**Device Management**

**Context Management**

**Module Management**

**Execution Control**

**Memory Management**

**Texture Reference Management**

**Interoperability with OpenGL and D3D**
Texture Reference Management

- Texture: 1D, 2D, or 3D pixel (texel) data.
- Texture is accessed using `texture references`:
  ```
texture<Type, Dim, ReadMode> texRef;
  ```
  - **Type**: must be either integer or single FP, or any 1-, 2-, or 4- component vector of these two types.
  - **Dim**: 1, 2, or 3
  - **ReadMode**: either `cudaReadModeNormalizedFloat` or `cudaReadModeElementType`.
Host Runtime for Textures

- Texture Reference Structure:

```c
struct textureReference {
    int normalized;
    enum filterMode;
    enum addressMode[3];
    struct channelDesc;
};
```

- Examples:

```c
texRef.normalized = true;
texRef.filterMode = cudaFilterModeLinear;
texRef.addressMode[0] = cudaAddressModeWrap;
```
Host Runtime for Textures

- Bind texture reference to a CUDA array:
  ```c
  cudaBindTextureToArray(texRef, cuArray);
  ```
- Can also bind texture reference to linear memory: but with several limitations.
Device Runtime for Textures

- Texturing from CUDA arrays using tex fetches:

  Type tex1D(texRef, float x);
  Type tex2D(texRef, float x, float y);
  Type tex3D(texRef, float x, float y, float z);
Texture Reference Management

• Advantages of using Textures:
  • Cached (high bandwidth if locality preserved)
  • Not subject to memory coalescing pattern
  • 'Free' linear interpolation (only valid for FP)
  • Normalized texture coordinates (resolution independent)
  • Addressing mode (automatic handling of out of boundary cases)
Example: Parallel Reduction

- Compute: \[ S = \sum_{i=1}^{N} a_i \]

- Put data in shared memory:
  - Fast (remember that shared memory can be treated as a user managed L1 cache)
  - Allows block-wise synchronization
Example: Parallel Reduction

- But there is no global synchronization
  - Split computation into blocks
  - A single block aggregates the final result
Example: Parallel Reduction

- First try: (consider a single block)
Example: Parallel Reduction

```c
__global__ void reduce0(int *g_idata, int *g_odata) {
  extern __shared__ int sdata[];

  // each thread loads one element from global to shared mem
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  sdata[tid] = g_idata[i];
  __syncthreads();

  // do reduction in shared mem
  for(unsigned int s=1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0) {
      sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
  }

  // write result for this block to global mem
  if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```
Example: Parallel Reduction

```c
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // do reduction in shared mem
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Problem!
(divergent in a warp; Shared memory bank conflict)
Example: Parallel Reduction

- Second try: (consider a single block)
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // do reduction in shared mem
    for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
        if (tid < s) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
Example: Parallel Reduction

```c
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // do reduction in shared mem
    for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
        if (tid < s) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Idling threads?
Quantitative Analysis of Parallel Algorithms

- How much speedup do I gain by going parallel?
- Does the parallel program fully utilize resources?
- Efficiency vs. work size
- ...
- ...
Source of Overhead in Parallel Algorithms

- Throwing in N processors doesn't mean your program runs N times faster
  - Interprocess communication
  - Idling due to load imbalancing, synchronization etc, serial sub-components
  - Excess computation
    - Sometimes you are forced to use a poorer but easily parallelizable algorithm
Performance Metrics

- Execution time: time elapsed between the beginning and end of its execution.
  \[ T_S : \text{serial runtime} \]
  \[ T_P : \text{parallel runtime on } p \text{ processors} \]

- Total Parallel Overhead
  \[ T_O = pT_P - T_S \]
  overhead is zero if perfect linear speedup achieved
Performance Metrics

- Speedup: $S = \frac{T_S}{T_P}$
- Example: speedup of parallel reduction?
Performance Metrics

- Speedup: 
  \[ S = \frac{T_S}{T_P} \]

- Example: speedup of parallel reduction? 
  \[ T_S = O(n) \]

For \( p \) multiprocessors:

\[ T_P = O(\log p) \]

Overall:

\[ T_P = O\left(\frac{n}{p} \log p\right) \]
Performance Metrics

- Theoretical maximum speedup: $S = p$
  - In practice, much less (refer to Amdahl's law)
- Is it possible to exceed $p$?
Performance Metrics

- Theoretical maximum speedup: \( S = p \)
  - In practice, much less (refer to Amdahl's law)
- Is it possible to exceed \( p \)?
  - Superlinear speedup
- Sources of superlinearity:
  - Hardware limitation that puts serial version at a disadvantage
  - Superlinearity due to exploratory decomposition
Performance Metrics

- Efficiency: \[ E = \frac{S}{p} = \frac{T_S}{pT_P} \]

  Measures how efficiently an algorithm is utilizing all the processors.

- Ideal efficiency: \( E = 1 \)

- Efficiency of parallel reduction?
Performance Metrics

- **Efficiency:**
  \[ E = \frac{S}{p} = \frac{T_S}{pT_P} \]

  Measures how efficiently an algorithm is utilizing all the processors.

- **Ideal efficiency:** \( E = 1 \)

- **Efficiency of parallel reduction?**
  \[ E = O\left(\frac{1}{\log p}\right) \]

  Hmm, not very efficient…
Performance Metrics

- **Cost:** \( C = p T_p \)
  Reflects the sum of time that each processor spends solving the problem.

- Compare the cost \( p T_p \) with the execution time of the fastest known sequential algorithm \( T_S \):
  - **Cost-optimal** if they are asymptotically equal.
  - Cost of parallel reduction?
Performance Metrics

- **Cost:** \( C = p T_p \)

  Reflects the sum of time that each processor spends solving the problem.

- Compare the cost \( p T_p \) with the execution time of the fastest known sequential algorithm \( T_s \):
  - **Cost-optimal** if they are asymptotically equal.
  - Cost of parallel reduction?

\[
p T_p = O(n \log p) \quad \text{NOT cost-optimal!}
\]

\[
T_s = O(n)
\]
Performance Metrics

- Revisit parallel reduction:
  How about we let each processor add $\frac{n}{p}$ elements, then sum up the resulting $p$ numbers?
Performance Metrics

- Revisit parallel reduction:

  How about we let each processor add $\frac{n}{p}$ elements, then sum up the resulting $p$ numbers?

  $$T_p = O\left(\frac{n}{p} + \log p\right)$$

  $$pT_p = O\left(n + p\log p\right)$$

  which is approximately $O(n)$ when $p \log p \ll n$

  Cost-optimal!