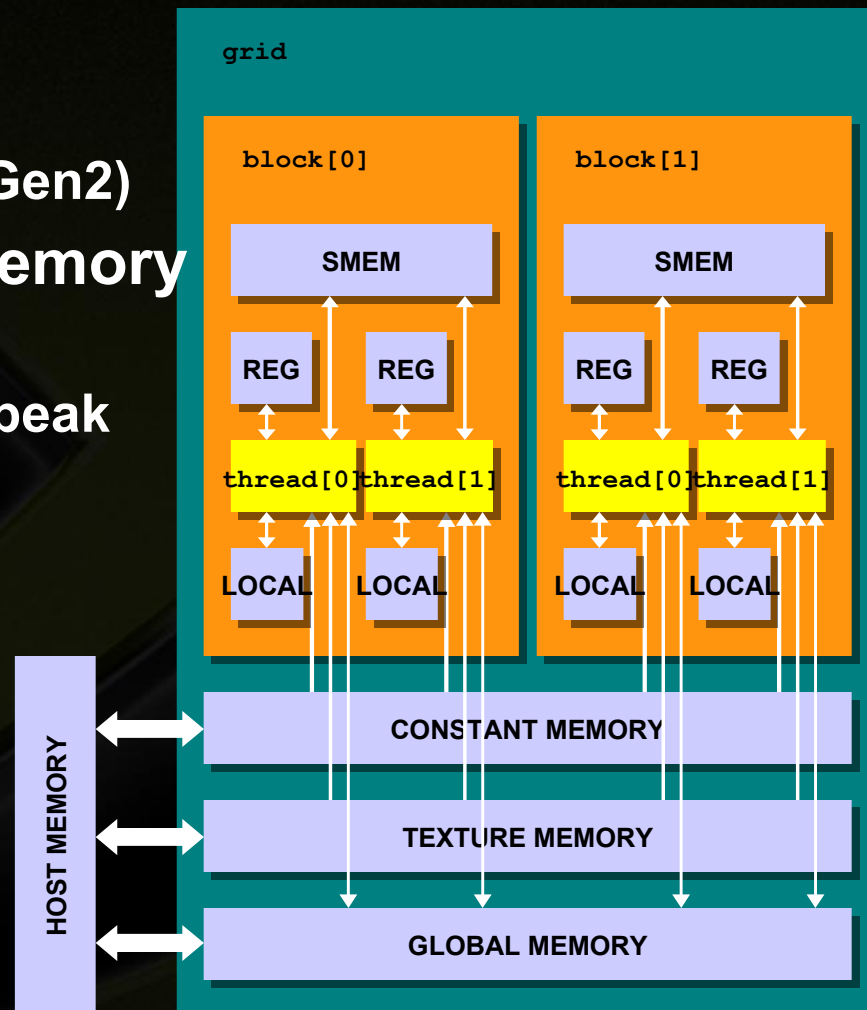
- **Easier and faster alternative to using Async API**
- **Data is transferred over the PCIe bus automatically, but it's slow**
 - Use when data is only read/written once
 - Use for small amounts of data (new variables, CPU/GPU communication)
 - Use when compute/memory ratio is very high and occupancy is high, so latency over PCIe is hidden
 - Coalescing is critically important
- **Zero copy will be a win for integrated devices**
 - you can check this using the **integrated** property in **cudaDeviceProp**

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

GPU Memory architecture



- **Host memory**
 - 6 GB/s peak (PCIe x16 Gen2)
- **Global / Local device memory**
 - 4GB
 - high latency, 141 GB/s peak
- **Constant memory**
 - 64 KB read-only
 - cached
- **Texture memory**
 - read-only
 - spatially cached
- **Shared memory**



Hierarchical thread structure



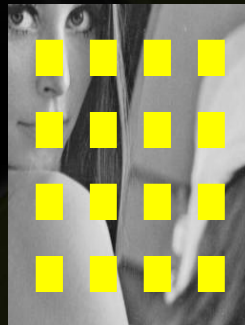
Individual **THREADS** operate on data elements.



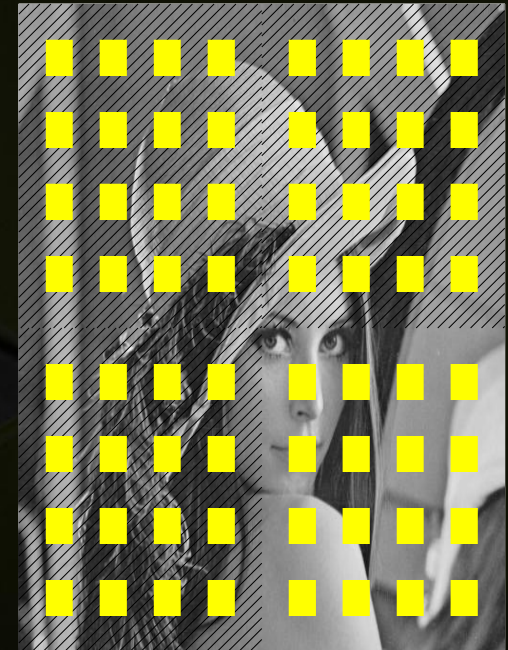
The unit of parallelism.

Negligible cost for creation, switching, and overhead.

Threads are grouped into **BLOCKS**, which can synchronize and cooperate.



A **GRID** contains multiple blocks and covers the entire data set.



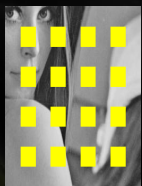
Execution Model



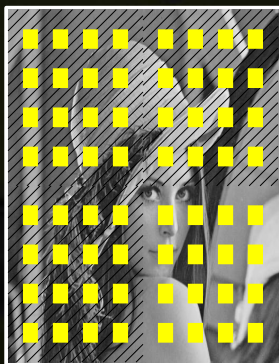
Programming model:



THREAD



BLOCK



GRID

Hardware:



Scalar Processor



Streaming Multiprocessor



Device

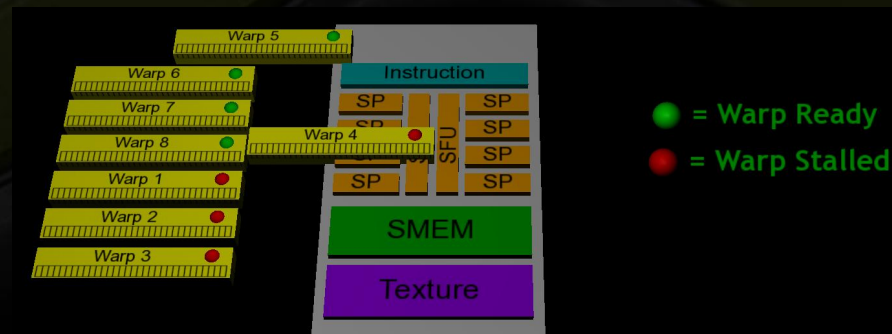
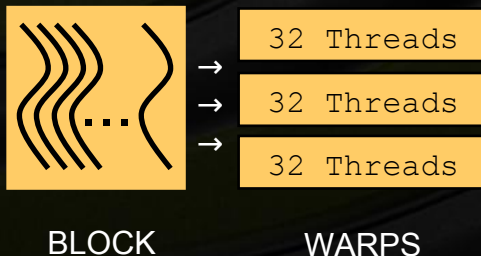
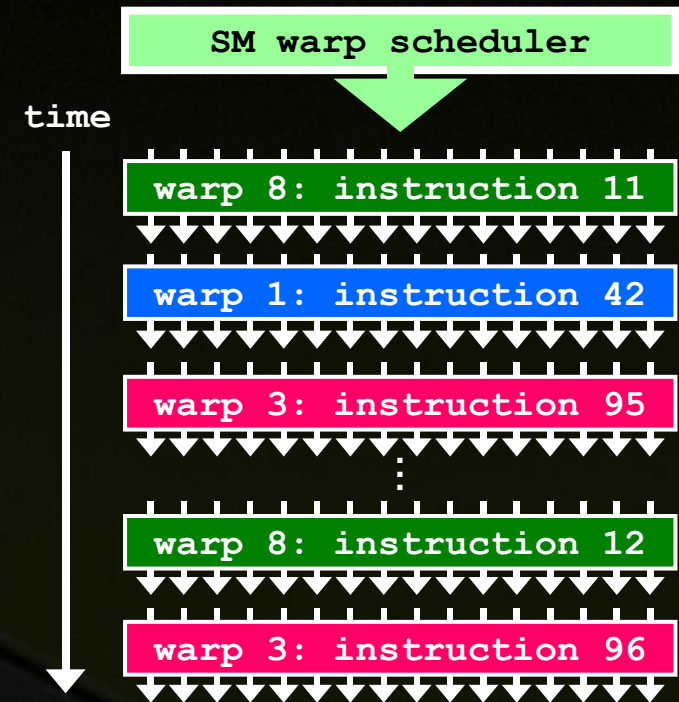
- Threads are executed by scalar processors
- BLOCKS are executed on multiprocessors
 - BLOCKS do not migrate
 - Several concurrent BLOCKS can reside on one SM.
 - This is limited by SM resources
- A kernel is launched as a GRID of BLOCKS
 - Only one kernel can execute on a device at one time

Warps



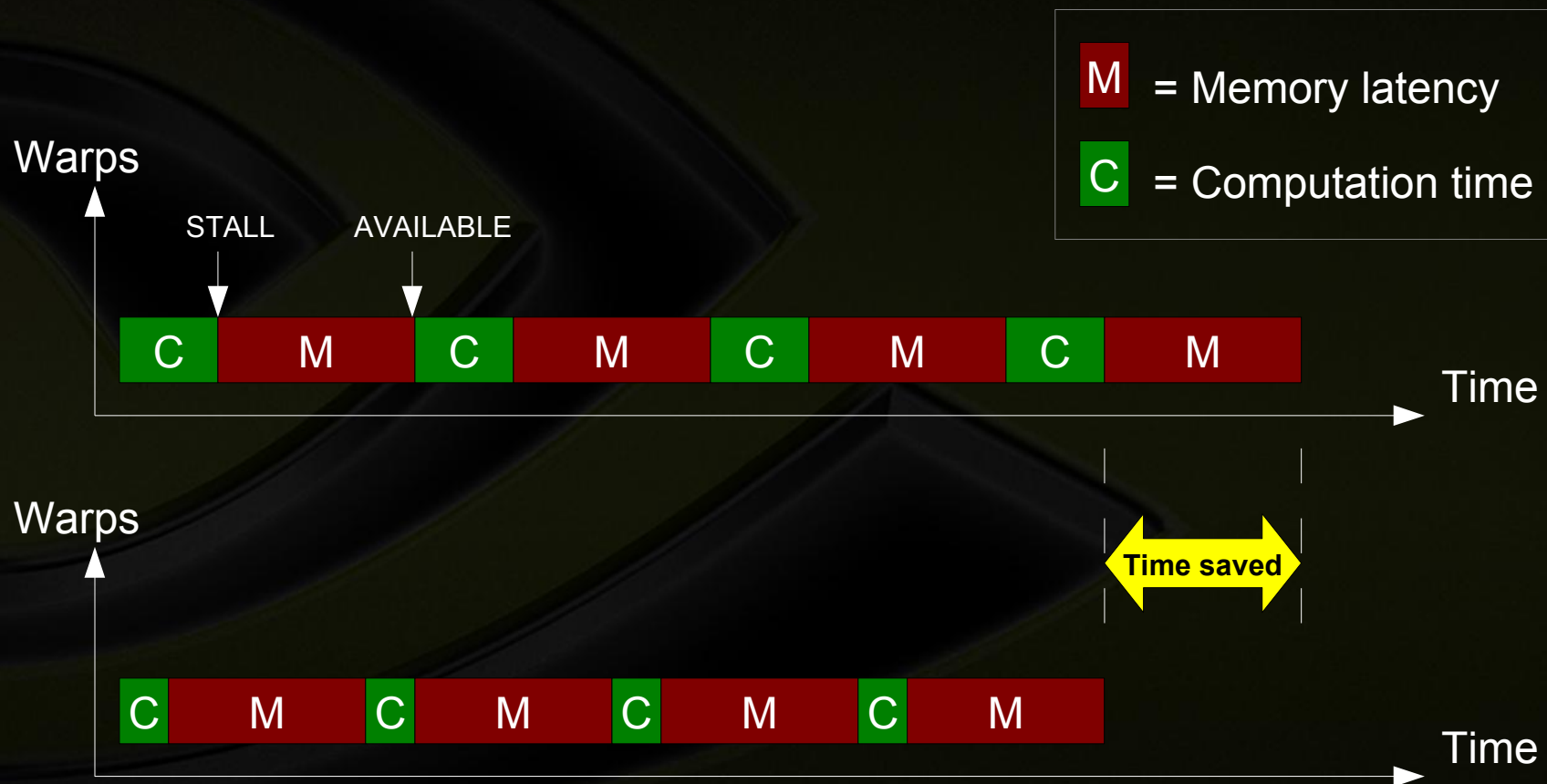
- **BLOCKS** divide into groups of threads called **WARPS**

