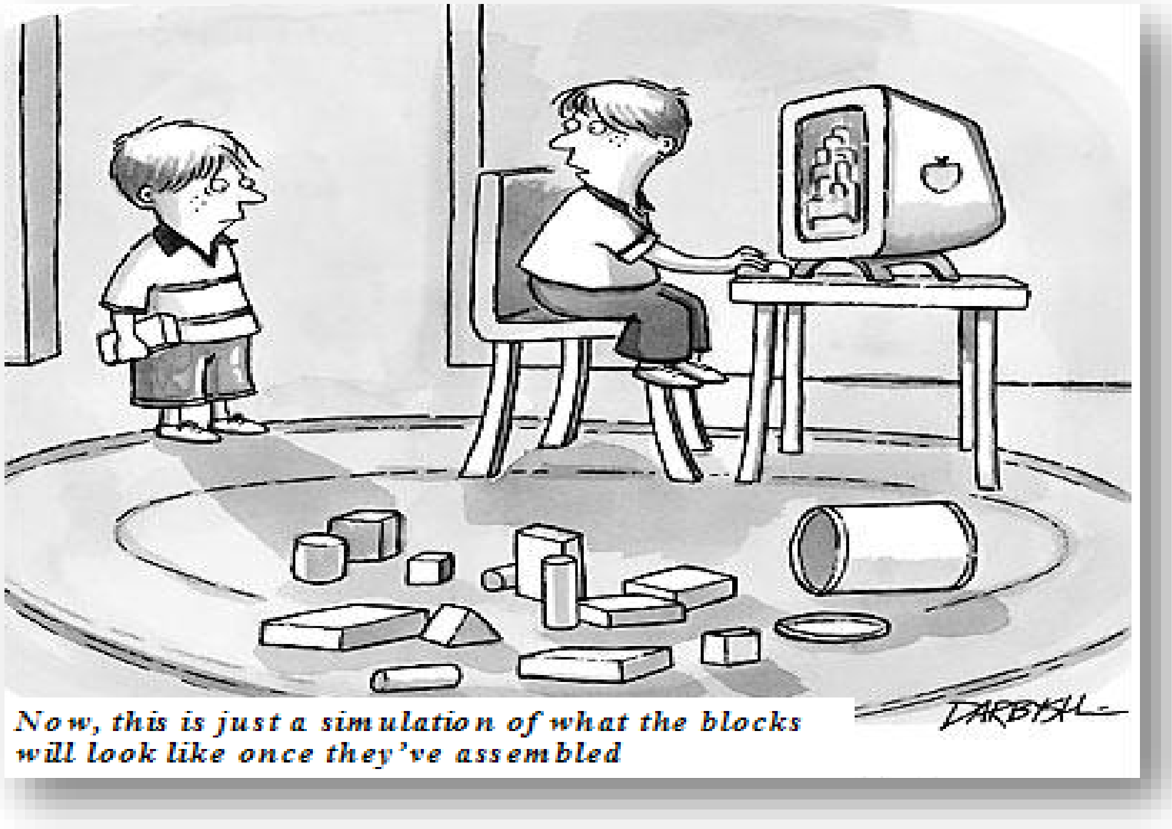
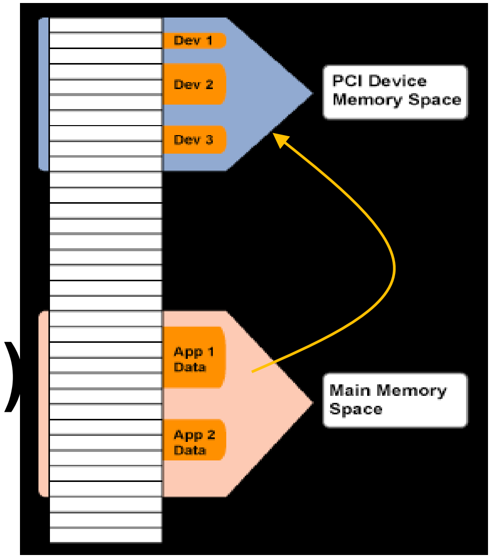
# ECE569
# Module 42

