
Massively Parallel Computing with Cuda

- Complexity and Profiling -

Hendrik Lensch
Robert Strzodka

Today

- **Last lecture**
 - N-body problems
- **Today**
 - Complexity and Profiling

Speedup

- **relative to best known sequential algorithm S**

$$\frac{t_S(n)}{t_A(n)}$$

- **absolute speedup for a given number of processors**

$$SU_{abs} \frac{t_S(n)}{t_A(p, n)} \leq p \frac{t_S(n)}{c_A(n)} \leq p$$

- **relative speedup**

$$\frac{t_A(1, n)}{t_A(p, n)}$$

What influences the speed-up?

- **Complexity of the implemented algorithm**
- **Computational intensity**
- **Selection of #blocks / #threads per block**
- **Resources per block**
- **Branch divergence**
- **Load balancing**
- **Communication with CPU**
- **Bandwidth**
- **Memory Access Patterns**
- **(Loop-unrolling)**

Complexity of the Implemented Algorithm

Brent's Theorem:

- An PRAM algorithm A which runs in $t_A(n)$ time steps and performs $w_A(n)$ work can be implemented to run on a p-processor PRAM in

$$O\left(t_A + \frac{w_A(n)}{p}\right)$$

- Don't go for a $O(N^2)$ algorithm if $O(N \log N)$ is around the corner.
- Keep an eye on the non-parallelizable work

Computational Intensity

- **Maximize**
instructions / # data item
- **A vector operation will only be that fast**
 - (# instructions ~ # data item)
- **Re-use data in shared memory if required by multiples threads**

Selection of #blocks / #threads per block

- **Keep all SP's busy**
-> **#blocks \geq #SPs**
- **Allow for enough re-scheduling to hide latencies due to memory access**
-> **> 16 threads per SPs**

Resources per Block

- **Shared Memory** limits the number of threads per block
- **Local memory and number of registers** can influence the number of active blocks

Branch Divergence

- ... as talked about in length ...

Load Balancing

- **Avoid getting dominated by the longest running block**
- **Especially true in applications where the data layout is flexible**
 - sorting
 - adaptive grids
 - n-body problems

Communication to the CPU

- **Memcpy is a serious bottleneck**
- **Single-threaded execution on the GPU rather than download to the CPU.**
- **Use page-locked memory for fast up/down load**
- **Use streams for continuous processing**

Bandwidth

- **recomputing instead of loading and storing**

Memory Access Patterns

- **If the memory transaction cannot be coalesced, then a separate memory transaction will be issued for each thread in the half-warp**
- **from the Programming Guide**
 - 32-bit data types will be roughly 10x slower
 - 64-bit data types will be roughly 4x slower
 - 128-bit data types will be roughly 2x slower

Blocking

- **Use of atomics?**
 - shared memory atomics
 - global memory atomics

Loop-unrolling

- (... unfortunately more than syntactic sugar.)
- **Ugly but possible to implement**
 - Use constant loop sizes
 - templates for operations on different sizes.

1000 Knobs – Which to turn?

- **A slow Cuda-program can have a lot of different causes.**
- **Use *profiling* to determine where the bottleneck really is.**
- **Cuda Visual profiler is easy to use**
- **... but unfortunately does not drill down the kernel**

Profiling Example

Halton Sequence

- a simple pseudo-random sequence
- generating numbers in the range (0,1)
- reverse the digit in a given base: e.g. 2:
 - $1 = 1.0 \Rightarrow 0.1 = 1/2$
 - $2 = 10.0 \Rightarrow 0.01 = 1/4$
 - $3 = 11.0 \Rightarrow 0.11 = 3/4$
 - $4 = 100.0 \Rightarrow 0.001 = 1/8$
 - $5 = 101.0 \Rightarrow 0.101 = 5/8$
 - $6 = 110.0 \Rightarrow 0.011 = 3/8$
 - $7 = 111.0 \Rightarrow 0.111 = 7/8$
 - ...

Halton Sequence (2)

```
float Halton( int I, int base) {
    float H = 0;
    float scale = 1./base;
    while ( I > 0 ) {
        int digit = I % base;
        H = H + digit * scale;
        I = ( I - digit ) / base;
        scale /= base;
    }
    return H;
}
```

AtomicAdd for Floats

- **as atomicAdd is only defined for integers we have to find a work around**
- **simply store the float in an integer**
- **need to apply addition explicitly**
- **requires at least a second access to the memory location**

AtomicAdd for Floats (2)

```
__device__ inline void atomicFloatAdd(float *address, float
val)
{
    int tmp0 = *address;
    int i_val = __float_as_int(val + __int_as_float(tmp0));
    int tmp1;

    // compare and swap v = (old == tmp0) ? i_val : old;
    // returns old
    while(
        (tmp1 = atomicCAS((int *)address, tmp0, i_val)) != tmp0)
    {
        tmp0 = tmp1;
        i_val = __float_as_int(val + __int_as_float(tmp1));
    }
}
```

Rule of Thumb

Perhaps in decreasing order of importance one should make sure that one's code (see the programming guide for discussions of all of these):

- **Uses all of the card: has a multiple of 32 threads per block and has at least as many blocks as multiprocessors,**
- **Accesses global memory properly,**
- **Avoids shared memory bank conflicts,**
- **Has as few branching conditional loops as possible,**
- **Has small loops unrolled,**
- **Has no unnecessary `__syncthreads()` calls in it.**