CUDA Optimizations

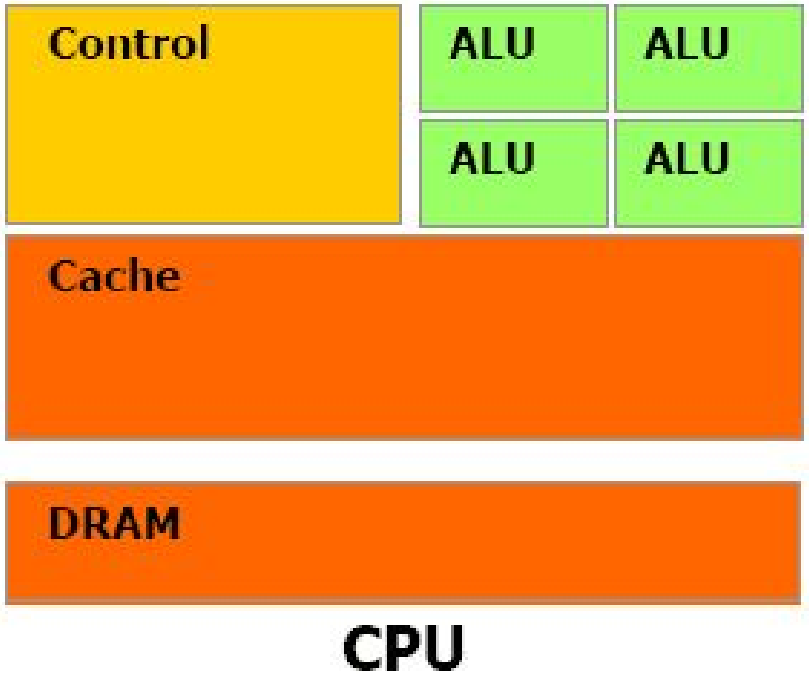
Advanced Research Computing

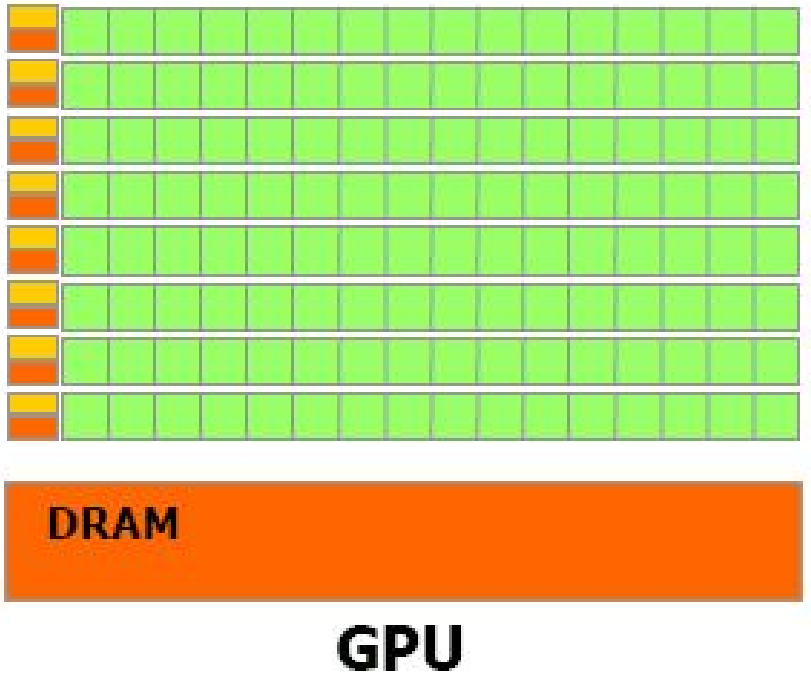
April 1, 2016

General Assumptions

- General working knowledge of CUDA
- Want kernels to perform better

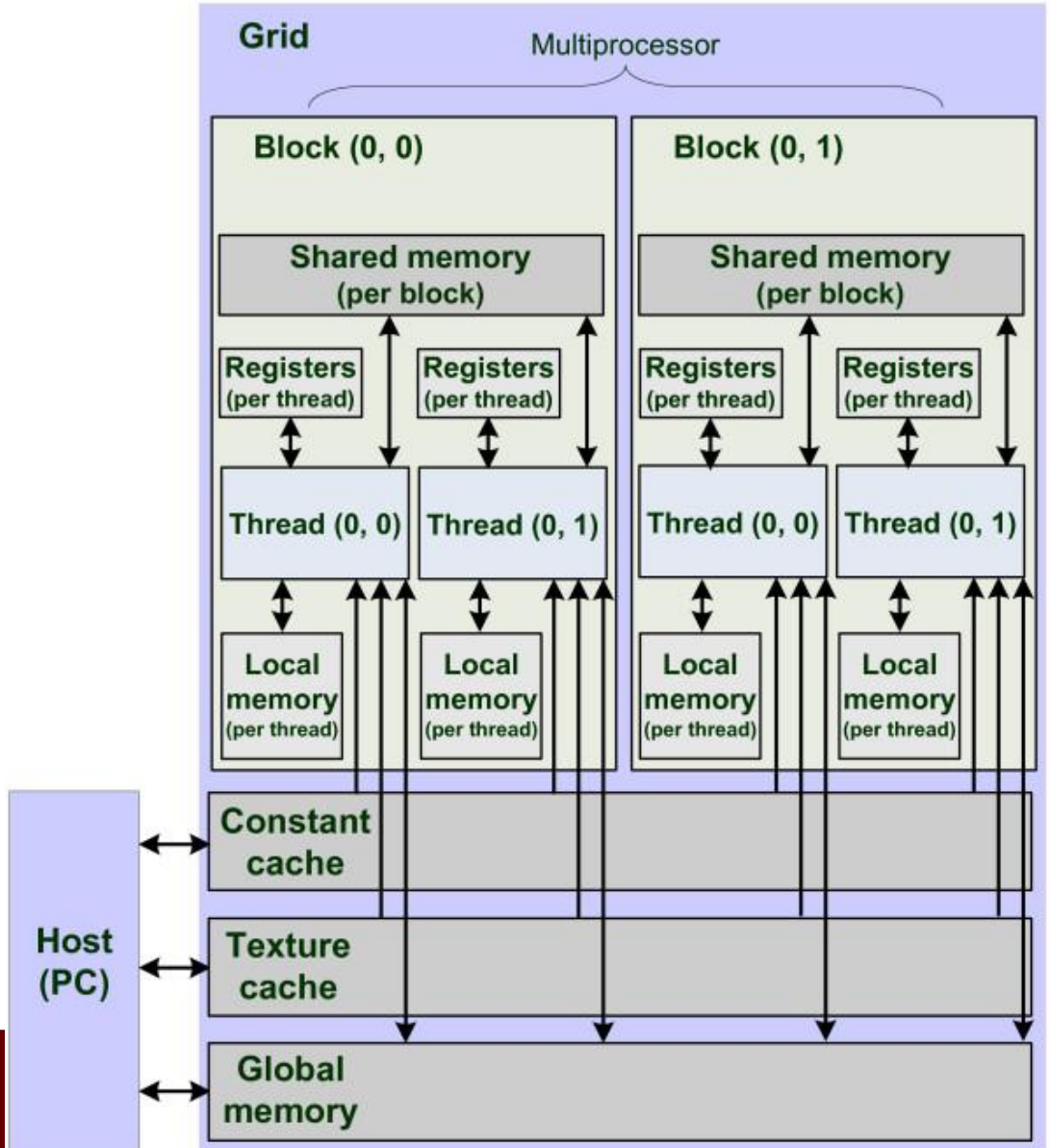
Refresher - CPU vs GPU





Refresher - Memory

Programmer is in charge of managing cache and locality



Profiling

- Before optimizing, make sure you are spending effort in correct location
- Nvidia Visual Profiler is run with nvvp
- This is a fantastic tool for optimizing performance
- Demo