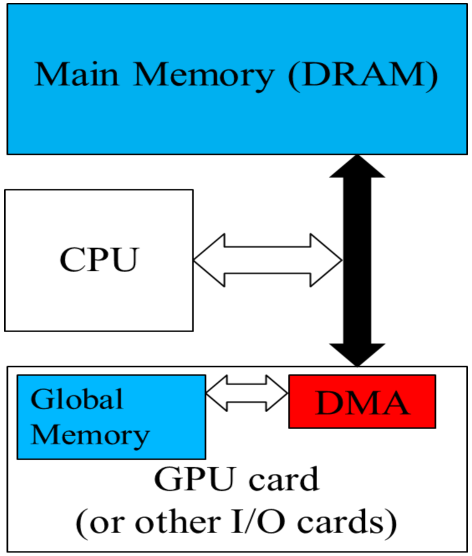



- CUDA Streams

# Host GPU Interaction: Pinned Memory

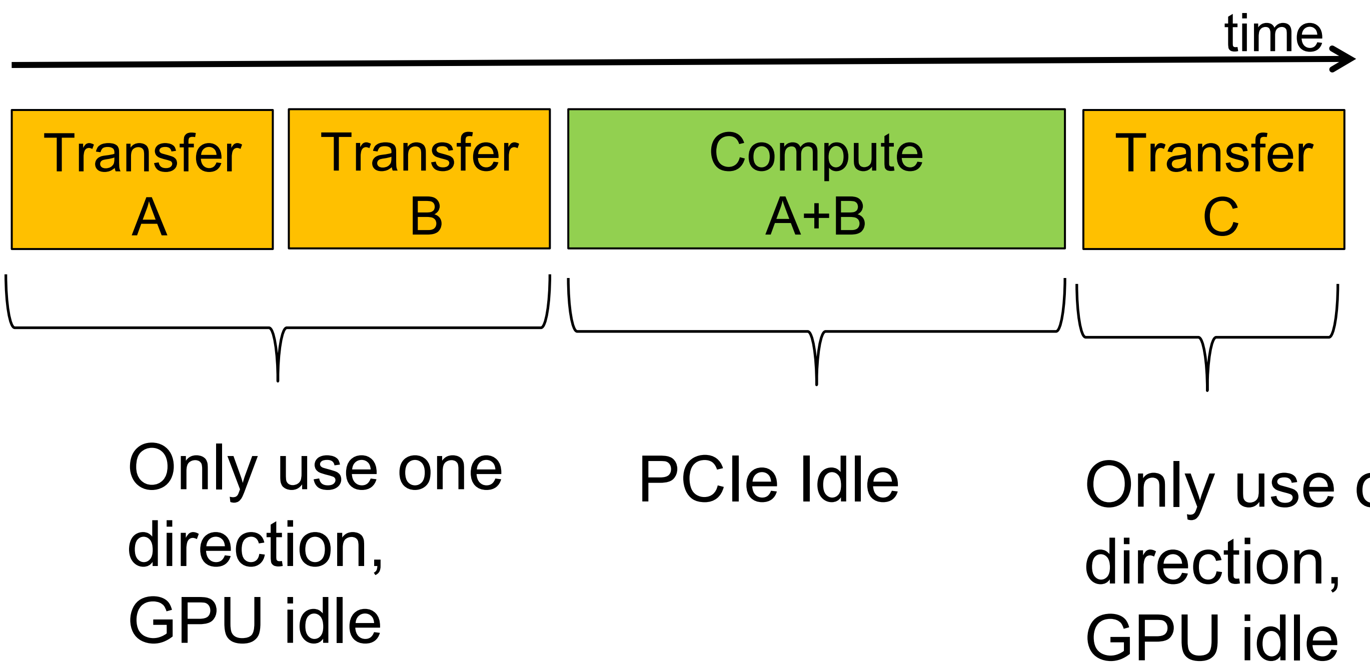
- **PCIe**
  - From host to device
    - Date preparation (staging) needed before transmitting the data
- **cudaHostMalloc()/cudaHostRegister()**
  - Allocate pinned host memory so that data copied over PCIe without the staging
  - Pinned host memory
    - Faster (2x)
    - Enables asynchronous memory transfer: cudaMemcpyAsync();
      - Execution continues on host
    - Allows CPU keep working while the data transfer in progress
      - cudaMemcpy(); (CPU execution blocked)