- The unit of scheduling
- All threads in warp perform same instruction (SIMT)
- Using many warps can hide memory latency
- **warpSize** = 32 threads



Latency hiding – single-threaded

- The time saved from maths performance increase is small because memory latency is the limiting factor

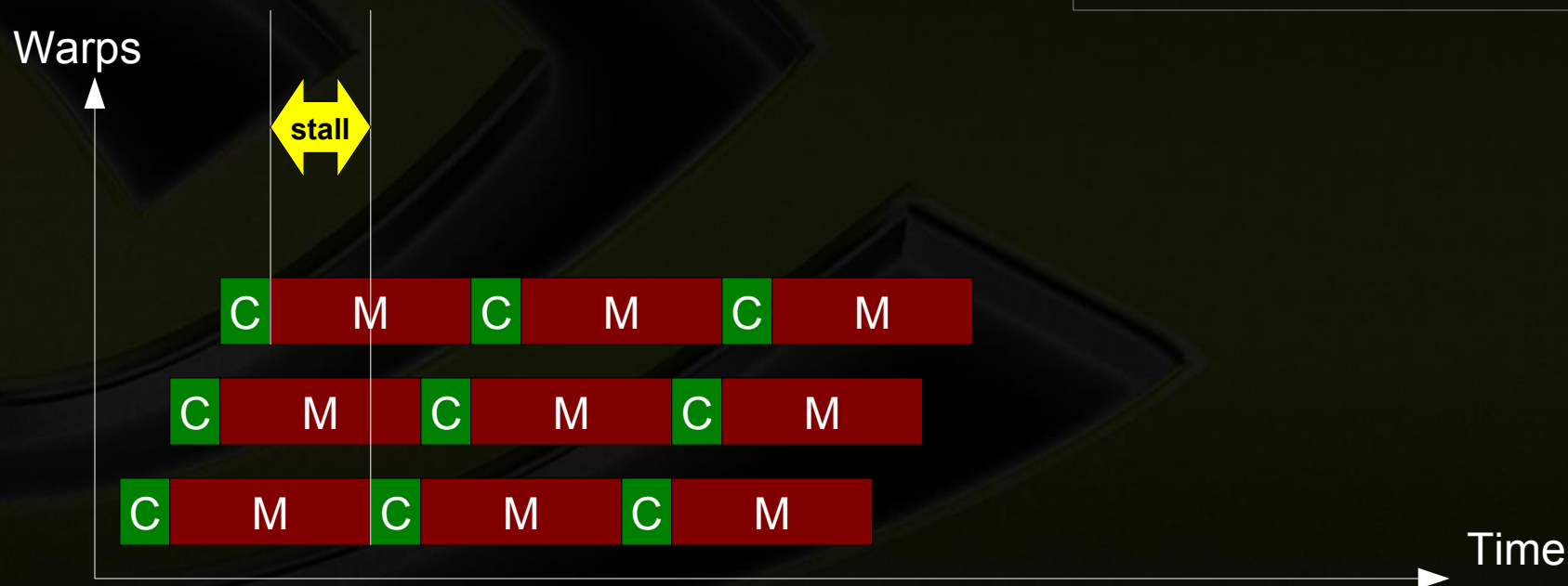


Latency hiding – multi-threaded



- We must try to ensure that the processor is always doing work

M = Memory latency
C = Computation time

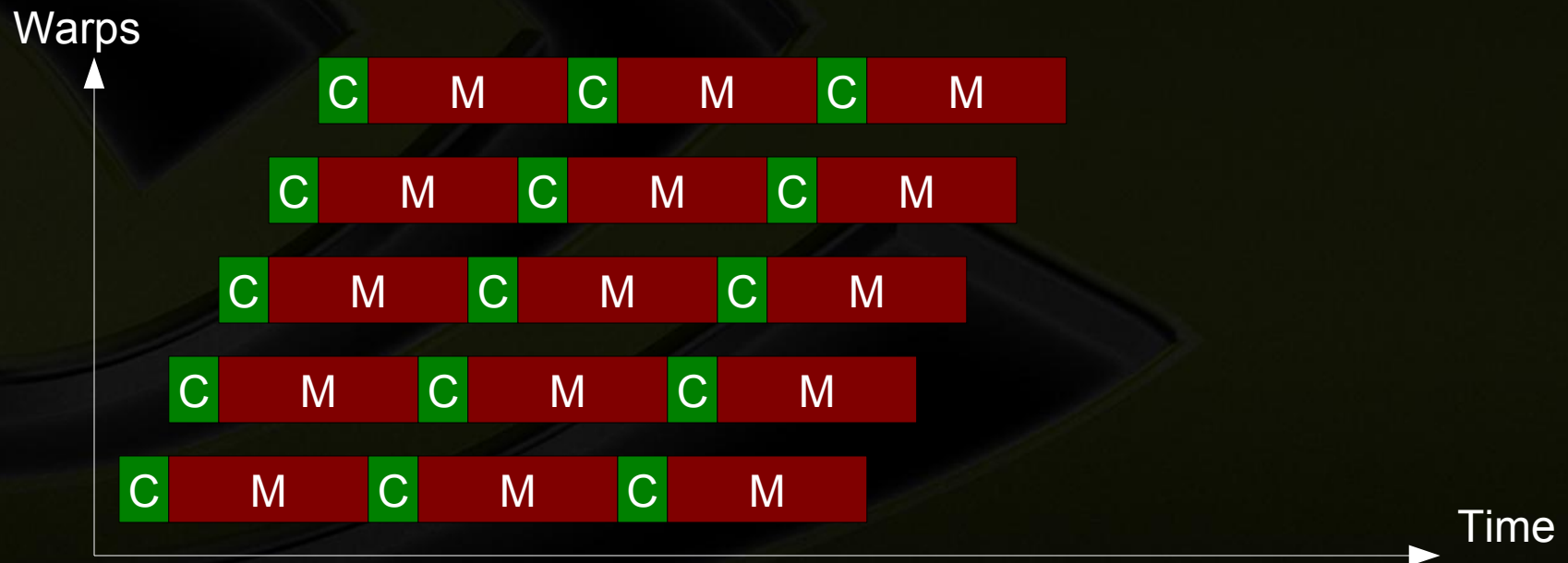


Latency hiding – multi-threaded



- To hide latency we can increase the amount of warps

M = Memory latency
C = Computation time



Latency hiding – Example



- **Instructions are executed sequentially, so executing other warps is the only way to hide latencies and keep the hardware busy**
- **How many warps to hide global memory access?**
 - **We need 100 ($400/4$) arithmetic instructions to hide the latency**
 - **e.g. Assume the code has 8 instructions ($8*4$ cycles) for each global memory access (~ 400 cycles)**
 - **$100 / 8 \approx 13$ warps**

Latency hiding – Example



- Read-after-write register dependency
 - Instruction's result can be read ~24 cycles later

KERNEL CODE:

```
x = y + 5;  
z = x + 3;
```

```
s_data[0] += 3;
```

```
add.f32  
add.f32
```

```
ld.shared.f32  
add.f32
```

PTX CODE:

```
$f3, $f1, $f2  
$f5, $f3, $f4
```

```
$f3, [$r31+0]  
$f3, $f3, $f4
```

- To completely hide the latency:
 - We need at least 6 warps (24 / 4) per multiprocessor

Occupancy



- **Occupancy =**
 - Number of warps running concurrently on a multiprocessor divided by hardware-limit of max possible number of simultaneous warps
- **Max warps = 32**
 - (24 on older hardware, CC \leq 1.1)
- **To hide GMEM latency on CC 1.2, we need at least:**
 - $13 / 32 = 40\%$ occupancy
- **To hide register dependency on CC 1.1, we need:**
 - $6 / 24 = 18.75\%$ occupancy

Occupancy – Considerations



- **Increase occupancy to achieve latency hiding**
- **Occupancy is limited by SM resource usage:**
 - **Registers = 64KB = 16384**
 - (32K on older hardware = 8192 registers)
 - **Shared memory = 16KB**
 - **Scheduling hardware**
 - max running warps per SM = 32
 - max blocks per SM = 8

Occupancy – Register pressure



- Increase warps by running more threads per SM
 - Get as many threads (and blocks) able to run as possible
- Limiting Factors:
 - Number of registers per kernel
 - 64KB per SM, partitioned among concurrent threads
 - Amount of shared memory
 - 16KB per SM, partitioned among concurrent blocks
 - kernel parameters go in Shared Memory – consider using constant memory instead

R = registers required by kernel

R_{\max} = maximum registers per SM (16384)

actual required registers = $\text{ceil}(R * \text{ceil}(\text{BLOCK_SIZE}, 32), R_{\max} / 32)$

Occupancy – Resource limit example



- **SM partitions registers and local memory for all active blocks:**
 - If every thread uses 10 registers and every block has 256 threads:
 - Each block uses $256 \times 10 = 2560$ registers.
 - $8192 / 2560 = 3.2 \rightarrow$ **3 blocks**
 - $(256 \times 3 = 768) / 32 \rightarrow$ **24 warps can run**
 - **$(24 / 24) \rightarrow 100\%$ occupancy can be achieved**
 - However, if every thread uses 17 registers:
 - $8192 / (256 \times 17) = 1.9 \rightarrow$ **1 block**
 - $(256 \times 1 = 256) / 32 \rightarrow$ **8 warps can run**
 - **So occupancy is reduced to $(8 / 24 =)$ 33%**
 - But, if block has 128 threads:
 - since $8192 / (128 \times 17 = 2176) = 3.8 \rightarrow$ **3 blocks (of 128 threads)**
 - **occupancy can be $((384/32) / 24) \rightarrow 50\%$**

