Lecture No.5: Advanced Parallel Reduction

Muhammad Osama Mahmoud, TA

Computer and Systems Eng. Dept. - Faculty of Engineering – University of Minia

Advanced Parallel Reduction #1

- The number of arithmetic instructions compared to loop and addressing instructions is low
- This instruction overhead can be evaded using loop unrolling
- A full loop unrolling on the reduction loop is considered
- After each reduction step the number of idle threads increases
- To minimize thread idle time, make each thread adds two elements from two different blocks and load to shared memory

Advanced Parallel Reduction #1 (Cont.)

```
__global__ void reduce_v3(int * inp_data, int * outp_data) {
    extern  __shared__  int  sh_data[ ]; //dynamically locate shared memory at kernel launch
    unsigned int  tx   = threadIdx.x;
    unsigned int  idx = blockIdx.x * (blockDim.x * 2) +  threadIdx.x;
    sh_data[tx]  =  inp_data[idx] + inp_data[idx + blockDim.x]; // perform first add step then load to shared memory
    __syncthreads();
    // perform full loop unrolling
    if (blockDim.x >= 512 &&  tx < 256) {
        if (tx < 256) { sh_data[tx] += sh_data[tx + 256]; } __syncthreads();
    }
    if (blockDim.x >= 256  &&  tx < 128) {
        if (tx < 128) { sh_data[tx] += sh_data[tx + 128]; } __syncthreads();
    }
    if (blockDim.x >= 128 &&  tx < 64) {
        if (tx < 64) { sh_data[tx] += sh_data[tx + 64]; } __syncthreads();
    }
// rest of code in the next slide
```

Advanced Parallel Reduction #1 (Cont.)

```
    // perform loop unrolling on the last warp
    if (tid < 32) {
        if (blockDim.x >= 64 && tx < 32) sh_data[tx] += sh_data[tx + 32];
        if (blockDim.x >= 32 && tx < 16) sh_data[tx] += sh_data[tx + 16];
        if (blockDim.x >= 16 && tx < 8) sh_data[tx] += sh_data[tx + 8];
        if (blockDim.x >= 8 && tx < 4) sh_data[tx] += sh_data[tx + 4];
        if (blockDim.x >= 4 && tx < 2) sh_data[tx] += sh_data[tx + 2];
        if (blockDim.x >= 2 && tx < 1) sh_data[tx] += sh_data[tx + 1];
    }

    if (tx == 0) outp_data[blockIdx.x] = sh_data[tx];
} // kernel end
```

Parallel Reduction Complexity

- $Log(N)$ sequential strides or steps, each stride performs 2^{stride} operations, that is, N independent operations
- With P threads physically initiated in parallel, time complexity is $O(N/P\ +\ log\ N)$
- If $P\ =\ N$ then complexity is $O(1 + log\ N) = O(log\ N)$
- The performance speed-up will be s = $\frac{T_{serial}}{T_{parallel}} = \frac{N}{log\ N}$ times faster than the sequential algorithm

Parallel Reduction Cost

- Cost of parallel algorithm = # of processors (threads) x time complexity
- In case of parallel reduction, cost $= N \times \mathrm{O}(log\ N) = \mathrm{O}(N \log N)$, not cost efficient
- Brent's theorem states that each processor should do $\mathrm{O}\ (\log N)$ sequential operations
- If applied to the reduction kernel, the number of threads needed is $\mathrm{O}\ (\mathrm{N}/\log N)$, and the cost will be$\mathrm{O}\ (\mathrm{N}/\log N) \times \mathrm{O}(log\ N) = \mathrm{O}(N)$, which is cost efficient

Advanced Parallel Reduction #2

- To apply Brent's theorem, we replace and modify the following code block from the last kernel

```
sh_data[tx]  =  inp_data[idx] + inp_data[idx +
blockDim.x];
```

with a loop to add and load as many elements as possible

```
unsigned int gridSize = + blockDim.x * 2 * gridDim.x;
sdata[tx] = 0;

while (i < n) {
	sh_data[tx] += inp_data[idx] + inp_data [idx + blockDim.x ];
	idx += gridSize;
}
__syncthreads();
```

References

[1] Wen-mei W. Hwu, “Heterogeneous Parallel Programming”. Online course, 2014. Available: https://class.coursera.org/hetero-002

[2] M. Harris, “Optimizing Parallel Reduction in CUDA”, Oct. 2007.