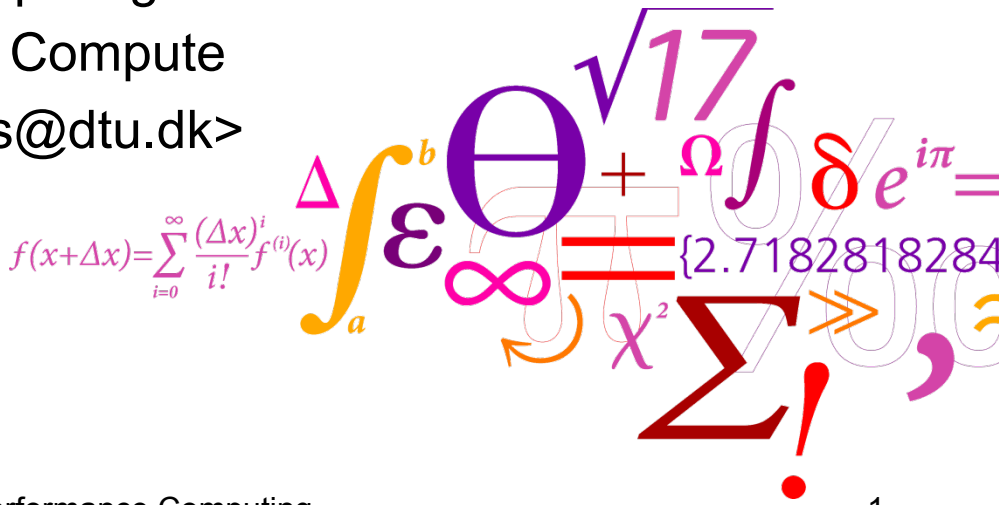
CUDA Performance Tuning Advanced Topics


Hans Henrik Brandenborg Sørensen
DTU Computing Center
DTU Compute
<hhbs@dtu.dk>

DTU DCC Computing Center

GPUlab DTU Informatics



Overview

- Thread divergence
- Warp shuffles
  - Four variations
  - Intra-warp reduction
- Reduction operation in CUDA
  - Warp reduction kernel
  - Block reduction kernel
  - Atomic operations
- Final tricks & tips

Thread divergence

Thread divergence

- `if`, `else`, `do`, `for`, and `switch` can significantly affect the instruction throughput by causing threads in a warp to take different execution paths
  - Less efficient SIMT execution / no coalescing …

Thread divergence

- `if`, `else`, `do`, `for`, and `switch` can significantly affect the instruction throughput by causing threads in a warp to take different execution paths
  - Less efficient SIMT execution / no coalescing …
- Worst case (every warp diverges)

```
if (threadIdx.x % 2 == 0) { ... } else { ... }
```

Thread divergence

- `if`, `else`, `do`, `for`, and `switch` can significantly affect the instruction throughput by causing threads in a warp to take different execution paths
  - Less efficient SIMT execution / no coalescing …
- Worst case (every warp diverges)

```
if (threadIdx.x % 2 == 0) { ... } else { ... }
```

- Better (some warps diverges)

```
if (threadIdx.x < 30) { ... } else { ... }
```

Thread divergence

- `if`, `else`, `do`, `for`, and `switch` can significantly affect the instruction throughput by causing threads in a warp to take different execution paths
  - Less efficient SIMT execution / no coalescing …
- Worst case (every warp diverges)

```
if (threadIdx.x % 2 == 0) { ... } else { ... }
```

- Better (some warps diverges)

```
if (threadIdx.x < 30) { ... } else { ... }
```

- Good (no warps divergent)

```
if (threadIdx.x < M * warpsize) { ... } else { ... }
```

Warp shuffles

Warp shuffles

- Warp shuffle intrinsics are the fastest way of moving data between threads in the same warp
- Threads within a warp are referred to as *lanes*
  - Indexed between `0` and `warpSize-1` (inclusive)
- The exchange occurs simultaneously for all *active* threads within the warp (given by a mask)
  - 4 or 8 bytes of data per thread depending on the type
- Threads may only read data from another thread which is actively participating in the shuffle
  - If target lane is inactive, the retrieved value is undefined