Determining resource usage



- Compile the kernel with the `-cubin` flag
 - Open the `.cubin` file with a text editor:

```
architecture {sm_10}
abiversion {0}
modname {cubin}
code {
    name = MyKernel
    lmem = 0
    smem = 68
    reg = 20
    bar = 0
    bincode {
        0xa0004205 0x04200780 0x40024c09 0x00200780
        ...
    }
}
```

per thread local memory

per thread block shared memory

per thread registers

- Or compile with `-ptxas-options=-v`

PTX – GPU assembly



- Compile with `-keep` or `-ptx`
- Interleaved code: `--openccl-options -LIST:source=on`
- Useful to check

```
1264 st.shared.u32 [%r63+12], %r78; // id:493 smemf+0x0
1265 @!%p1 bra $Lt_4_77; //
1266 add.u32 %r79, %r2, 1; //
1267 mul24.lo.u32 %r80, %r18, %r79; //
1268 add.u32 %r81, %r8, %r80; //
1269 cvt.rn.f32.u32 %f53, %r81; //
1270 mov.f32 %f54, 0fc0000000; // -2
1271 add.f32 %f55, %f53, %f54; //
1272 mov.f32 %f56, %f36; //
1273 mov.f32 %f57, 0f00000000; // 0
1274 mov.f32 %f58, 0f00000000; // 0
1275 tex.2d.v4.u32.f32 (%r82,%r83,%r84,%r85),[normFloatTex,{%f55,%f56,%f57,%f58}];
1276 .loc 2 556 0
1277 // 552 if(threadIdx.x < 4)
1278 // 553 {
1279 // 554
1280 // 555 sidx = __umul24(blockDim.y+threadIdx.y,smem_pitch) + blockDim.x + threadIdx.x;
1281 // 556 smemf[sidx] = tex2D(normFloatTex,(float)(__umul24(blockDim.x,blockIdx.x+1)+threadIdx.x)-2.0f, Ytex);
1282 mov.s32 %r86, %r82; //
1283 add.u32 %r87, %r18, %r60; //
1284 add.u32 %r88, %r8, %r87; //
1285 mul.lo.u32 %r89, %r88, 4; //
1286 add.u32 %r90, %r1, %r89; //
1287 st.shared.u32 [%r90+0], %r86; // id:494 smemf+0x0
1288 $Lt_4_77:
```


Minimizing register pressure

- To maximize occupancy compiler will minimize register usage
- Use compiler option: `-maxrregcount=<N>`
 - N = desired maximum registers / kernel
- **WARNING:**
 - At some point, “spilling” into Local memory may occur
 - LMEM is located in slow device memory
 - Large arrays & structures are stored in LMEM
 - Check .cubin file for LMEM usage
- By default `nvcc` forces *all* device code to be inline
 - Use `__noline__` function qualifier as compiler hint

Grid Size Heuristics



- **# of blocks > # of multiprocessors**
 - So all multiprocessors have at least one block to execute
- **# of blocks / # of multiprocessors > 2**
 - Multiple blocks can run concurrently in a multiprocessor
 - Blocks that aren't waiting at a `__syncthreads()` keep the hardware busy
- **# of blocks > 100 to scale to future devices**
 - Blocks executed in pipeline fashion
 - 1000 blocks per grid will scale across multiple generations

Block Size Heuristics



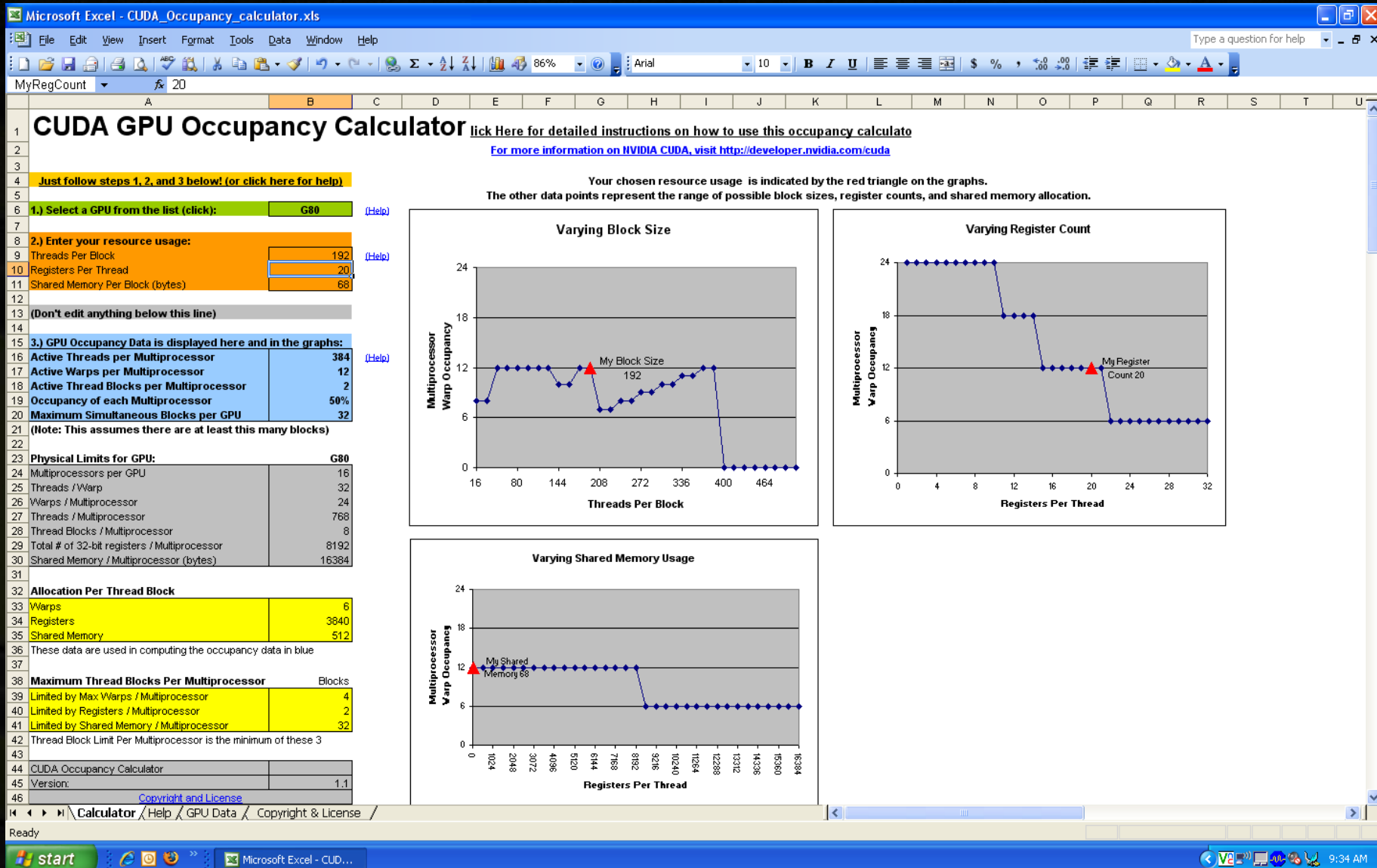
- **More threads per block = fewer registers per thread**
 - Kernel invocations can fail if too many registers are used
- **Use occupancy heuristic**
 - More threads per block = better memory latency hiding
- **Choose threads per block as a multiple of warp size**
 - Avoid wasting computation on under-populated warps
- **Help hardware thread scheduler minimize register bank conflicts**
 - Use multiple of 64 threads for best efficiency
- **Heuristics**
 - Minimum of 64 threads per block (allows 2 warps)
 - 192 or 256 threads is a better choice
 - Usually still enough registers to compile and invoke successfully

Occupancy – Conclusions



- **After some point (e.g. 50%), further increase in occupancy won't lead to performance increase**
- **So occupancy calculation in realistic case is complicated, thus...**

CUDA Occupancy Calculator



Execution Configuration – Summary



- **Use optimal number of threads per block**
 - **More warps per block, deeper pipeline**
 - hides latency, gives better SM occupancy
 - at least 192 hides read after write dependency
 - **Limited by available resources**
- **Maximize concurrent blocks on SM**
 - **Multiple blocks keep SM busy when waiting for synchronization**
 - **Can be a trade-off for shared memory usage**
 - **Less than 8KB shared memory per block allows more than one block to run**

Occupancy != Performance



- Increasing occupancy does not necessarily increase performance

BUT...

- Low-occupancy multiprocessors cannot adequately hide latency on memory-bound kernels
 - It all comes down to arithmetic intensity and available parallelism

Optimize Memory Access – Outline



- **Optimize Global Memory access**
- **Using Shared Memory**
- **Using Texture & Constant Memory**

Global Memory



- **Global memory is not cached**
- **Highest latency instructions**
 - 400-600 clock cycles
 - Launching more threads can help hide this latency
- **Likely to be a bottleneck**
 - Optimizations can greatly increase performance
- **Important to minimize accesses**
 - Use 64 / 128-bit load/store instructions...
 - Coalesce global memory accesses...

Global Memory – Load & store



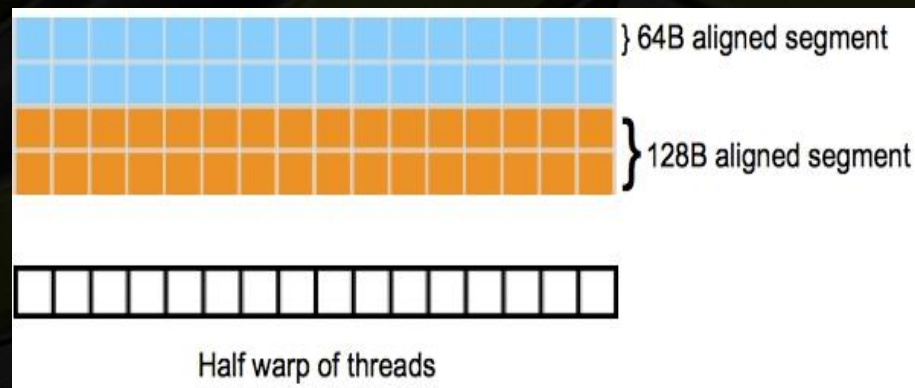
- Variables must have a size of 4, 8, or 16 bytes, and must be aligned to a multiple of their size
- Use -ptx flag of nvcc to inspect instructions:

4 byte load and store:	<code>ld.global.f32</code>	<code>\$f1, [\$rd4+0];</code>
	<code>st.global.f32</code>	<code>[\$rd4+0], \$f2;</code>
	...	
8 byte load and store:	<code>ld.global.v2.f32</code>	<code>{ \$f3, \$f5 }, [\$rd7+0];</code>
	<code>st.global.v2.f32</code>	<code>[\$rd7+0], { \$f4, \$f6 };</code>
	...	
16 byte load and store:	<code>ld.global.v4.f32</code>	<code>{ \$f7, \$f9, \$f11, \$f13 }, [\$rd10+0];</code>
	<code>st.global.v4.f32</code>	<code>[\$rd10+0], { \$f8, \$f10, \$f12, \$f14 };</code>

Coalescing (CC <= 1.1)



- **Coalescing occurs when a half warp (16 threads) accesses contiguous region of GMEM**
 - 16 data elements loaded in one instruction
 - int, float: 64 bytes (fastest)
 - int2, float2: 128 bytes
 - int4, float4: 256 bytes (2 transactions)
- **If un-coalesced, hardware issues 16 sequential loads**

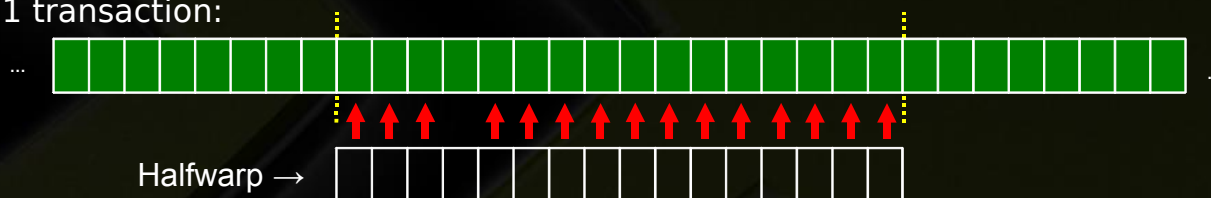


Coalescing in CC 1.0 and 1.1

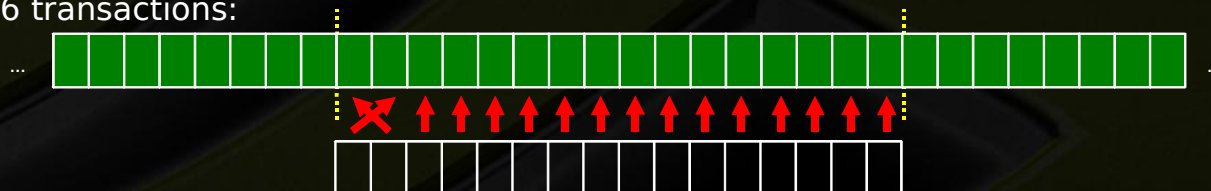


- k^{th} thread in halfwarp must access k^{th} word in segment
 - not all threads need to participate
 - Start address of region must be multiple of region size