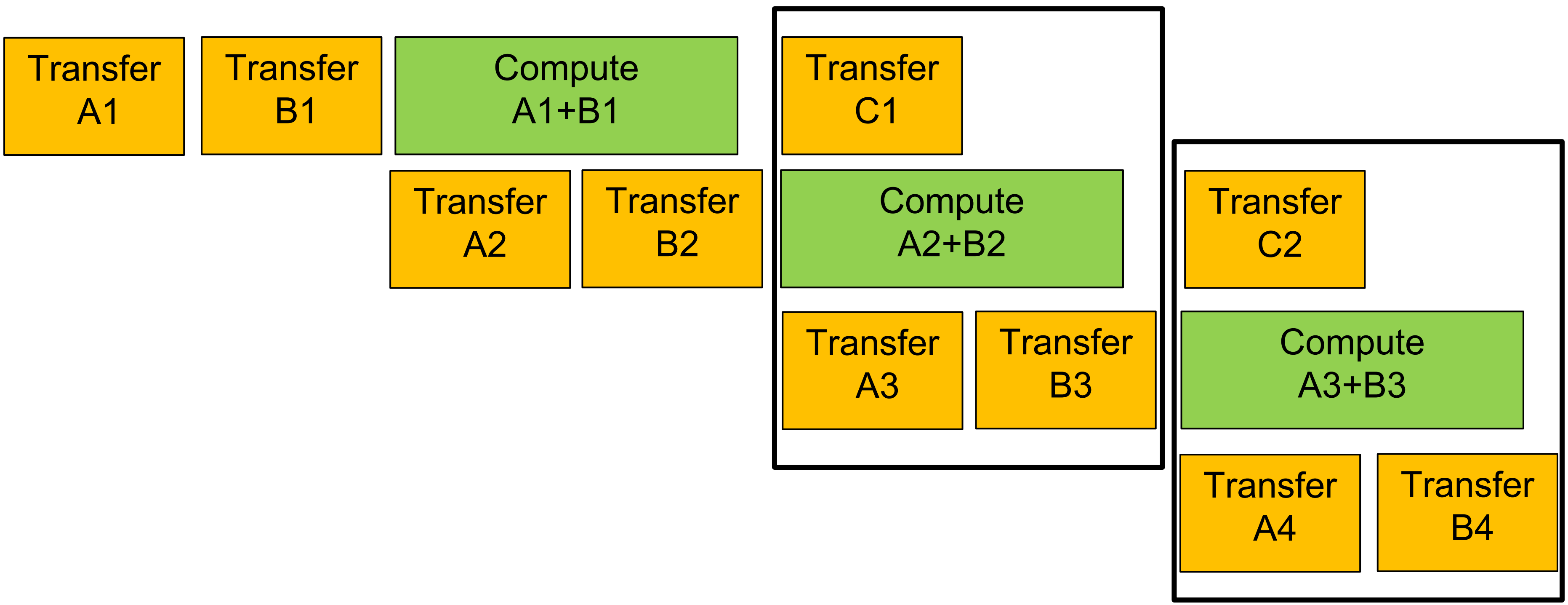
# Vector Addition – Data Transfer and Execution