Warp shuffles

- Since CUDA 9.0 there are 4 variants
  - `__shfl_up_sync()`
    - Copy from a lane with lower ID relative to caller
  - `__shfl_down_sync()`
    - Copy from a lane with higher ID relative to caller
  - `__shfl_xor_sync()`
    - Copy from a lane based on bitwise XOR of own lane ID
  - `__shfl_sync()`
    - Direct copy from indexed lane ID
- Deprecated shuffle commands
  - `__shfl()`, `__shfl_up()`, `__shfl_down()`, `__shfl_xor()`

Warp shuffles

- `T __shfl_down_sync(unsigned mask, T var, unsigned int delta, int width=warpSize)`
  - `mask` controls which threads are involved — usually set to `-1` or `0xffffffff`, equivalent to all 1's
  - `var` is a local variable (built-in C types)
  - `delta` is the offset within the warp – if the appropriate thread does not exist (i.e. it's off the end of the warp) then the value is taken from the current thread
  - `width` optional parameter which alters the behavior of the intrinsic – results are undefined if `width` is not a power of 2, or is a number greater than `warpSize`

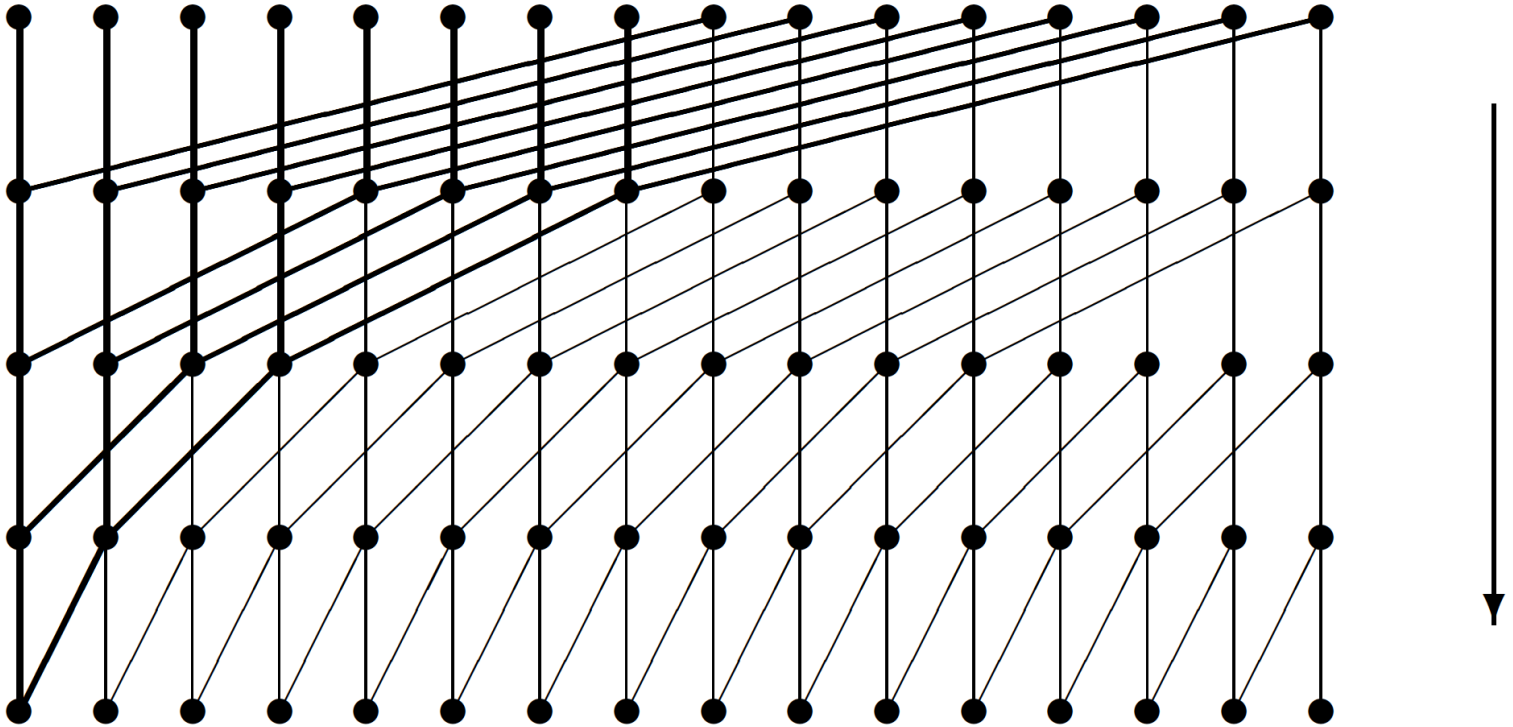
Warp shuffles

- `T __shfl_xor_sync(unsigned mask, T var, int laneMask, int width=warpSize)`
  - An XOR (exclusive or) operation is performed between `laneMask` and the calling thread's laneID to determine the lane from which to shuffle the value
  - A "butterfly" type of addressing (usefull for e.g., FFT)
- `T __shfl_sync(unsigned mask, T var, int srcLane, int width=warpSize)`
  - All lanes get a copy of value in `srcLane` (broadcast)

Warp sum reduction: Method 1

- Two ways to sum all elements in a warp

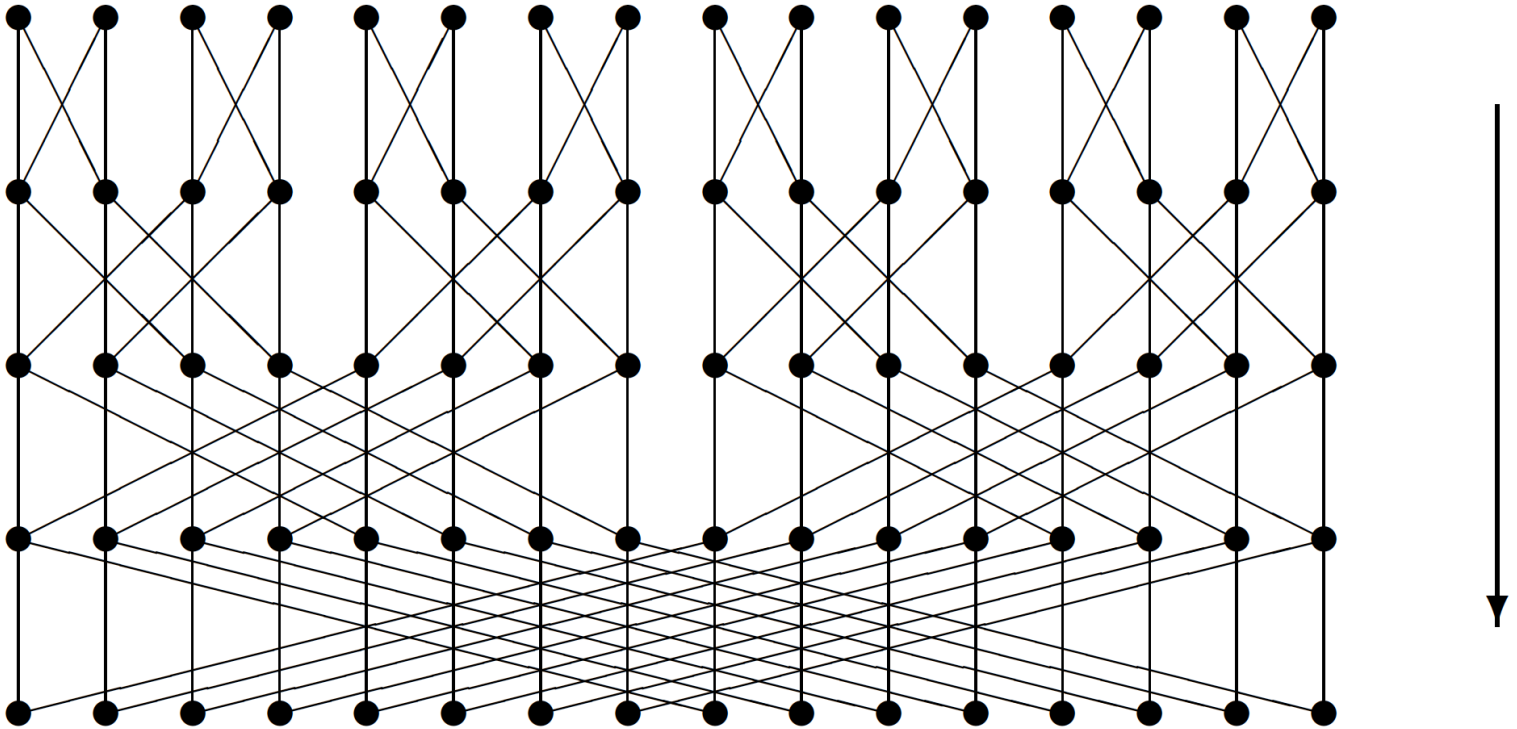
```
for (int i = 16; i > 0; i /= 2)
    value += __shfl_down_sync(-1, value, i);
```