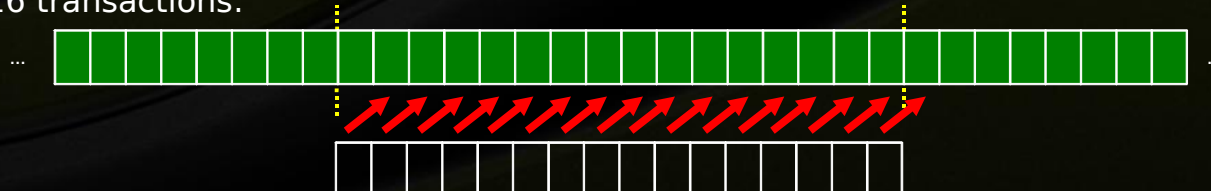
Coalesces - 1 transaction:



Permuted - 16 transactions:



Misaligned - 16 transactions:



Coalescing (CC \geq 1.2)



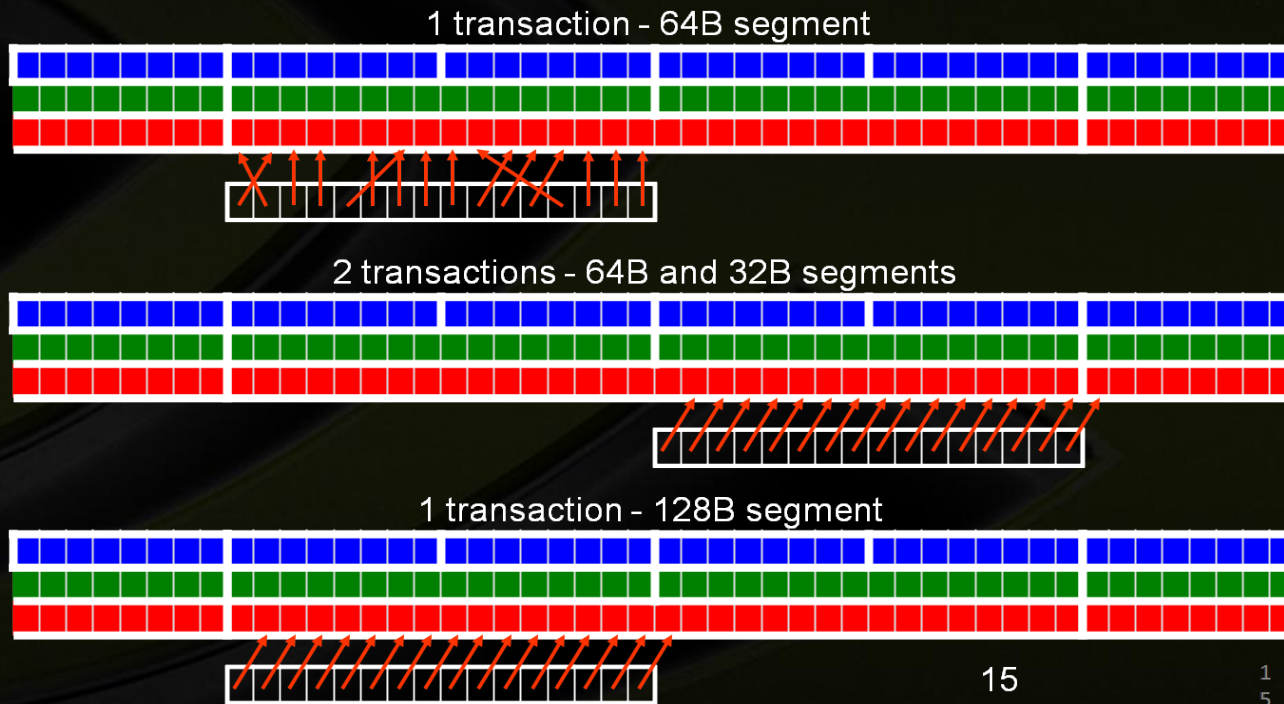
- **Much improved coalescing capabilities in 10-series architecture**
- **Hardware combines addresses within a half-warp into one or more aligned segments**
 - 32, 64, or 128 bytes
- **All threads with addresses within a segment are serviced with a single memory transaction**
 - Regardless of ordering or alignment within the segment

Coalescing in CC 1.2 & 1.3



- Any pattern of access that fits into an aligned segment size
- # of transactions = # of accessed segments

32-byte segment
64-byte segment
128-byte segment



15

1
5

Coalescing – Examples

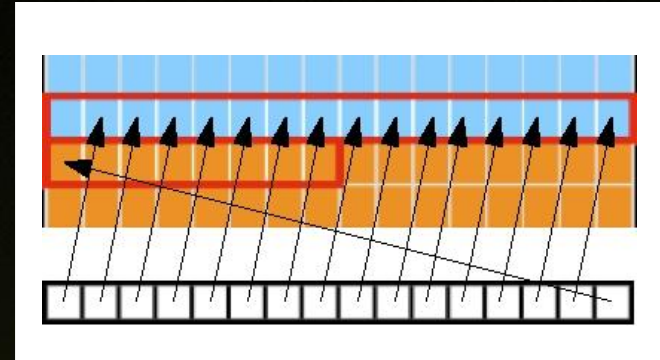


- **Effective bandwidth of small kernels that copy data**
 - **Effects of offset and stride on performance**
- **Two GPUs**
 - **GTX 280**
 - **Compute Capability 1.3**
 - **Peak bandwidth of 141 GB/s**
 - **FX 5600**
 - **Compute Capability 1.0**
 - **Peak bandwidth of 77 GB/s**

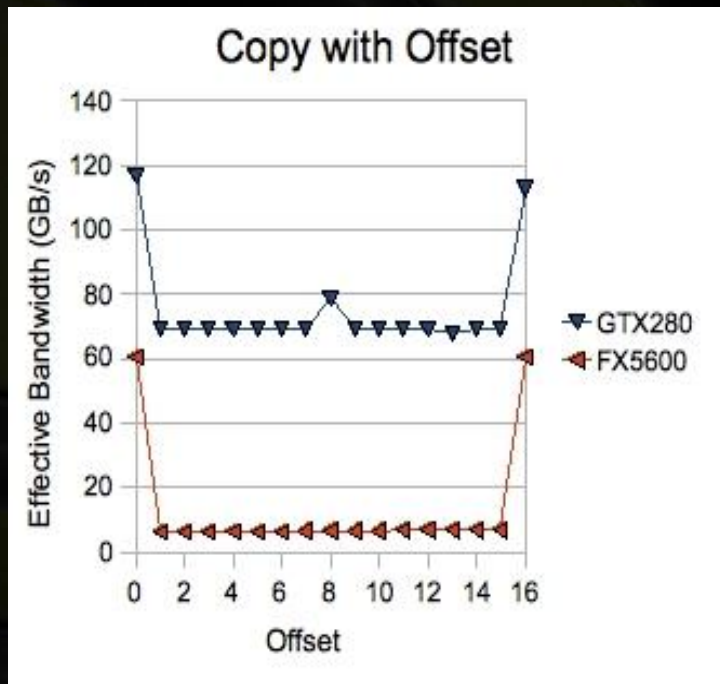
Coalescing – Misaligned Accesses



```
__global__ void offsetCopy(float* out,  
                           float* in,  
                           int offset)  
{  
    int i = threadIdx.x + blockIdx.x * blockDim.x;  
    out[i + offset] = in[i + offset];  
}
```



Memory access of halfwarp when offset = 1



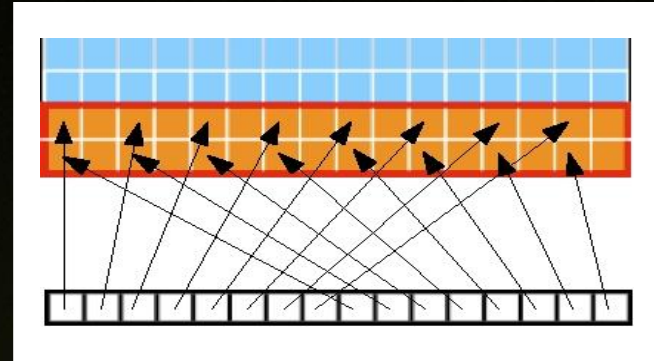
GTX-280 (compute capability 1.3) drops by a factor of 1.7

FX-5600 (compute capability 1.0) drops by a factor of 8. This is because 32 bytes (minimum transaction size) are fetched for each thread, and we only need 4 bytes.
 $4 / 32 = 1 / 8$ performance

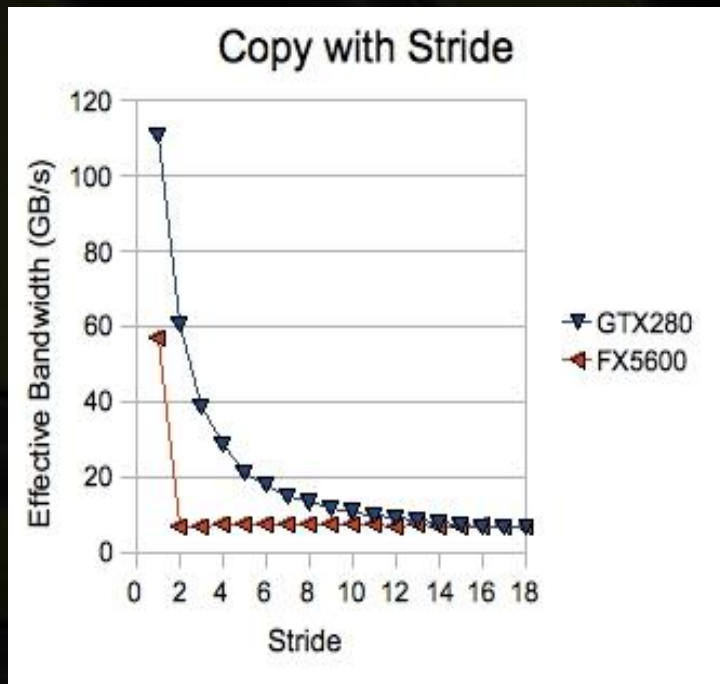
Coalescing – Strided Accesses



```
__global__ void strideCopy(float* out,  
                           float* in,  
                           int stride)  
{  
    int i = threadIdx.x + blockIdx.x * blockDim.x;  
    out[i * stride] = in[i * stride];  
}
```



Memory access of halfwarp when stride = 2



Large strides often arise in applications. However, strides can be avoided using shared memory.