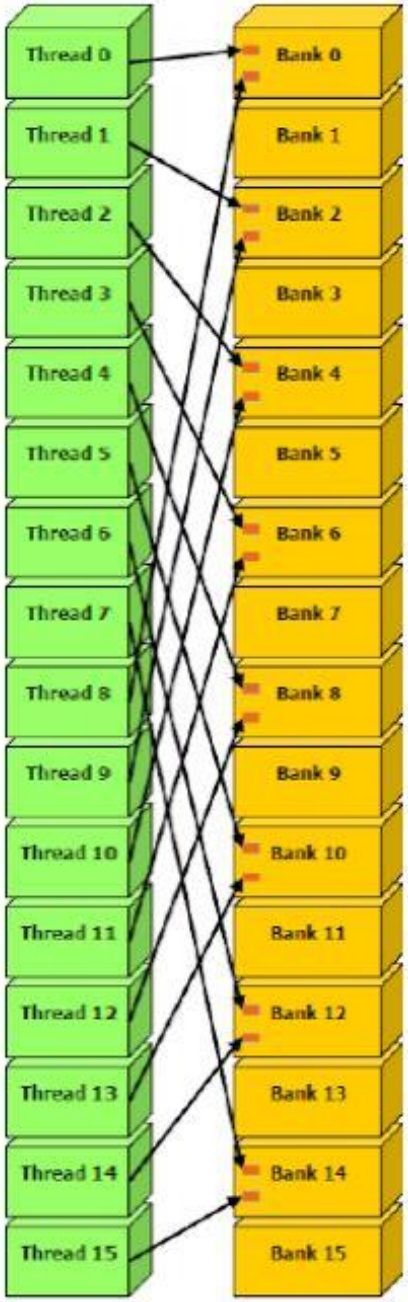
Global Memory First

- Before looking at instructions/Math, get the global memory accesses correct
- Comment out any logic in the program and make sure the memory accesses are coalesced and the throughput is where you expect
- This class focuses on optimizations after that point, but they won't be helpful if the memory access is your bottleneck

Occupancy

- Occupancy refers to the utilization of the CUDA cores.
- Trying achieve 100% occupancy is a good first goal, but is not always the best. See website in References.
- Look at occupancy spreadsheet from Nvidia site.

Shared Memory Bank Conflicts



Shared memory is divided into banks to allow each thread in a wrap access simultaneously. Each bank can service only one request at a time. The shared memory is interleaved by 32 bits or one float data type. The total number of banks is fixed at 32 for Compute 2.0 devices and later.

Shared Memory Bank Conflicts

Reading global memory into shared is a common task. The following may seem like a good way when thinking about CPU thread locality caching.

```
int tid = threadIdx.x;
shared[2*tid] = global[2*tid];
shared[2*tid + 1 ] = global[2*tid + 1];
```

The following is the more correct way considering banking issues and coalescing:

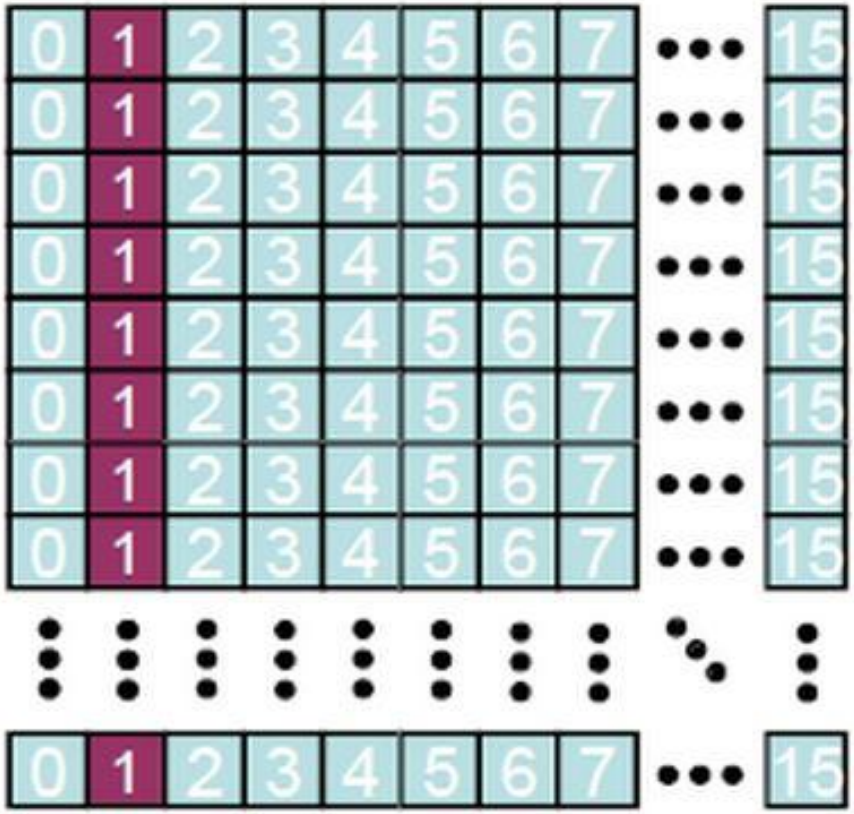
```
shared[tid] = global[tid];
shared[tid + blockDim.x] = global[tid + blockDim.x];
```