Warp sum reduction: Method 2

- Two ways to sum all elements in a warp

```
for (int i = 1; i < 32; i *= 2)
    value += __shfl_xor_sync(-1, value, i);
```



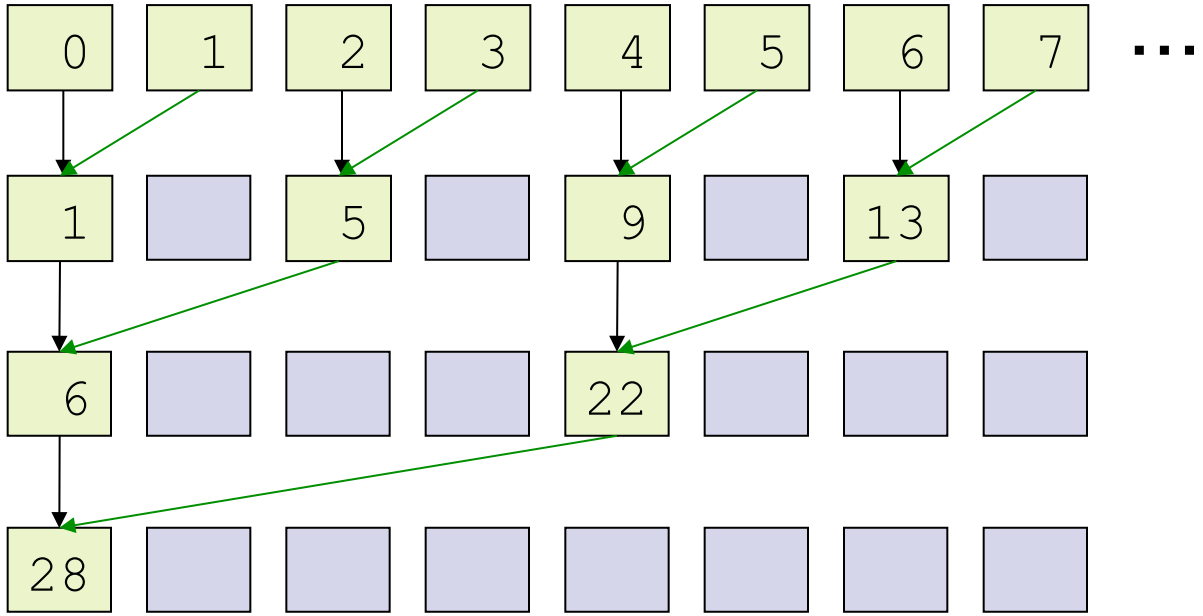
Reduction operation in CUDA

Reduction operation in CUDA

- Key requirement for a reduction operator $\circ$ are:
  - Commutative: $a \circ b = b \circ a$
  - Associative: $a \circ (b \circ c) = (a \circ b) \circ c$
  - Together, they mean that the elements can be re-arranged and combined in any order, e.g.,
  - Summation, minimum, maximum, etc.
- There is no built-in reduction operation in CUDA like in OpenMP and MPI − we have to write the code from scratch ourselves
- Here we will describe things for a sum reduction – the extension to other reductions is obvious