Coalescing structs of size_t ≠ 4,8,16



- Use a “Structure of Arrays” (SoA) instead of “Array of Structures” (AoS)

Point structure:

x	y	z
---	---	---

AoS:

x	y	z	x	y	z	x	y	z	x	y	z
---	---	---	---	---	---	---	---	---	---	---	---

SoA:

x	x	x	x	y	y	y	y	z	z	z	z
---	---	---	---	---	---	---	---	---	---	---	---

```
struct __align__(16)
{
    float a;
    float b;
    float c;
};
```

- If SoA is not viable then...
 - Force structure alignment
 - `__align__(X)` where X = 4, 8, or 16

Aligned:

x	y	z
---	---	---

x	y	z
---	---	---

x	y	z
---	---	---

x	y	z
---	---	---

See SDK!
alignedTypes

Coalescing – Summary



- **Coalescing greatly improves throughput**
 - Critical for memory-bound kernels
- **Reading structs of size other than 4, 8, or 16 bytes breaks coalescing**
 - Prefer “Structures of Arrays” over AoS
 - Pad using: `__align__(X)`
- **Strided memory access is inherent in many multidimensional problems**
 - Stride is generally large ($\gg 18$)
 - But strided access to global memory can be avoided using SMEM...

Shared Memory



- **~Hundred times faster than global memory**
- **Cache data to reduce global memory accesses**
- **Threads can cooperate via shared memory**
 - Use one or more threads to load / compute data shared by all threads
- **Use it to avoid non-coalesced access**
 - Stage loads and stores in shared memory to re-order non-coalesceable addressing

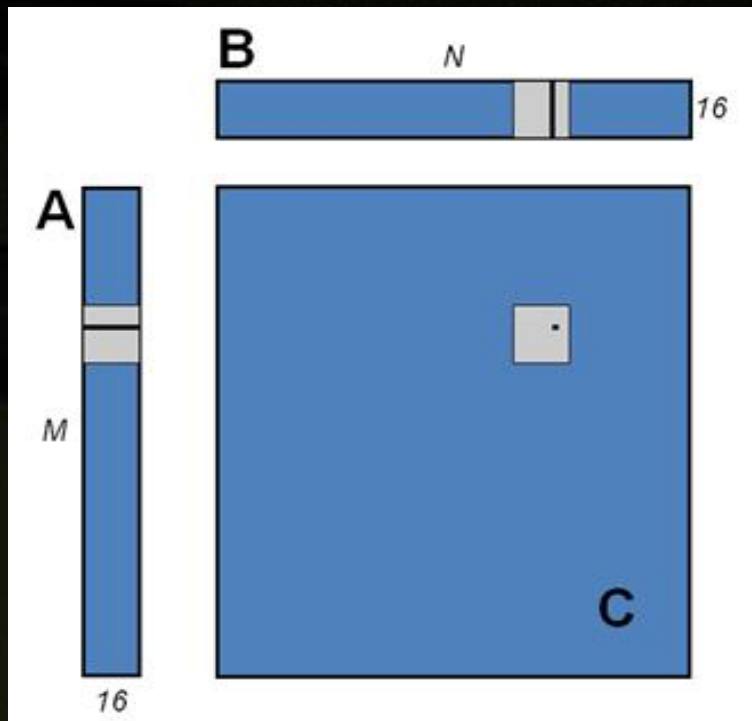
See SDK!
matrixTranspose

Memory Bandwidth



- **Effective bandwidth depends on access patterns**
- **Minimize device memory accesses**
 - Much lower bandwidth than on-chip shared memory
- **Common CUDA kernel structure:**
 - 1. Load data from global memory to shared memory
 - 2. `__syncthreads()`
 - 3. Process the data in shared memory with many threads
 - 4. `__syncthreads()` (if needed)
 - 5. Store results from shared memory to global memory
- **Notes:**
 - Steps 2 to 4 may be repeated, looped, etc.
 - Step 4 is not necessary if there is no dependence of stored data on other threads

Caching – MatMult example ($C=A \times B$)



Uncached version:

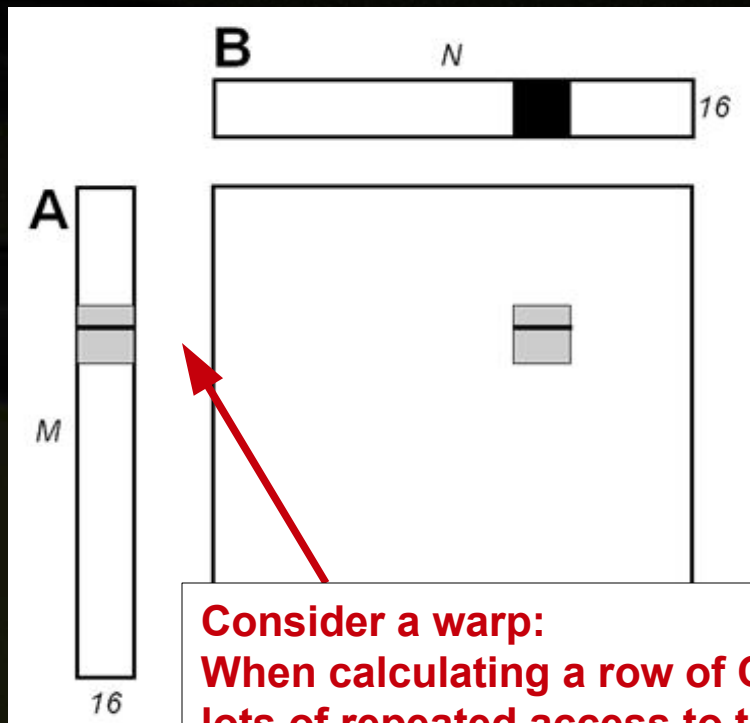
```
__global__ void simpleMultiply(float* a,
                                float* b,
                                float* c,
                                int N)
{
    int col = threadIdx.x+blockIdx.x*blockDim.x;
    int row = threadIdx.y+blockIdx.y*blockDim.y;

    float sum = 0.f;
    for (int i = 0; i < TILE_DIM; i++)
        sum += a[row*TILE_DIM+i] * b[i*N+col];

    c[row*N+col] = sum;
}
```

Every thread corresponds to one entry in C.

Caching – MatMult example ($C=A \times B$)



Consider a warp:
When calculating a row of C,
lots of repeated access to the same row of A.
Un-coalesced in CC <= 1.1.

Uncached version:

```
__global__ void simpleMultiply(float* a,
                                float* b,
                                float* c,
                                int N)
{
    int col = threadIdx.x+blockIdx.x*blockDim.x;
    int row = threadIdx.y+blockIdx.y*blockDim.y;

    float sum = 0.f;
    for (int i = 0; i < TILE_DIM; i++)
        sum += a[row*TILE_DIM+i] * b[i*N+col];

    c[row*N+col] = sum;
```

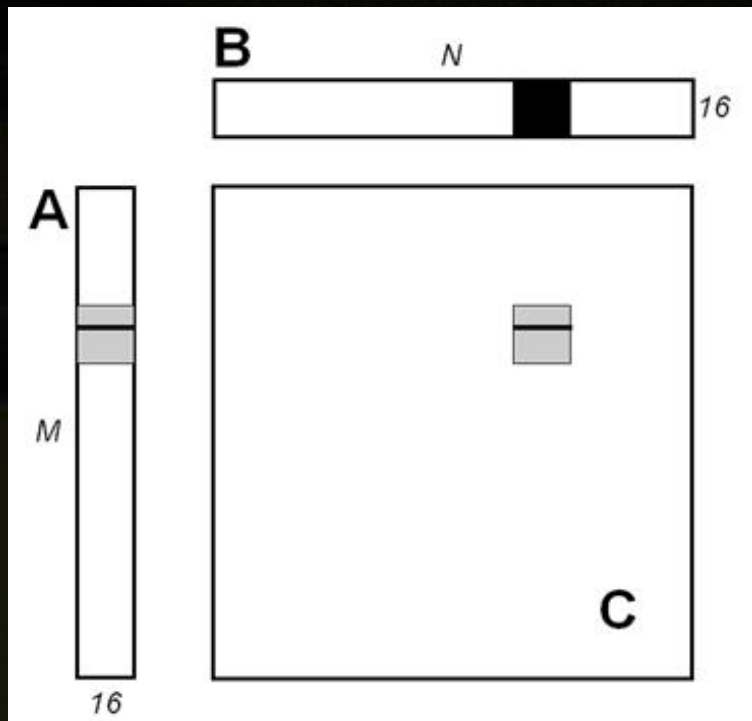
Every thread corresponds to one entry in C.

Caching – MatMult example – Results



Optimization	NVIDIA GeForce GTX 280	NVIDIA Quadro FX 5600
No optimization	8.8 GBps	0.62 GBps
Coalesced using shared memory to store a tile of A	14.3 GBps	7.34 GBps
Using shared memory to eliminate redundant reads of a tile of B	29.7 GBps	15.5 GBps

Caching – MatMult example ($C=A \times B$)



Cached & coalesced version:

```
__global__ void coalescedMultiply(float* a,
                                   float* b,
                                   float* c,
                                   int N)
{
    __shared__ float aTile[TILE_DIM][TILE_DIM];
    int col = threadIdx.x + blockIdx.x * blockDim.x;
    int row = threadIdx.y + blockIdx.y * blockDim.y;

    // coalesced load of tile into smem
    int x = threadIdx.x;
    int y = threadIdx.y;
    aTile[y][x] = a[row * TILE_DIM + x];

    // no synchronization required

    float sum = 0.f;
    for (int i = 0; i < TILE_DIM; i++)
        sum += aTile[y][i] * b[i * N + col];

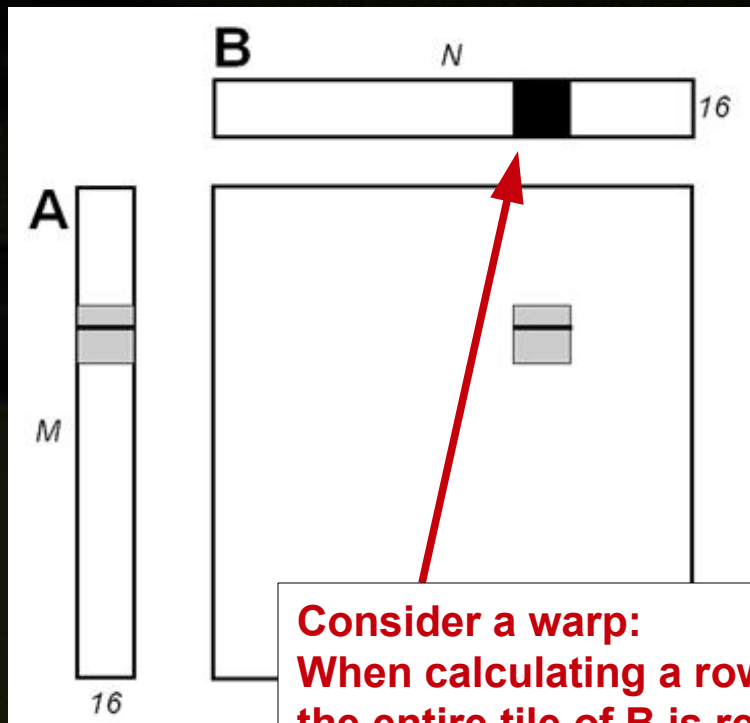
    c[row * N + col] = sum;
}
```


Caching – MatMult example – Results



Optimization	NVIDIA GeForce GTX 280	NVIDIA Quadro FX 5600
No optimization	8.8 GBps	0.62 GBps
Coalesced using shared memory to store a tile of A	14.3 GBps	7.34 GBps
Using shared memory to eliminate redundant reads of a tile of B	29.7 GBps	15.5 GBps

Caching – MatMult example ($C=A \times B$)