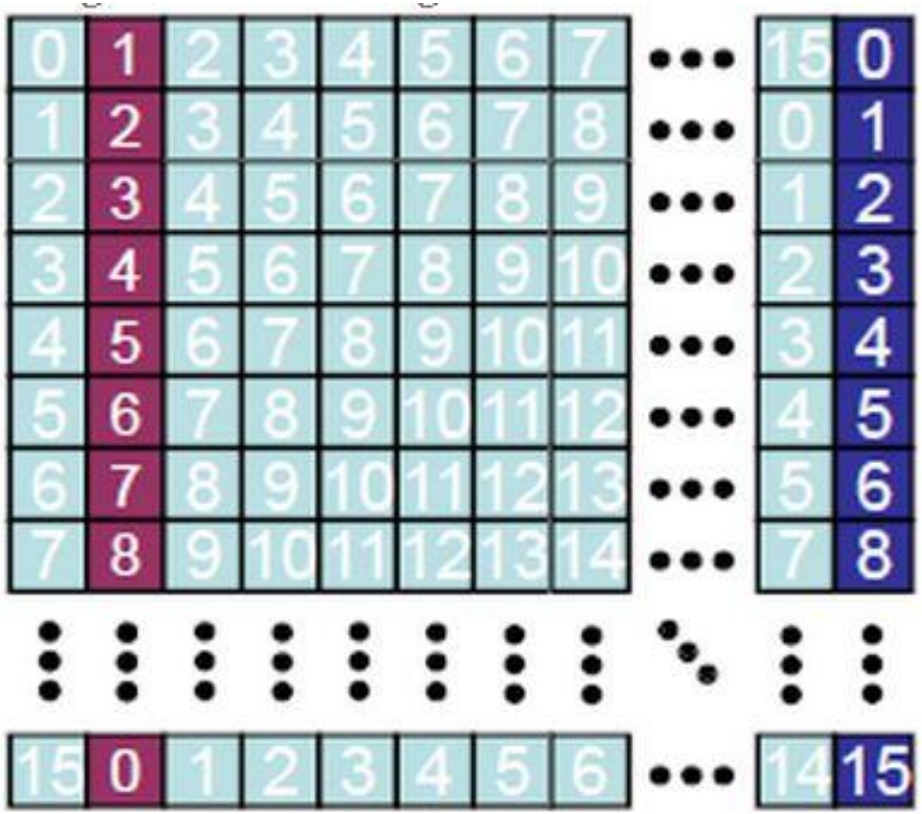
Shared Memory Bank Conflicts

Processing a 2D matrix with every thread working on a row

__shared__ int shared[TILE_WIDHT][TILE_HEIGHT] ;

__shared__ int shared[TILE_WIDHT][TILE_HEIGHT + 1]





Registers vs Shared Memory

- In compute devices prior to 2.0, registers were not much faster than shared memory. The documentation suggests just using shared memory
- In compute devices 2.0 +, the performance gap between registers and shared memory has increased significantly.
- To get the theoretical FLOPS of a device, values must be in registers.

Skip __syncthreads()

- All threads in a warp operate together. If you are only concerned with synchronizing the threads within a warp
- The is useful in reduction cases; see below:

```
__device__ void warpReduce(volatile int* sdata, int tid) {
    sdata[tid] += sdata[tid + 32];
    sdata[tid] += sdata[tid + 16];
    sdata[tid] += sdata[tid + 8];
    sdata[tid] += sdata[tid + 4];
    sdata[tid] += sdata[tid + 2];
    sdata[tid] += sdata[tid + 1];
}
```

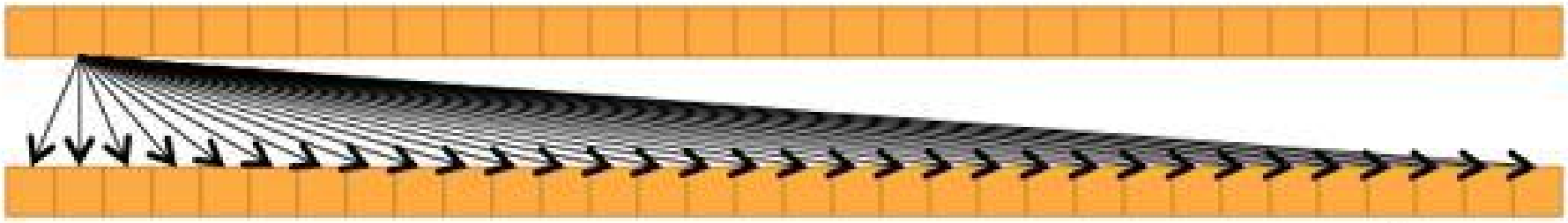
NOTE - the warp size may change in future devices