- cudaMemcpy serializes data transfer and GPU computation



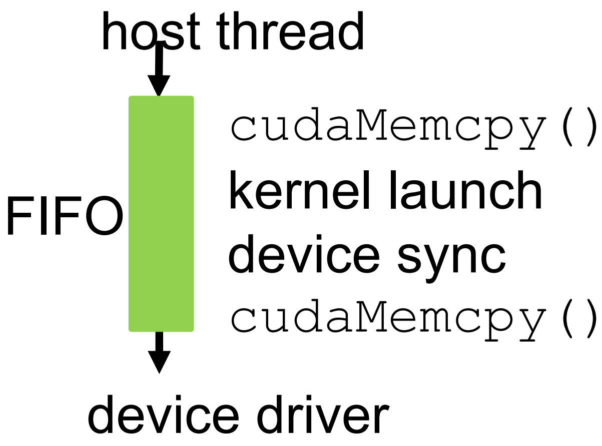
# Pipelined Execution for Vector Addition

- Divide large vectors into segments
- Overlap transfer and compute of adjacent segments



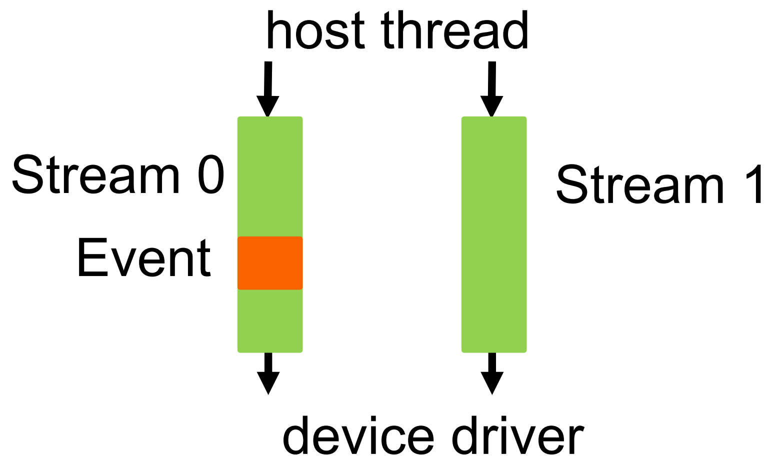
# CUDA Streams

- CUDA supports parallel execution of kernels and cudaMemcpy() with “Streams”
- Each stream is a queue of operations
  - kernel launches and cudaMemcpy() calls
- Operations (tasks) in different streams can go in parallel
  - “Task parallelism”
- Requests made from the host code are put into First-In-First-Out queues
  - Queues are read and processed asynchronously by the driver and device
  - Driver ensures that commands in a queue are processed in sequence.  E.g., Memory copies end before kernel launch, etc.



# CUDA Streams

- **To allow concurrent copying and kernel execution, use multiple queues, called "streams"**
  - CUDA "events" allow the host thread to query and synchronize with individual queues (i.e. streams).



# Streams

```
cudaStream_t s1;
cudaStreamCreate(&s1);
cudaStreamDestroy(s1);
```

# Conceptual View of Streams: Host Code

```
cudaStream_t stream0, stream1;
cudaStreamCreate(&stream0);
cudaStreamCreate(&stream1);

// device memory for stream 0
float *d_A0, *d_B0, *d_C0;
// device memory for stream 1
float *d_A1, *d_B1, *d_C1;

// cudaMalloc() calls for
// d_A0, d_B0, d_C0, d_A1, d_B1, d_C1
```