**Consider a warp:
When calculating a row of C,
the entire tile of B is read**

Cached & coalesced version:

```
__global__ void coalescedMultiply(float* a,
                                   float* b,
                                   float* c,
                                   int N)
{
    __shared__ float aTile[TILE_DIM][TILE_DIM];
    int col = threadIdx.x + blockIdx.x * blockDim.x;
    int row = threadIdx.y + blockIdx.y * blockDim.y;

    // coalesced load of tile into smem
    int x = threadIdx.x;
    int y = threadIdx.y;
    aTile[y][x] = a[row * TILE_DIM + x];

    // no synchronization required

    float sum = 0.f;
    for (int i = 0; i < TILE_DIM; i++)
        sum += aTile[y][i] * b[i * N + col];

    c[row * N + col] = sum;
}
```

Caching – MatMult example



Cached & coalesced version:

```
__kernel void sharedABMultiply(__global float* a, __global float* b, __global float*
c, int N)
{
    __shared__ float aTile[TILE_DIM][TILE_DIM];
    __shared__ float bTile[TILE_DIM][TILE_DIM];
    int col = threadIdx.x+blockIdx.x*blockDim.x;
    int row = threadIdx.y+blockIdx.y*blockDim.y;

    // coalesced load of tile into smem
    int x = threadIdx.x;
    int y = threadIdx.y;
    aTile[y][x] = a[row * TILE_DIM + x];
    bTile[y][x] = b[y * N + col];
    // we need to sync block because we are reading from different columns of bTile
    barrier(CLK_LOCAL_MEM_FENCE);

    float sum = 0.f;
    for (int i = 0; i < TILE_DIM; i++)
        sum += aTile[y][i] * bTile[i][x];

    c[row*N+col] = sum;
}
```

Caching – MatMult example – Results

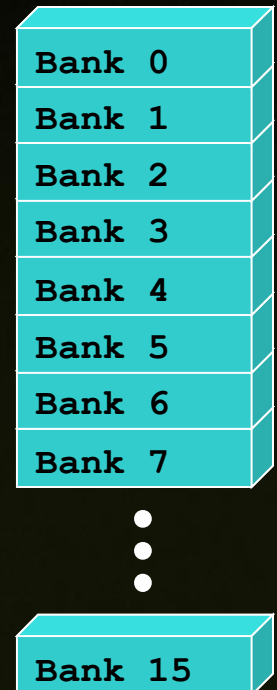


Optimization	NVIDIA GeForce GTX 280	NVIDIA Quadro FX 5600
No optimization	8.8 GBps	0.62 GBps
Coalesced using shared memory to store a tile of A	14.3 GBps	7.34 GBps
Using shared memory to eliminate redundant reads of a tile of B	29.7 GBps	15.5 GBps

Shared Memory – Banked Architecture



- Shared memory is divided into banks
 - 32-bit words assigned to successive banks
 - Number of banks = 16 for CC 1.x
 - $\text{bank} = \text{address} \% 16$
- Each bank services one address per cycle
 - Memory can service as many simultaneous accesses as it has banks
- Simultaneous accesses to a bank result in a **bank conflict**
 - Conflicting accesses are serialized
 - Conflicts can only occur within a half-warp

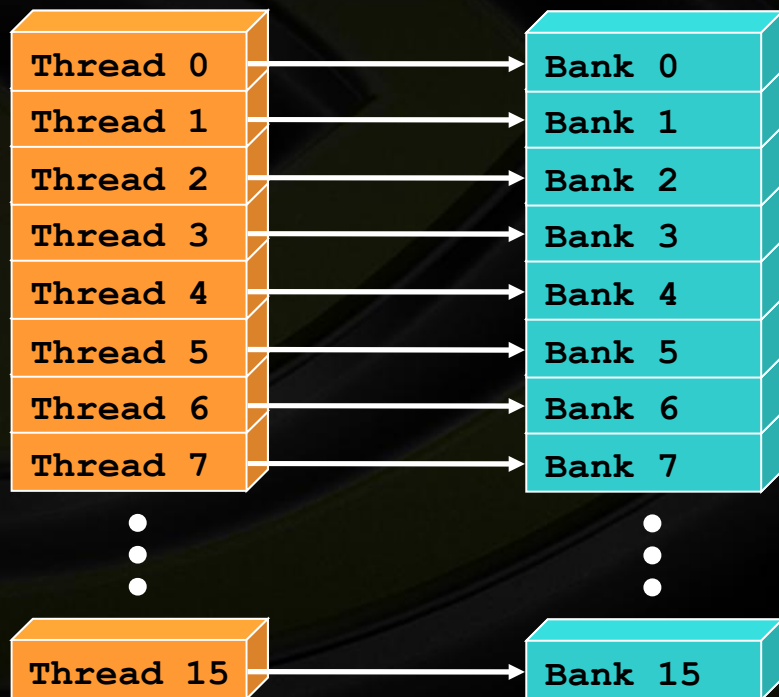


Shared Memory – Bank Addressing



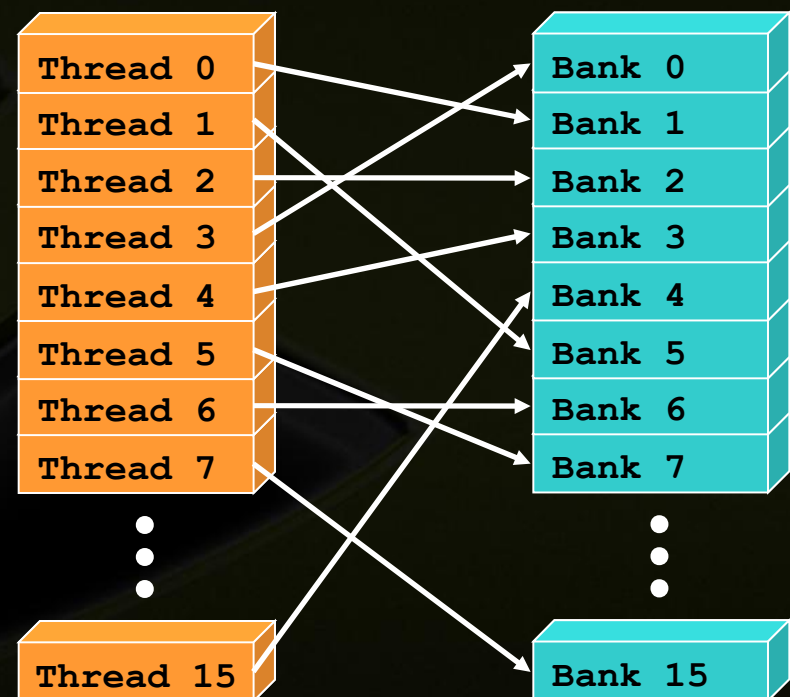
- **No Bank Conflicts**

- Linear addressing



- **No Bank Conflicts**

- Random 1:1 Permutation

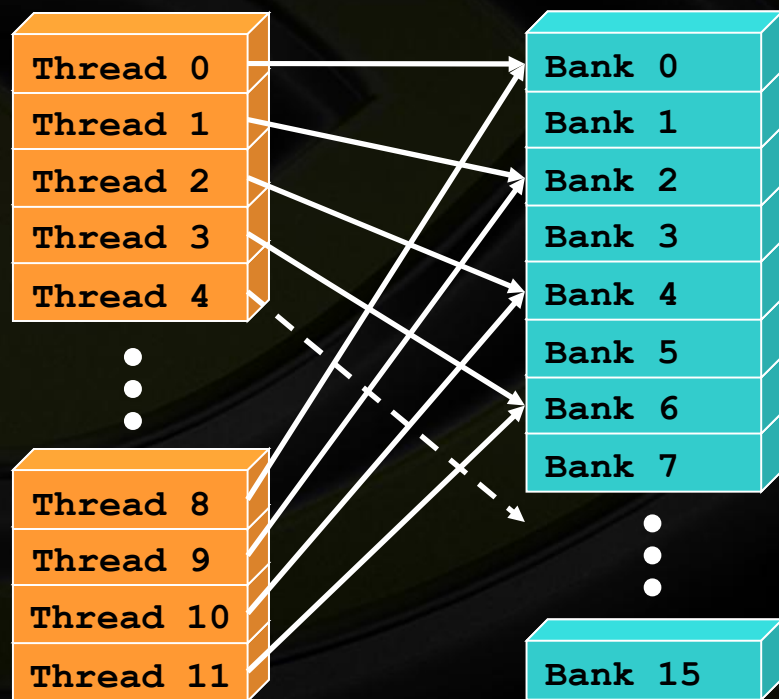


Shared Memory – Bank Addressing



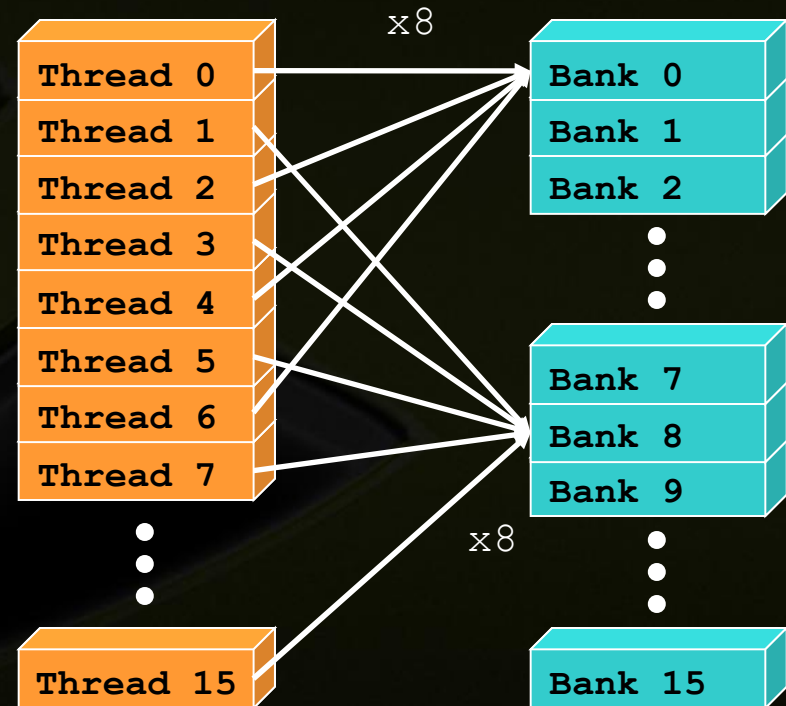
2-way Bank Conflicts

- Linear addressing (stride = 2)



8-way Bank Conflicts

- Linear addressing (stride = 8)



Shared Memory - Bank conflicts



- Shared memory is as fast as registers **if there are no bank conflicts**
 - The fast case:
 - All threads of halfwarp access different banks → no bank conflict
 - All threads of halfwarp read **identical address** → no bank conflict
 - The slow case:
 - multiple threads in the halfwarp access same bank → bank conflict
 - Must serialize the accesses
 - **Cost = max # of simultaneous accesses to a single bank**
- Use the bank checker macro in the SDK to check for conflicts
- A 2nd order effect compared to GMEM coalescing
 - No benefit if it costs more instructions to avoid it