Shuffle

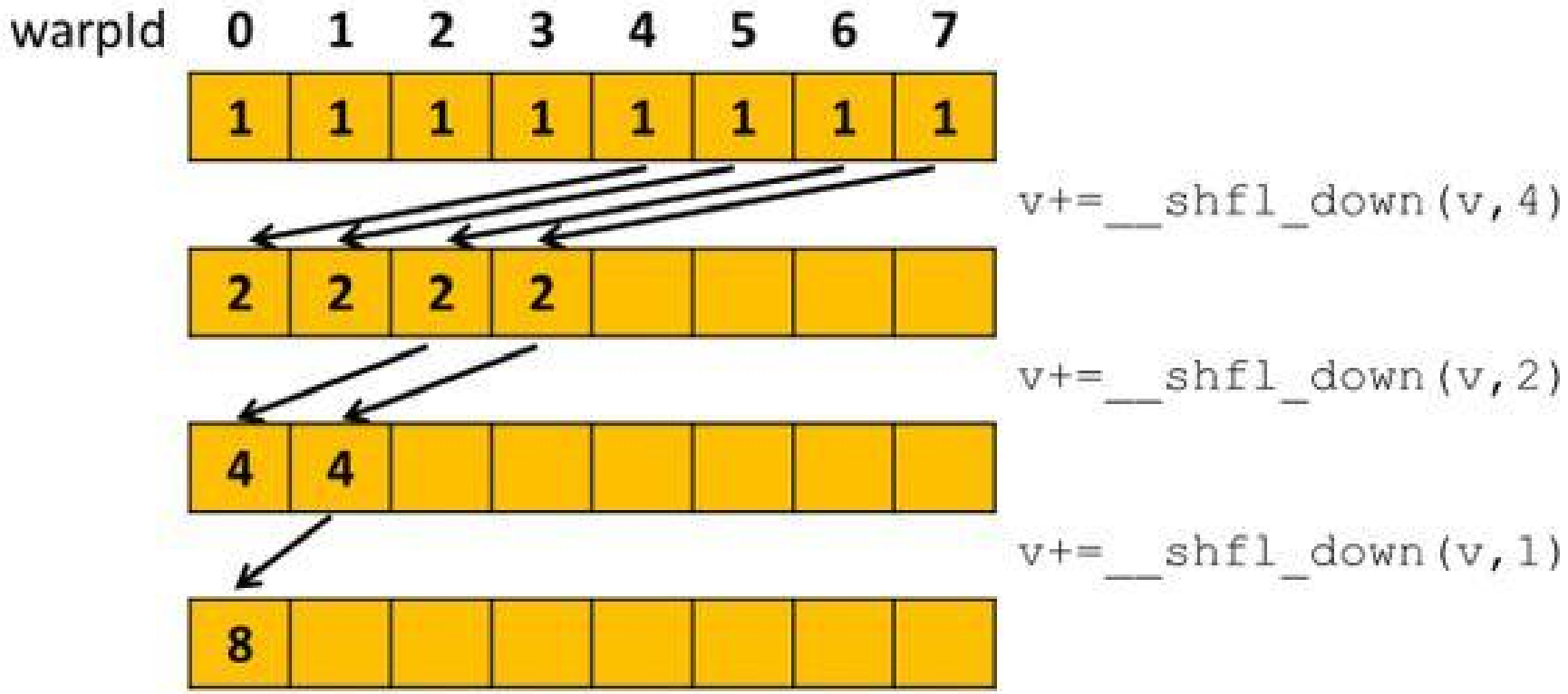
- Only available in Compute 3.0 + devices
- 3.0 devices have twice the shared memory bandwidth but 6x the number of CUDA cores
- Allows threads in a warp to share data faster than shared memory

```
float __shfl(  float var,     // Variable you want to read from source thread
    int srcLane,              // laneID of the source thread
    int width=warpSize        // Division of warp into segments of size
```

`__shfl(var,1)`



Reduce with Shuffle



All threads will be shifting values even though they are not needed in the reduction. Only needed shifts shown.

Recalculate vs Lookup Table

When optimizing CPU code, lookup tables are common. For example, if you only need 8192 distinct values of sine & cosine, a lookup table is generally faster. Lookup tables exploit the CPU cache hierarchy. But on the GPU there is little cache and lots of compute.