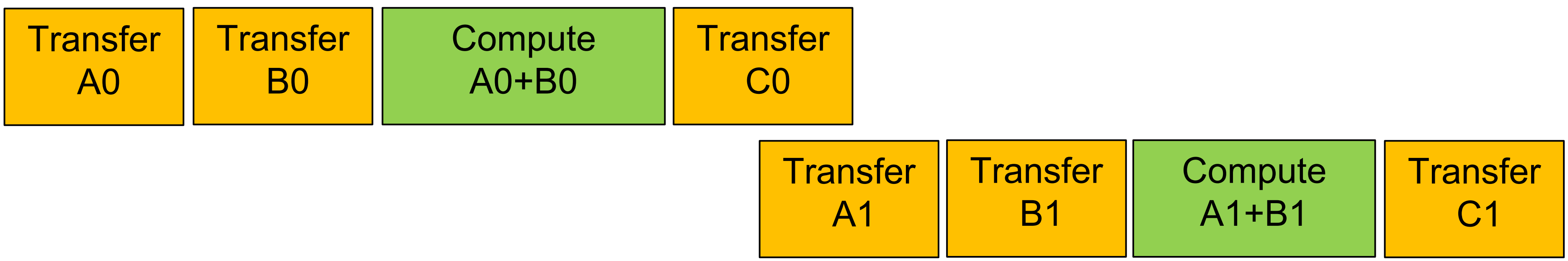
# Execution of Two Streams

Section size = 256*4bytes = 1KB

```
for (int i=0; i<n; i+=SecSize*2) {
// Stream 0 queue
   cudaMemcpyAsync(d_A0, h_A+i, SecSize*sizeof(float),…,
stream0);
   cudaMemcpyAsync(d_B0, h_B+i, SecSize*sizeof(float),…,
stream0);
   vecAdd<<<Size/256, 256, 0, stream0>>>(d_A0, d_B0,…);
   cudaMemcpyAsync(h_C+i, d_C0, SecSize*sizeof(float),…,
stream0);
// Stream 1 queue
   cudaMemcpyAsync(d_A1, h_A+i+SecSize,
SecSize*sizeof(float),…, stream1);
   cudaMemcpyAsync(d_B1, h_B+i+SecSize,
SecSize*sizeof(float),…,   stream1);
   vecAdd<<<Size/256, 256, 0, stream1>>>(d_A1, d_B1, …);
   cudaMemcpyAsync(h_C+i+SecSize, d_C1,
SecSize*sizeof(float),…,   stream1);
}
```

# Overlapping Data Transfers with Computation

- **Kernel Launches and cudaMemcopy()**
  - Streaming code we implemented generates the following execution



# Execution of Two Streams

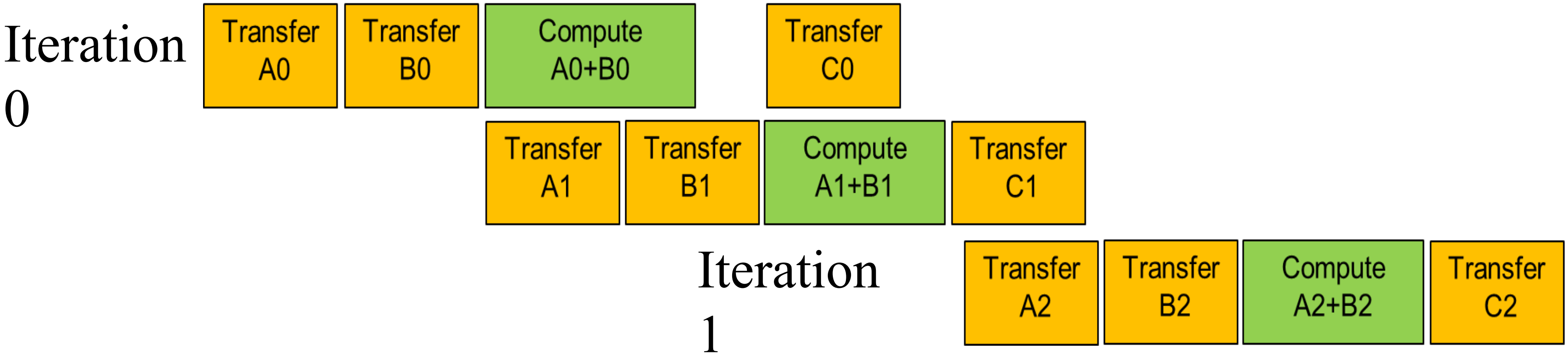
```
for (int i=0; i<n; i+=SecSize*2) {
cudaMemcpyAsync(d_A0, h_A+i, SecSize*sizeof(float),…,
stream0);
cudaMemcpyAsync(d_B0, h_B+i, SecSize*sizeof(float),…,
stream0);
cudaMemcpyAsync(d_A1, h_A+i+SecSize, SecSize*sizeof(float),…,
stream1);
cudaMemcpyAsync(d_B1, h_B+i+SecSize, SecSize*sizeof(float),…,
       stream1);

vecAdd<<<Size/256, 256, 0, stream0>>>(d_A0, d_B0,…);
vecAdd<<<Size/256, 256, 0, stream1>>>(d_A1, d_B1, …);

cudaMemcpyAsync(h_C+i, d_C0, SecSize*sizeof(float),…,
stream0);
cudaMemcpyAsync(h_C+i+SecSize, d_C1, SecSize*sizeof(float),…,
       stream1);
}
```