Sum reduction in CUDA

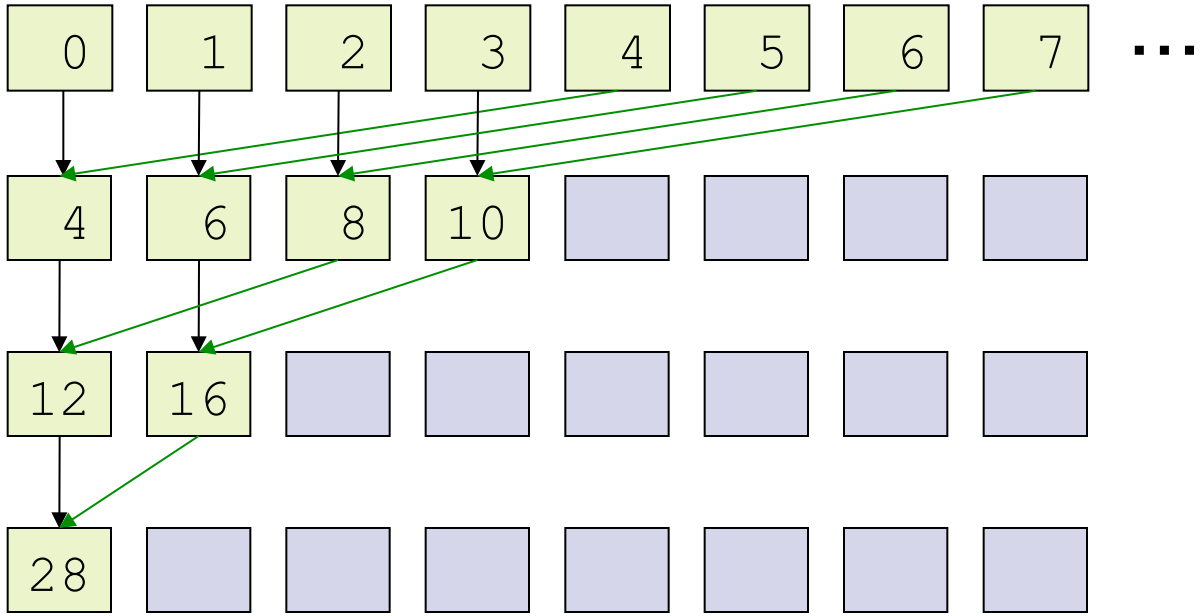
- log2(`n`) passes for `n` elements



- Not good − Why?

Sum reduction in CUDA

- log2(`n`) passes for `n` elements



- Better! How would you implement this in CUDA?

Sum reduction in CUDA

- We assume each thread starts with one value
  - Phase 1: Sum values within each warp to partial sum
  - Phase 2: Sum partial sums within each block
  - Phase 3: Sum from each block
- Data should be held in fastest memory possible
  - Warp partial sums computed in registers
  - Block partial sums computed in shared memory
  - Final result in global memory
- Baseline + tuning steps
  - Slow but simple: atomic add

Sum reduction in CUDA – baseline

```
__global__
void reduction_baseline(double *a, int n, double *res)
{
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < n) atomicAdd(res, a[idx]);
}
```

```
…
dim3 dimBlock(256);
dim3 dimGrid((n+dimBlock.x-1)/dimBlock.x);
reduction_baseline<<<dimGrid, dimBlock>>>(d_a, n, d_res);
…
```

Version	**baseline**
Time [ms]	523
Memory utilization	0.4 %

Warp sum reduction

- Device function for warp sum reduction

```
__inline__ __device__
double warpReduceSum(double value) {
    for (int i = 16; i > 0; i /= 2)
        value += __shfl_down_sync(-1, value, i);
    return value;
}
```

  - Only lane 0 will return the final summation result
  - NB: Parallel sums of floating point values may produce different results depending on summation order
    - No guarantee for specific order in the CUDA execution model

Sum reduction in CUDA – ver. 1

```
__global__
void reduction_warp(double *a, int n, double *res)
{
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    double value = idx < n ? a[idx] : 0;
    value = warpReduceSum(value);
    if (threadIdx.x % warpSize == 0)
        atomicAdd(res, value);
}
```