Avoiding un-coalesced float3 access



```
__global__ void calc_float3(float3* in, float3* out)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    float3 v = in[i];

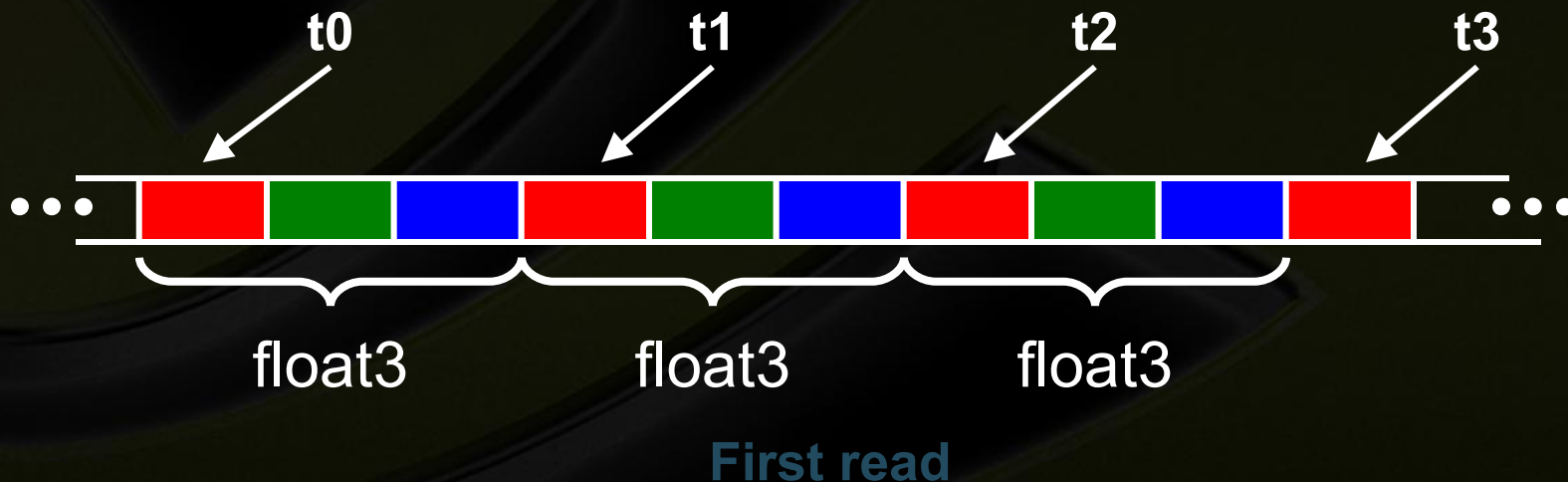
    v.x += 2;
    v.y += 2;
    v.z += 2;

    out[i] = v;
}
```

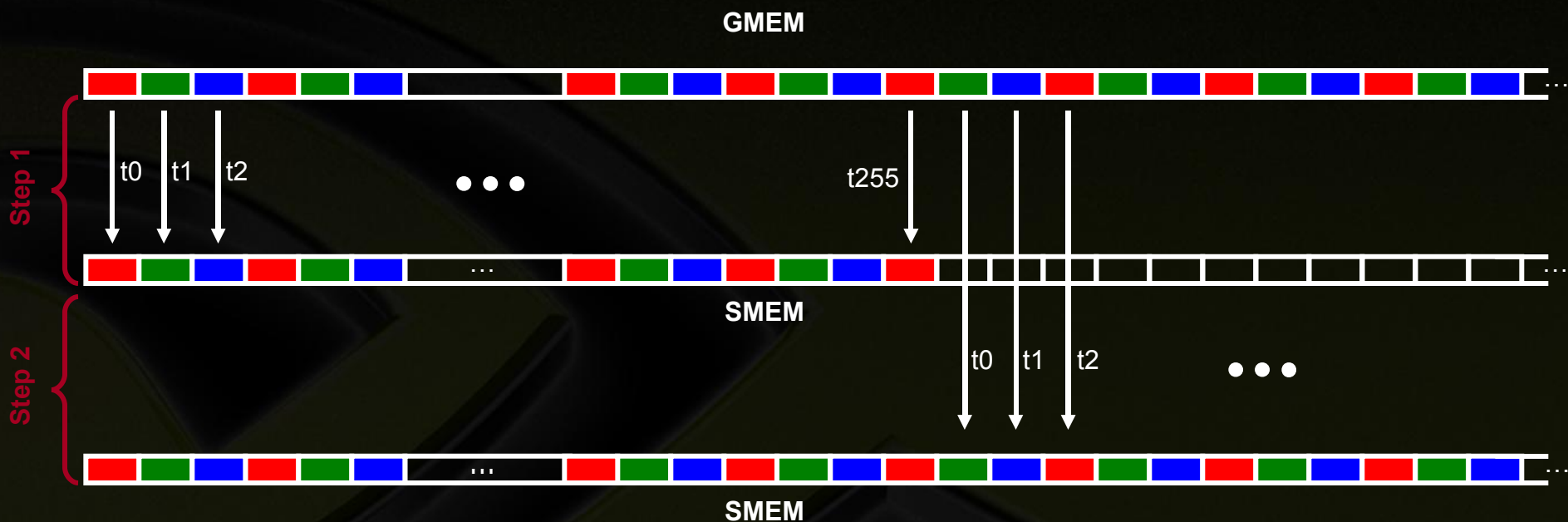
Avoiding un-coalesced float3 access



- float3 is 12 bytes
- Each thread ends up executing 3 reads
 - `sizeof(float3) ≠ 4, 8, or 16`
 - Halfwarp reads three 64B **non-contiguous** regions



Avoiding un-coalesced float3 access



Similarly, Step3 starting at offset 512

Avoiding un-coalesced float3 access



- **Use shared memory to allow coalescing**
 - Need `sizeof(float3) * (threads/block)` bytes of SMEM
 - Each thread reads 3 scalar floats:
 - Offsets: 0, `(threads/block)`, `2*(threads/block)`
 - These will likely be processed by other threads, so sync
- **Processing**
 - Each thread retrieves its float3 from SMEM array
 - Cast the SMEM pointer to `(float3*)`
 - Use thread ID as index
 - Rest of the compute code does not change!

Avoiding un-coalesced float3 access



Read the input
through SMEM

Compute code
is not changed

Write the result
through SMEM

```
__global__ void calc_float3_smem(float *in, float *out)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    __shared__ float smem[256 * 3];
    smem[threadIdx.x + 0] = in[i + 0];
    smem[threadIdx.x + 256] = in[i + 256];
    smem[threadIdx.x + 512] = in[i + 512];
    __syncthreads();
    float3 v = ((float3*) smem)[threadIdx.x];

    v.x += 2;
    v.y += 2;
    v.z += 2;

    ((float3*) smem)[threadIdx.x] = v;
    __syncthreads();
    out[i + 0] = smem[threadIdx.x + 0];
    out[i + 256] = smem[threadIdx.x + 256];
    out[i + 512] = smem[threadIdx.x + 512];
}
```

Avoiding un-coalesced float3 access



- **Experiment:**
 - Kernel: read a float, increment, write back
 - 3M floats (12MB)
 - Times averaged over 10K runs
- **12K blocks x 256 threads:**
 - 356 μ s – coalesced
 - 357 μ s – coalesced, some threads don't participate
 - 3,494 μ s – permuted/misaligned thread access (G80)
- **4K blocks x 256 threads:**
 - 3,302 μ s – float3 uncoalesced
 - **359 μ s – float3 coalesced through shared memory**

Texture and Constant Memory Performance



- **Texture partition is cached**
 - Uses the texture cache also used for graphics
 - Optimized for 2D spatial locality
 - Best performance when threads of a warp read locations that are close together in 2D
- **Constant memory is cached**
 - 4 cycles per address read within a single warp
 - Total cost 4 cycles if all threads in a warp read same address
 - Total cost 64 cycles if all threads read different addresses

Texture overview



- **Texture is an object for reading data**
- **Benefits:**
 - **Data is cached (optimized for 2D locality)**
 - Helpful when coalescing is a problem
 - **Filtering**
 - Linear / bilinear / trilinear
 - dedicated hardware
 - **Wrap modes (for “out-of-bounds” addresses)**
 - Clamp to edge / repeat
 - **Addressable in 1D, 2D, or 3D**
 - Using integer or normalized coordinates
- **Usage:**
 - CPU code binds data to a texture object
 - Kernel reads data by calling a fetch function

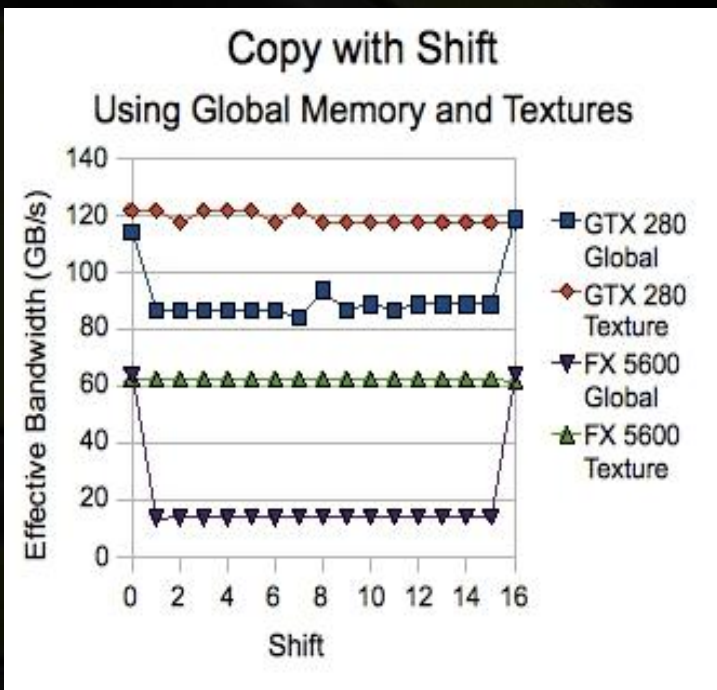
Textures – Misaligned Accesses



- Texture fetch read
- Coalesced write

```
__global__ void shiftCopy(float* odata,  
                          float* idata,  
                          int offset)  
{  
    int i = blockIdx.x*blockDim.x+threadIdx.x;  
  
    odata[i] = idata[i + offset];  
}
```

```
texture<float> tex_ref;  
  
__global__ void texShiftCopy(float* odata,  
                             float* idata,  
                             int offset)  
{  
    int i = blockIdx.x*blockDim.x+threadIdx.x;  
  
    odata[i] = tex1Dfetch(tex_ref, i + offset);  
}
```



Instruction Performance



- **Instruction cycles (per warp) = sum of**
 - **Operand read cycles**
 - **Instruction execution cycles**
 - **Result update cycles**
- **Therefore instruction throughput depends on**
 - **Nominal instruction throughput**
 - **Memory latency**
 - **Memory bandwidth**
- **“Cycle” refers to the multiprocessor clock rate**
 - **1.35 GHz on the Tesla C870, for example**

Instruction Throughput



- In SIMT architecture,
 - T = number of operations per cycle
 - SM instruction throughput = one instruction every $(\text{warpSize} / T)$ cycle
- Maximizing throughput
 - using smaller number of cycles to get the job done

Arithmetic Instruction Throughput



- **integer & float:** `add, shift, min, max`
- **float:** `mul, mad`
 - `T = 8 ops per cycle, 32 / 8 = 4 cycles per warp`
 - Integer multiply defaults to 32-bit
 - Requires multiple cycles / warp
 - Use `__mul24()` / `__umul24()` intrinsics for 4-cycle 24-bit integer multiply
- Integer divide and modulo are more expensive
 - Compiler will convert literal power-of-2 divides to shifts
 - But we have seen it miss some cases
 - Be explicit in cases where compiler can't tell that divisor is a power of 2!
 - Useful trick: `foo%n == foo&(n-1)` if `n` is a power of 2

Runtime Math Library



- **Two types of runtime math operations**

- **`__func()`**

- direct mapping to native hardware ISA
 - Fast (16 cycles) but lower accuracy (see prog. guide for details)
 - Examples:

- `__sin(x)`, `__exp(x)`, `__pow(x,y)`

- **`func()`**

- compile to multiple instructions, e.g. `sqrt(x) == x * rsqrt(x)` (20 cycles per warp)
 - Slower but higher accuracy (5 ulp or less)
 - Examples:

- `sin(x)`, `exp(x)`, `pow(x,y)`

- **trigonometric funcs**

- **WARNING:**

- slower path `x > 48039.0f` and `x > 2147483648.0`

- uses LMEM for intermediate values

Runtime Math Library – OpenCL



- Two types of runtime math operations

- `native_func()`

- direct mapping to native hardware ISA

- Examples:

- `native_sin(x)`, `native_exp(x)`, `native_divide(x,y)`

- `func()`

- Examples:

- `sin(x)`, `exp(x)`, `pow(x,y)`

Compile time optimization



- **CUDA-C**

- `-use_fast_math`
 - coerces all `func()` calls to compile as `__func()`

- **OpenCL**

- `-cl-fast-relaxed-math`
- `-cl-mad-enable` permits use of FMADS

Conversion instructions



- chars and shorts will likely need to be converted to int when used in functions
- Newer hardware has double precision support
 - Double precision has additional cost
 - Be float-safe to avoid using double precision where it is not needed
 - Add 'f' specifier on float literals:
 - `foo = bar * 0.123;` // double assumed
 - `foo = bar * 0.123f;` // float explicit
 - Use float version of standard library functions:
 - `foo = sin(bar);` // double assumed
 - `foo = sinf(bar);` // float explicit

Divergence – Control Flow



- **Main performance concern with branching is divergence**
 - If threads within a single warp take different paths, different execution paths must be serialized
- **Avoid divergence when branch condition is a function of thread ID**
 - **Example with divergence:**
 - `if (threadIdx.x > 2) {...}`
 - Branch granularity < warp size
 - **Example without divergence:**
 - `if (threadIdx.x / WARP_SIZE > 2) {...}`
 - Branch granularity is a whole multiple of warp size

Divergence – Instruction Predication



- **Comparison instructions set condition codes (CC)**
- **Instructions can be predicated to write results only when CC meets criterion (CC != 0, CC >= 0, etc.)**
- **Compiler tries to predict if a branch condition is likely to produce many divergent warps**
 - If guaranteed not to diverge: only predicates if < 4 instructions
 - If not guaranteed: only predicates if < 7 instructions
- **May replace branches with instruction predication**
- **ALL predicated instructions take execution cycles**
 - Those with false conditions don't write their output
 - Or invoke memory loads and stores
 - Saves branch instructions, so can be cheaper than serializing divergent paths

Divergence – Compiler hints



- The compiler unrolls small loops with known trip count
- For more control use: `#pragma unroll <n>`
- Up to the programmer to ensure efficiency
- Example:
 - The loop below is unrolled 5 times

```
#pragma unroll 5
for (int i=0; i<n; ++i)
{
    ...
}
```

The Art of Performance Optimization



- GPU can achieve great performance on data-parallel computations if you follow a few simple guidelines:

Minimize, speed up, or hide host-transfer

Use parallelism efficiently

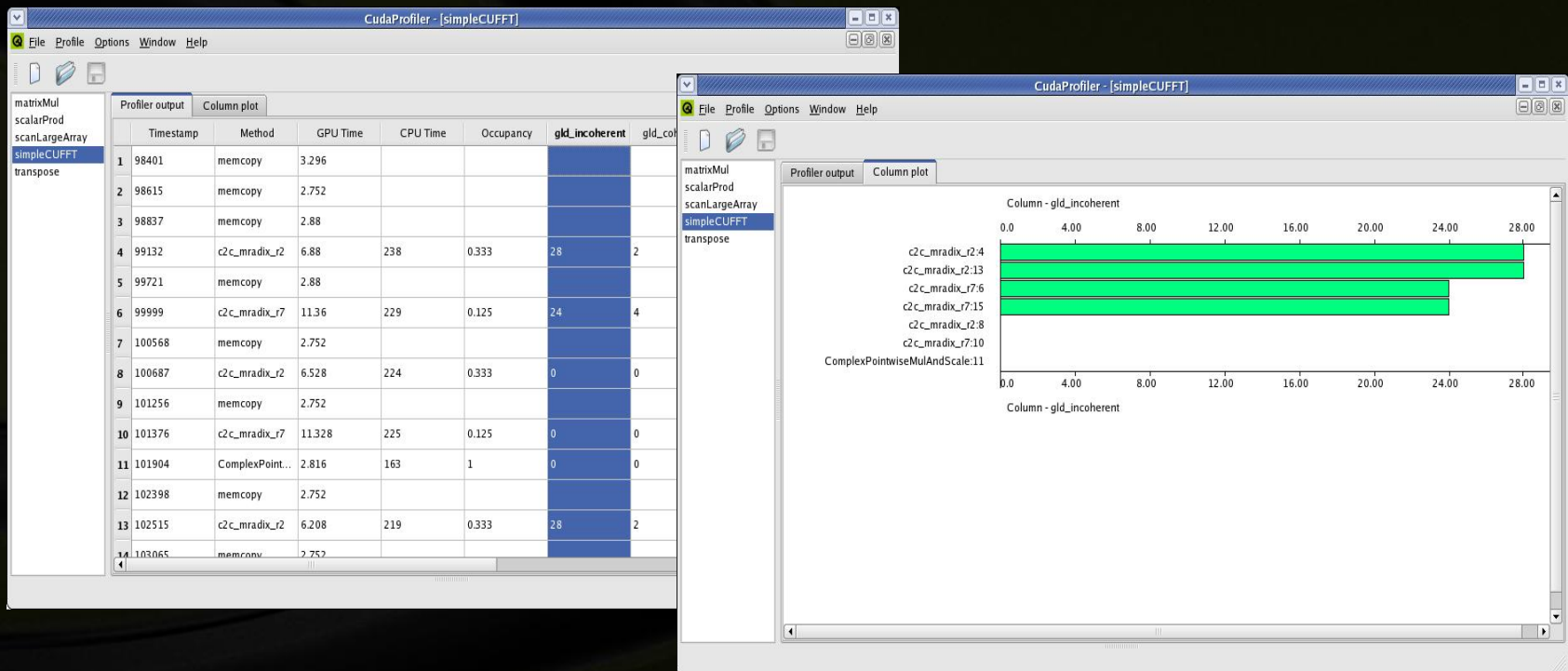
Keep memory aligned, access it coalesced, explore other memory spaces

Maximize instruction throughput

CUDA Visual Profiler



- Uses a special operation mode of the GPU to log important signals
- Best to isolate kernels in a simple application



Visual Profiler



- **Profiler facilitates analysis and optimization of CUDA programs by**
 - **Reporting hardware counter values:**
 - Number of various bus transactions
 - Branches
 - Effective Parallelism
 - Etc.
 - **Computing per kernel statistics:**
 - Effective instruction throughput
 - Effective memory throughput
 - **Visually displaying time spent in various GPU calls**
 - Requires no instrumentation of the source code
- **Works with OpenCL too...**

Visual Profiler – Signals



- Events are tracked with hardware counters on signals in the chip:

timestamp

- gld_incoherent
 - gld_coherent
 - gst_incoherent
 - gst_coherent
 - local_load
 - local_store
 - branch
 - divergent_branch
- } Global memory loads/stores are coalesced (coherent) or non-coalesced (incoherent)
- } Local loads/stores
- } Total branches and divergent branches taken by threads
- instructions – instruction count
 - warp_serialize – thread warps that serialize on address conflicts to shared or constant memory
 - cta_launched – executed thread blocks

Profiling on the command line



- **Text file output**

- **Environment variables:**

- **CUDA_PROFILE=1**

- Tells CUDA to calculate and output profiling data

- **CUDA_PROFILE_CSV=1**

- data is exported as csv for loading into spreadsheet or VisualProfiler

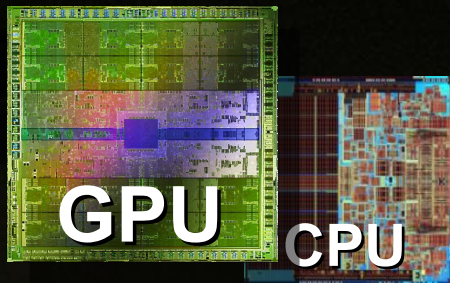
```
# CUDA_PROFILE_LOG_VERSION 1.5
# CUDA_DEVICE 1 Quadro CX
# CUDA_PROFILE_CSV 1
# TIMESTAMPFACTOR fb085cc80547cc8
method,gputime,cputime,occupancy,gld_coherent,gld_incoherent,gst_coherent,gst_incoherent
_Z18integrate_GPU_SMEMP6float3S0_,10.208,42.000,0.500,0,0,0,0
_Z18integrate_GPU_SMEMP6float3S0_,10.048,560.000,0.500,0,0,0,0
_Z18integrate_GPU_SMEMP6float3S0_,10.080,1468.000,0.500,0,0,0,0
...
_Z18integrate_GPU_SMEMP6float3S0_,9.472,893.000,0.500,192,0,1152,0
```

Interpreting profiler counters



- **Values represent events within a thread warp**
- **Only targets one multiprocessor**
 - Values will not correspond to the total number of warps launched for a particular kernel.
 - Launch enough thread blocks to ensure that the target multiprocessor is given a consistent percentage of the total work.
- **Values are best used to identify relative performance differences between unoptimized and optimized code**
 - e.g., make the number of non-coalesced loads go from some non-zero value to zero

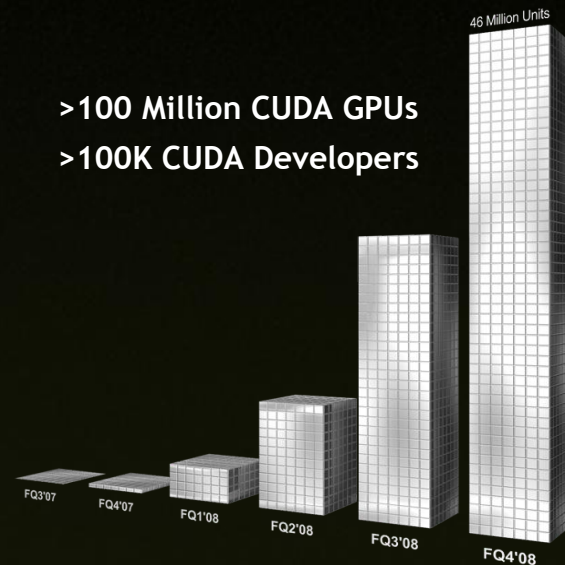
Questions?



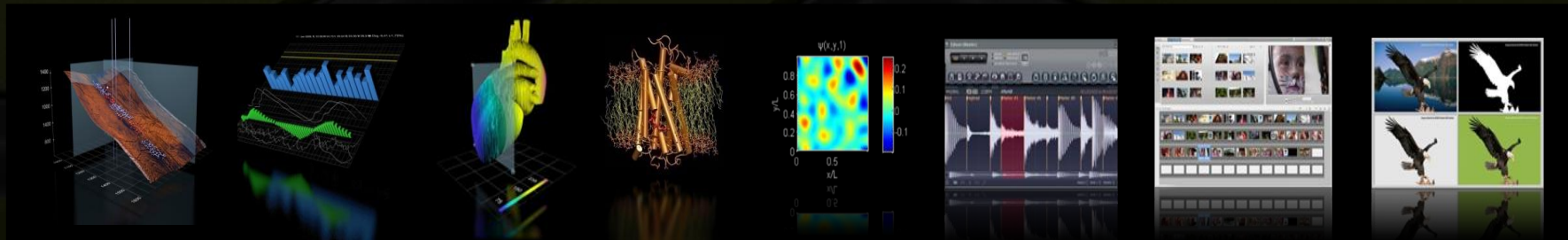
Heterogeneous Computing

CUDA

>100 Million CUDA GPUs
>100K CUDA Developers



www.nvidia.com/CUDA



Oil & Gas

Finance

Medical

Biophysics

Numerics

Audio

Video

Imaging