# Somewhat better overlap

- C1 blocks A2 and B2 from next iteration



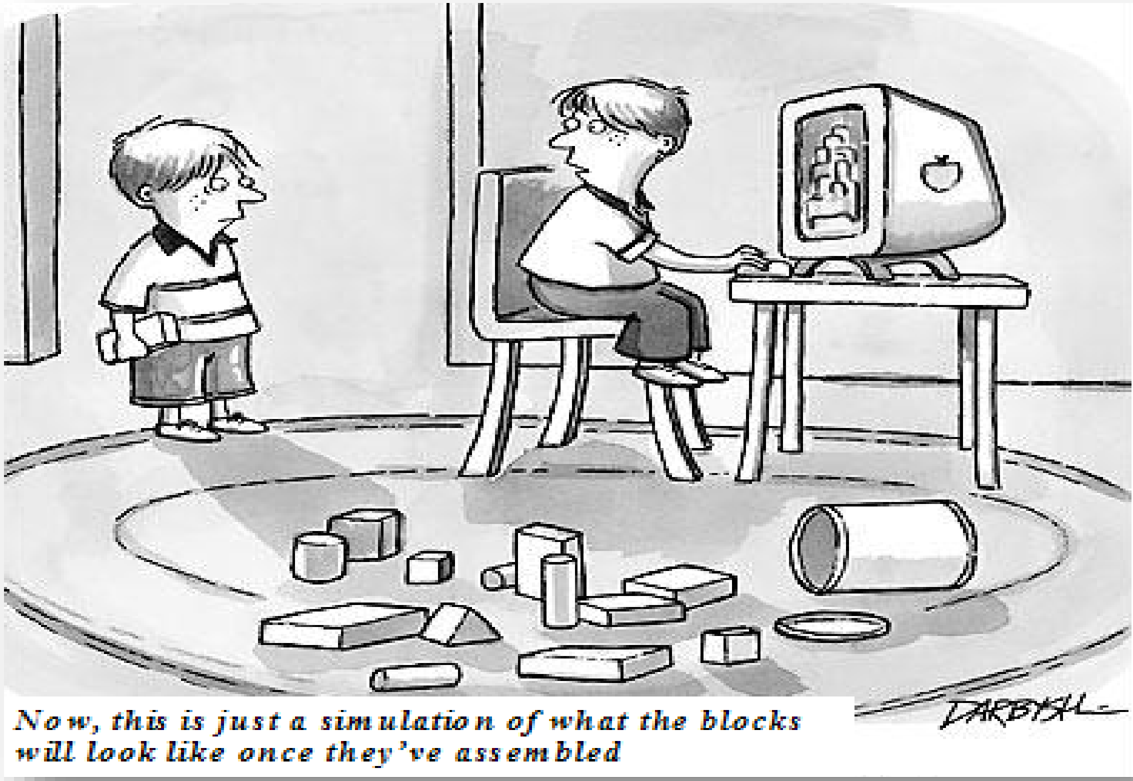
# Synchronization

- cudaDeviceSynchronize(no parameter)
  - halts execution on the host until all preceding commands in all CUDA streams have completed
    - **Halts execution of the host**
- cudaStreamSynchronize(stream_Id)
  - To synchronize the host with a specific stream, allowing other streams to continue executing on the device
    - **Halts execution of the host**
- cudaStreamWaitEvent()
  - takes a CUDA stream and an event as parameters and makes all the commands added to the given stream after the call to cudaStreamWaitEvent() delay their execution until the given event has completed.
    - **Halts execution within a stream**