Version	baseline	v1 warp
Time [ms]	523	16.6
Memory utilization	0.4 %	13.7 %

Block sum reduction

```
__inline__ __device__
double blockReduceSum(double value) {
    __shared__ double smem[32]; // Max 32 warp sums

    if (threadIdx.x < warpSize)
        smem[threadIdx.x] = 0;
    __syncthreads();

    value = warpReduceSum(value);

    if (threadIdx.x % warpSize == 0)
        smem[threadIdx.x / warpSize] = value;
    __syncthreads();

    if (threadIdx.x < warpSize)
        value = smem[threadIdx.x];
    return warpReduceSum(value);
}
```

Sum reduction in CUDA – ver. 2

```
__global__
void reduction_block(double *a, int n, double *res)
{
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    double value = idx < n ? a[idx] : 0;
    value = blockReduceSum(value);
    if (threadIdx.x == 0)
        atomicAdd(res, value);
}
```

Version	baseline	v1 warp	v2 block
Time [ms]	523	16.6	2.96
Memory utilization	0.4 %	13.7 %	77.0 %

Sum reduction in CUDA – ver. 3

```
__global__
void reduction_presum(double *a, int n, double *res)
{
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    double value = 0;
    for (int i = idx; i < n; i += blockDim.x * gridDim.x)
        value += a[i];
    value = blockReduceSum(value);
    if (threadIdx.x == 0)
        atomicAdd(res, value);
}
```

Version	baseline	v1 warp	v2 block	v3 presum
Time [ms]	523	16.6	2.96	2.39
Memory utilization	0.4 %	13.7 %	77.0 %	95.0 %

Tips & tricks

Instruction optimizations

- Loop unrolling / branch predication
  - `for`, `do` and `while` has counter overhead
  - `#pragma unroll <n>` can be used to unroll loops

```
/* Before unrolling */
for (i = 0; i < N; ++i)
{
   c[i] = a[i] + b[i];
}
```

- The same idea as you learned in week 1
- Replace body of loop with multiple copies of loop content
- Fewer compare and branch instructions

```
/* After unrolling */
for (i = 0; i < N - (4 - 1); i += 4)
{
   c[i] = a[i] + b[i];
   c[i+1] = a[i+1] + b[i+1];
   c[i+2] = a[i+2] + b[i+2];
   c[i+3] = a[i+3] + b[i+3];
}
/* Remainder loop */
for (; i < N; ++i)
{
   c[i] = a[i] + b[i];
}
```

Be careful; `#pragma unroll` may result in more registers!

Instruction optimizations

- Low-level tuning
  - Awareness of how instructions are executed sometimes permits low-level optimizations at “hot-spots” in a kernel
  - Low priority - do it when everything else has been tuned
- Arithmetic
  - Single precision floats are at least twice as fast as doubles
    - Use float whenever higher precision is not needed
  - Integer division and modulo operations are costly
    - Replace `(i / n)` ➔ `(i >> log2(n))`, for $n = 2^P$
    - Replace `(i % n)` ➔ `(i & (n - 1))` , for $n = 2^P$
- Type conversions
  - Type conversions require extra instructions
    - In single precision make sure the use `1.0f` instead of `1.0`!

Instruction optimizations

- Loop counters
  - The compiler can optimize most aggressively on types that have unspecified overflow semantics
    - Use `int` rather than `unsigned int` for loop counters
- Math functions
  - Functions with underscores maps directly to the HW but with somewhat lower accuracy (24 bit)
    - Use `__sinf(x), __cosf(x),__expf(x)` etc.
    - Compiler option `–use_fast_math` sets this as default
  - Use special HW implemented functions
    - `rsqrtf(x), sincosf(x), exp2f(x), powf(x)`

Exercises

- Finish exercise 4 today!
- Next presentation “Assignment 3 intro” at 9.00 tomorrow morning (Wednesday)!
- Note on how to use the Nsight Profiler (Wed/Thu)
  - `nv-nsight-cu` / `nv-nsight-cu-cli`
  - https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html
  - https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html
  - https://developer.nvidia.com/nsight-compute-blogs

End of lecture