Harness the Power of GPUs:
An Introduction to GPGPU Programming
Lecture 5: Using Shared Memory; Atomic Operations

Guido Juckeland
Visiting Scholar
Technische Universität Dresden, Germany

June 18, 2014
Shared memory

Kepler GPUs have up to 48kB of shared memory per SMX.
What: Memory for fast data exchange or common access within a thread block.
How: Declared as `__shared__ float commondata[threadsPerBlock]` in the kernel.
Who: All threads of one thread block.
Lifetime: data lives as long as the thread block lives.
Access: read and write.
Using shared memory efficiently is the key to high performance!
Allocating fixed-size shared memory

```c
__global__ void foo()
{
    __shared__ int a[256];
    ...
}
```

- Allocates an array `a` of 256 elements to be shared among all threads in the thread block \( \rightarrow \) all threads access the same data
- Size is set up in the kernel
Dynamically sized shared memory

```c
__global__ void foo(){
    extern __shared__ int a[];
    ...
}

int main(...){
    ...
    foo<<< NBLOCKS, NTHREADS, NTHREADS*sizeof(int)>>>();
    ...
}
```

- Creates an array `a` with `NTHREADS` elements in shared memory for every thread block
- Can only be done for one array
Thread barriers for synchronization

• Barriers are used to ensure all threads (warps) have reached the same point in the kernel
• __syncthreads
• All threads need to reach __syncthreads
• Every __syncthreads is unique, no guarantee that two will work in combination

  – Don’t do this:
    ```
    if (some test){
      doSomething();
      __syncthreads;
    }else{
      __syncthreads;
      doSomethingElse();
    }
    ```

  Do it like this:
  ```
  if (some test){
    doSomething();
    __syncthreads;
    if (!(some test)){
      doSomethingElse();
    }
  ```
Filling shared memory

__global__ void foo(int *b_gpu){

__shared__ int a[256];
idx=... //get global index
...
// every thread copies one b to a
a[threadIdx.x]=b[idx];

// wait for all threads in a block
__syncthreads;
...
Typical usage of shared memory

```c
__global__ void foo(int *b_gpu){
    __shared__ int a[256];
    idx=... //get global index

    ... while (still_work_left) {
        // copy reused date to shared memory
        // syncthreads
        // compute
        // syncthreads
        // write back to global memory
        // if necessary syncthreads
    }
}
```
Example: Blocking matrices

Original access pattern

Blocked access pattern

Copy data to shared memory

Carry out multiplication with data from shared memory

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2010
ECE 408, University of Illinois, Urbana-Champaign
Code example: blocking matrices

const int blocksize=16; // assuming 16x16 blocks
const int N=256; // that makes 256 threads per block, using N for 1D matrix addressing

// only works for squared blocks and grids
__global__ void mm_gpu(float *a, float *b, float *c){
    const int cx=blockIdx.x*blockDim.x+threadIdx.x; // get my position in x
    const int cy=blockIdx.y*blockDim.y+threadIdx.y; // get my position in y
    const int tx=threadIdx.x; // get my local x
    const int ty=threadIdx.y; // get my local y
    __shared__ float as[blocksize][blocksize]; // set up shared a
    __shared__ float bs[blocksize][blocksize]; // set up shared b
    float c_temp=0.0f; // register to add up my c result

    // loop over blocks
    for (int l=0; l<gridDim.x; l++)
    {
        // copy data to shared mem
        as[ty][tx]=a[cy*N+l*blocksize+tx]; // select row with cy*N, add column by blocksize*l+tx
        bs[ty][tx]=b[(l*blocksize+ty)*N+cx]; // select row with (l*blocksize+ty)*N, column=cx
        __syncthreads();
    }
}
Reductions

- Used to aggregate values into one location
- Typical examples: global sum, min, max

Easy (but wrong) solution:

```c
__global__ void sum(float *sum, float *add) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    *sum += add[tid];  //Denkfehler
}
```

- Execution 1: sum=7; A reads sum; A adds 1; A writes sum; sum=8; B reads sum; B adds 1; B writes sum; **sum=9**
- Execution 2: sum=7; A reads sum; B reads sum; A adds 1; A writes sum; sum=8; B adds 1; B writes sum; **sum=8**
Parallel reduction

- Recursively halve # threads, add two values per thread
- Takes $\log(n)$ steps for $n$ elements, requires $n/2$ threads

- Assume an in-place reduction using shared memory
  - The original vector is in device global memory
  - The shared memory used to hold a partial sum vector
  - Each iteration brings the partial sum vector closer to the final sum
  - The final solution will be in element 0
__shared__ float partialSum[]

// copy my chunk into partialSum
unsigned int t = threadIdx.x;
for (unsigned int stride = 1; stride < blockDim.x;
    stride *= 2)
{
    __syncthreads();
    if (t % (2*stride) == 0)
        partialSum[t] += partialSum[t+stride];
}
Simple implementation visualized

Thread 0  Thread 2  Thread 4  Thread 6  Thread 8  Thread 10

0  1  2  3  4  5  6  7  8  9  10  11

1  0+1  2+3  4+5  6+7  8+9  10+11

2  0...3  4...7  8...11

3  0...7  8...15

iterations  Array elements

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2010
ECE 408, University of Illinois, Urbana-Champaign
Why is that a bad implementation?

• In each iterations, two control flow paths will be sequentially traversed for each warp
  – Threads that perform addition and threads that do not
  – Threads that do not perform addition may cost extra cycles depending on the implementation of divergence

• No more than half of threads will be executing at any time
  – All odd index threads are disabled right from the beginning!
  – On average, less than ¼ of the threads will be activated for all warps over time.
  – After the 5th iteration, entire warps in each block will be disabled, poor resource utilization but no divergence.
    • This can go on for a while, up to 4 more iterations (512/32=16= 2^4), where each iteration only has one thread activated until all warps retire
Better implementation

```c
__shared__ float partialSum[]
// copy my chunk into partialSum
unsigned int t = threadIdx.x;
for (unsigned int stride = blockDim.x; stride > 1;
    stride >>= 1)
{
    __syncthreads();
    if (t < stride)
        partialSum[t] += partialSum[t+stride];
}
```
No divergence until <16 subsums
Using atomic operations

```c
__global__ void sum(float *sum, float *add) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    atomicAdd( sum, add[tid] ); //does all the magic
}
```

- Atomic operations ensure a locked read, op, and write operation per thread
- “Serialize” execution of all threads
- Atomics improved in every CUDA version
- Atomics now faster than custom reductions
Notes on atomics

- List of atomic functions:
  - atomicAdd()
  - atomicSub()
  - atomicExch()
  - atomicMin()
  - atomicMax()
  - atomicInc()
  - atomicDec()
  - atomicCAS()
  - atomicAnd()
  - atomicOr()
  - atomicXor()

- Work both in global and shared memory
- Function calls return the original value
- Implemented for type int
- Only atomicAdd() and atomicExch() also work with floats
Build what you need

```c
__device__ double atomicAdd(double* address, double val) {
    unsigned long long int* address_as_ull = (unsigned long long int*)address;
    unsigned long long int old = *address_as_ull, assumed;
    do {
        assumed = old;
        old = atomicCAS(address_as_ull, assumed,
                        __double_as_longlong(val +
                        __longlong_as_double(assumed)));
    } while (assumed != old);
    return __longlong_as_double(old);
}
```

<http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions>
QUESTIONS?