# Next

- CUDA Streams by Example

# ECE569
# Module 43




- CUDA Streams by Example

# Streaming – Events

```
cudaEvent_t event;

cudaEventCreate (&event); // create event

cudaMemcpyAsync ( d_in, in, size, HostToDevice, stream1 );

cudaEventRecord (event, stream1);

cudaMemcpyAsync ( out, d_out, size, DeviceToHost, stream2 );

cudaStreamWaitEvent ( stream2, event );
// kernel launch waits till memcpy for s1 is completed
// actually kernel launch waits for both memcpy events!
myKernel<<< 1000, 512, stream2 >>> ( d_in, d_out );
// CPU function gets executed without waiting
CPUfunction ( blah, blahblah )
```

# Exercise

- **Which of the following CUDA API call can be used to perform an asynchronous data transfer?**
  - cudaMemcpy();
  - cudaAsyncMemcpy();
  - cudaMemcpyAsync();
  - cudaDeviceSynchronize();

# Exercise

- **What is the CUDA API call that makes sure that all previous kernel executions and memory copies in a device have been completed?**

  – __syncthreads()
  – cudaDeviceSynchronize()
  – cudaStreamSynchronize()
  – __barrier()

# Exercise

If each memory copy and kernel launch operations take 1 unit of time to complete, what is the **minimum** taken before the results are available on the host? Consider the **best case scenario**. Assume that device array is allocated through CudaMalloc and host array is allocated on pinned memory with cudaHostMalloc.

```
cudaStream_t s1, s2:
cudaStreamCreate(&s1); cudaStreamCreate(&s2);
```

Case 1

```
cudaMemcpy(&d_arr, &h_arr, numbytes, cudaHostToDevice);
A<<<1,128>>>(d_arr);
cudaMemcpy(&h_arr, &d_arr, numbytes, cudaDeviceToHost);
```

2

```
cudaMemcpyAsync(&d_arr, &h_arr, numbytes, cudaHostToDevice, s1);
A<<<1,128,s1>>>(d_arr);
cudaMemcpyAsync(&h_arr, &d_arr, numbytes, cudaDeviceToHost, s1);
```

3

```
cudaMemcpyAsync(&d_arr1, &h_arr1, numbytes, cudaHostToDevice, s1);
A<<<1,128>>>(d_arr1,s1);
cudaMemcpyAsync(&h_arr1, &d_arr1, numbytes, cudaDeviceToHost, s1);
cudaMemcpyAsync(&d_arr2, &h_arr2, numbytes, cudaHostToDevice, s2);
B<<<1,192>>>(d_arr2,s2);
cudaMemcpyAsync(&h_arr2, &d_arr2, numbytes, cudaDeviceToHost, s2);
```

# Exercise

If each memory copy and kernel launch operations take 1 unit of time to complete, what is the minimum taken before the results are available on the host? Assume that device array is allocated through CudaMalloc and host array is allocated on pinned memory with cudaHostMalloc.

```
cudaStream_t s1, s2:
cudaStreamCreate(&s1); cudaStreamCreate(&s2);
```

Case

1

```
cudaMemcpyAsync(&d_arr1, &h_arr1, numbytes, cudaHostToDevice, s1);
A<<<1,128,s2>>>(d_arr2);
```

2

```
cudaMemcpyAsync(&d_arr1, &h_arr1, numbytes, cudaHostToDevice, s1);
A<<<1,128,s2>>>(d_arr1);
```

# Streaming Example

- **We will set up multiple streams,**
  - Host memory >> GPU memory
- **We will split data into smaller chunks and stream into the GPU**
  - Similar to tiling on the gpu (global to shared memory)
  - But happens on the host!