Example:
Recalculating the window for a triangle filter is faster than reading from memory

```
for (INT32 j = 0; j < filterLen; j++) {
        INT32 scaleI = ((j < (filterLen / 2)) ? (j + 1) :
(filterLen - j));
        sum += smem[tid + j] * (FLOAT32)scaleI;
}
```

Templates

Branching is expensive. Templates can be used to make code more general while removing unneeded branching

```
Template <unsigned int blockSize>
__device__ void warpReduce(volatile int* sdata, int tid) {
if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
}
warpReduce<blockSize>(sdata, tid);
```

Note all items in red are evaluated at compile time

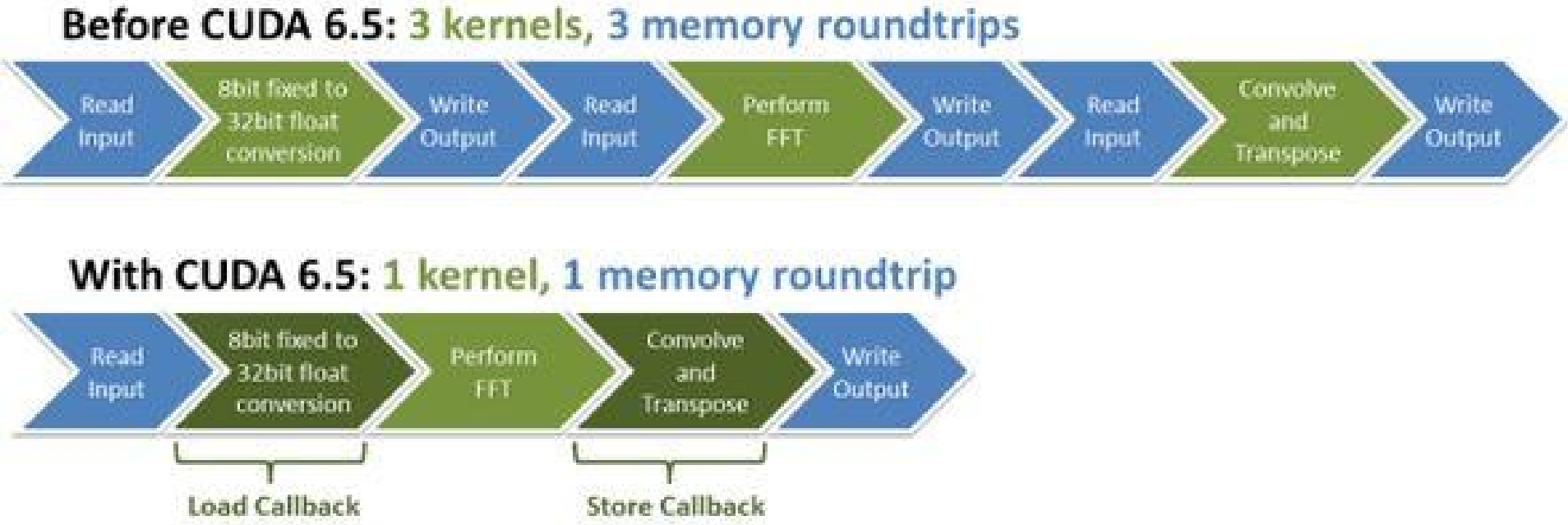
Instruction Level Parallelism

- The scheduler can issue multiple instructions if they are independent. This is another way to hide memory latency.
- Compute devices 3.0 + require ILP to get theoretical FLOPS. Older devices still benefit.

```
for( int i = 0; i < N_ITERATIONS; i++ )
{
    a = a * b + c;
    d = d * b + c;
}
```

Device Callbacks

- New in CUDA 6.5
- Callback routines can be specified for loading and storing data during FFT operations.



References

- http://cuda-programming.blogspot.com/2013/02/bank-conflicts-in-shared-memory-in-cuda.html
- http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf
- http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf
- http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-use-cufft-callbacks-custom-data-processing/
- http://acceleware.com/blog/keplers-shuffle-instruction
- http://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/

Instruction Level Parallelism

- The scheduler can issue multiple instructions if they are independent. This is another way to hide memory latency.
- Compute devices 3.0 + require ILP to get theoretical FLOPS. Older devices still benefit.

```
for( int i = 0; i < N_ITERATIONS; i++ )
{
    a = a * b + c;
    d = d * b + c;
}
```

Questions

???

TAKE THE NLI SURVEY