# Parameterized Streaming based Vector Add

```
int main( void ) {
  cudaDeviceProp prop;
  int whichDevice;
  HANDLE_ERROR(cudaGetDevice( &whichDevice ) );
  HANDLE_ERROR(cudaGetDeviceProperties(&prop,whichDevice));
  if (!prop.deviceOverlap) {
    printf( “Can not handle overlaps\n" );
    return 0; }
cudaEvent_t start, stop;
float elapsedTime;
int stream_count = 4;
cudaStream_t stream[stream_count];
int *host_a, *host_b, *host_c;
int *dev_a[stream_count], *dev_b[stream_count], *dev_c[stream_count];
int i, k, Seglen = 1024;
int FULL_DATA_SIZE =1<<20;
cudaEventCreate( &start );
cudaEventCreate( &stop );
int blockSize = 256;
int Gridlen = (Seglen-1)/blockSize + 1;
```

# Memory Allocation

```
// allocate the memory on the GPU
for(i==0; i<stream_count; i++) {
    cudaStreamCreate( &stream[i] );
    cudaMalloc( (void**)&dev_a[i], Seglen * sizeof(int) );
    cudaMalloc( (void**)&dev_b[i], Seglen * sizeof(int) );
    cudaMalloc( (void**)&dev_c[i], Seglen * sizeof(int) );
 }
// allocate host locked memory, used to stream
cudaHostAlloc( (void**)&host_a, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault );
cudaHostAlloc( (void**)&host_b, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault );
cudaHostAlloc( (void**)&host_c, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault );
for (int i=0; i<FULL_DATA_SIZE; i++) {
    host_a[i] = rand();
    host_b[i] = rand();
}
cudaEventRecord( start, 0 );
```

# Stream Execution

```
for (int i=0; i<FULL_DATA_SIZE; i+= Seglen*stream_count) {
  for(k=0;k<stream_count;k++) {
    cudaMemcpyAsync( dev_a[k], host_a+i+k*Seglen, Seglen * sizeof(int),
                                        cudaMemcpyHostToDevice, stream[k] );
   cudaMemcpyAsync( dev_b[k], host_b+i+k*Seglen, Seglen * sizeof(int),
                                        cudaMemcpyHostToDevice, stream[k] );
   vecAdd<<<Gridlen,blockSize,0,stream[k]>>>(dev_a[k],dev_b[k],
                                                  dev_c[k],Seglen);
   } \\ end of k loop

   for(k=0; k< stream_count; k++)
       cudaStreamSynchronize(stream[k]);

   // copy the data from device to locked memory
   for(k=0; k< stream_count; k++)
      cudaMemcpyAsync( host_c+i+k*Seglen, dev_c[k], Seglen*sizeof(int),
                                  cudaMemcpyDeviceToHost, stream[k] );

} \\ end of i loop
cudaDeviceSynchronize();
```

# Collect Timing and Clean up

```
  cudaEventRecord( stop, 0 );
  cudaEventSynchronize( stop );
  cudaEventElapsedTime( &elapsedTime, start, stop ) );
  printf( "Time taken: %3.1f ms\n", elapsedTime );
  // cleanup the streams and memory
  cudaFreeHost( host_a );
  cudaFreeHost( host_b );
  cudaFreeHost( host_c );
  for (k = 0; k < 4; k++) {
    cudaFree( dev_a[k] );
    cudaFree( dev_b[k] );
    cudaFree( dev_c[k] );
    cudaStreamDestroy( stream[k] );
  return 0;
}
_global__ void vecAdd(float *in1, float *in2, float *out, int len) {
        int i = threadIdx.x + blockDim.x * blockIdx.x;
        if (i < len)    out[i] = in1[i] + in2[i];
}
```

# Common optimization techniques

- **Suggested Reading**
  - J. A. Stratton *et al*., "Algorithm and Data Optimization Techniques for Scaling to Massively Threaded Systems," in *Computer*, vol. 45, no. 8, pp. 26-32, August 2012.
  - J. A. Stratton *et al*., "Optimization and architecture effects on GPU computing workload performance," *2012 Innovative Parallel Computing (InPar)*, San Jose, CA, 2012, pp. 1-10.

# Next

- CUDA Streams with a real life application