CUDA Performance Tuning Control Flow
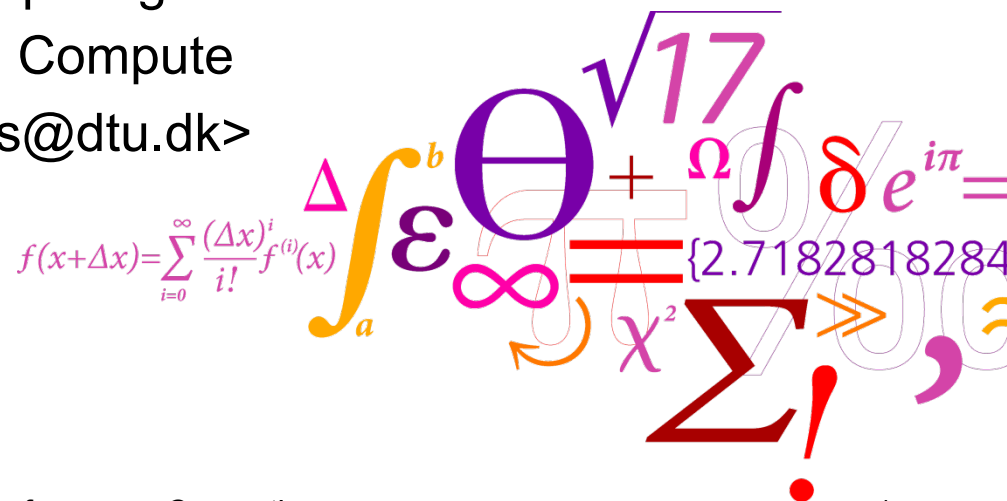
Hans Henrik Brandenborg Sørensen

DTU Computing Center

DTU Compute

<hhbs@dtu.dk>

Overview

- Optimizing control flow
  - ❑ Thread divergence

- Unrolling loops etc.

- A few tricks

Tuning of control flow

- **Branching and divergence**
  - ❑ `if`, `else`, `do`, `for`, and `switch` can significantly affect the instruction throughput by causing threads in a warp to take different execution paths
    - The different execution paths are serialized (run sequentially)

Tuning of control flow

- **Branching and divergence**
  - `if`, `else`, `do`, `for`, and `switch` can significantly affect the instruction throughput by causing threads in a warp to take different execution paths
    - The different execution paths are serialized (run sequentially)
  - Worst case (every warp diverges)

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

Tuning of control flow

- ## Branching and divergence

  - `if`, `else`, `do`, `for`, and `switch` can significantly affect the instruction throughput by causing threads in a warp to take different execution paths

    - The different execution paths are serialized (run sequentially)

  - Worst case (every warp diverges)

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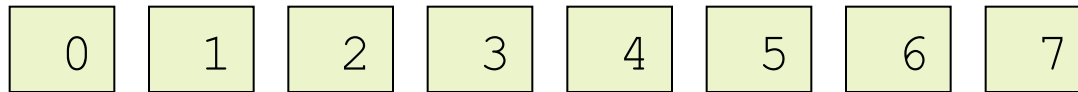    ```

  - Better (some warps diverges)

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

Tuning of control flow

- **Branching and divergence**
  - `if`, `else`, `do`, `for`, and `switch` can significantly affect the instruction throughput by causing threads in a warp to take different execution paths
    - The different execution paths are serialized (run sequentially)
  - Worst case (every warp diverges)

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

  - Better (some warps diverges)

    ```
    if (threadIdx.x < 30) { ... } else { ... }
    ```
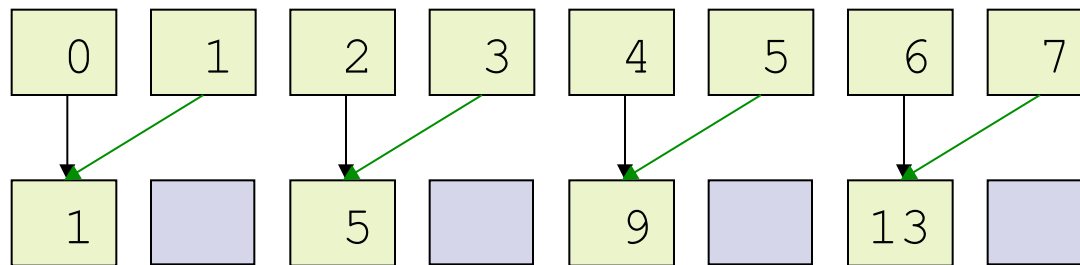
  - Good (no warps divergent)

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

Example: Parallel sum reduction

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Example: Parallel sum reduction

02614 – High Performance Computing

Example: Parallel sum reduction

02614 – High Performance Computing
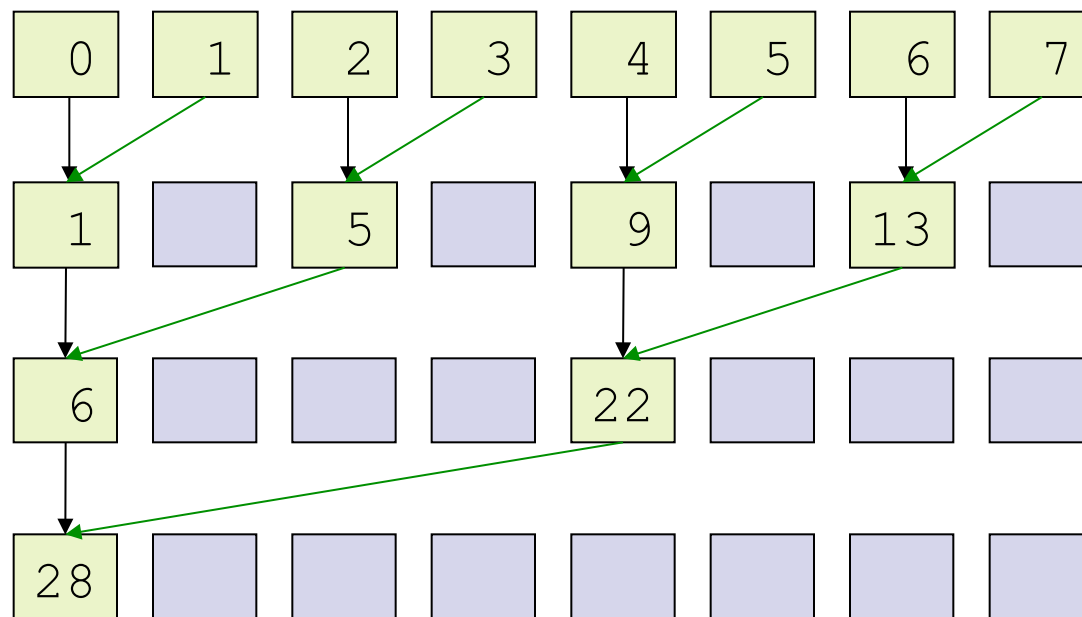
Example: Parallel sum reduction

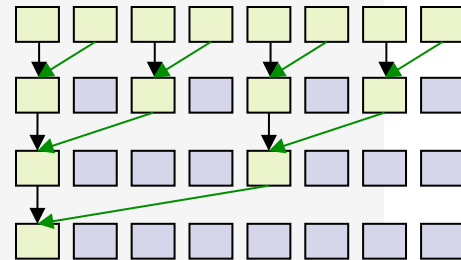■ log2(n) passes for n elements

■ How would you implement this in CUDA?

Example: Parallel sum reduction
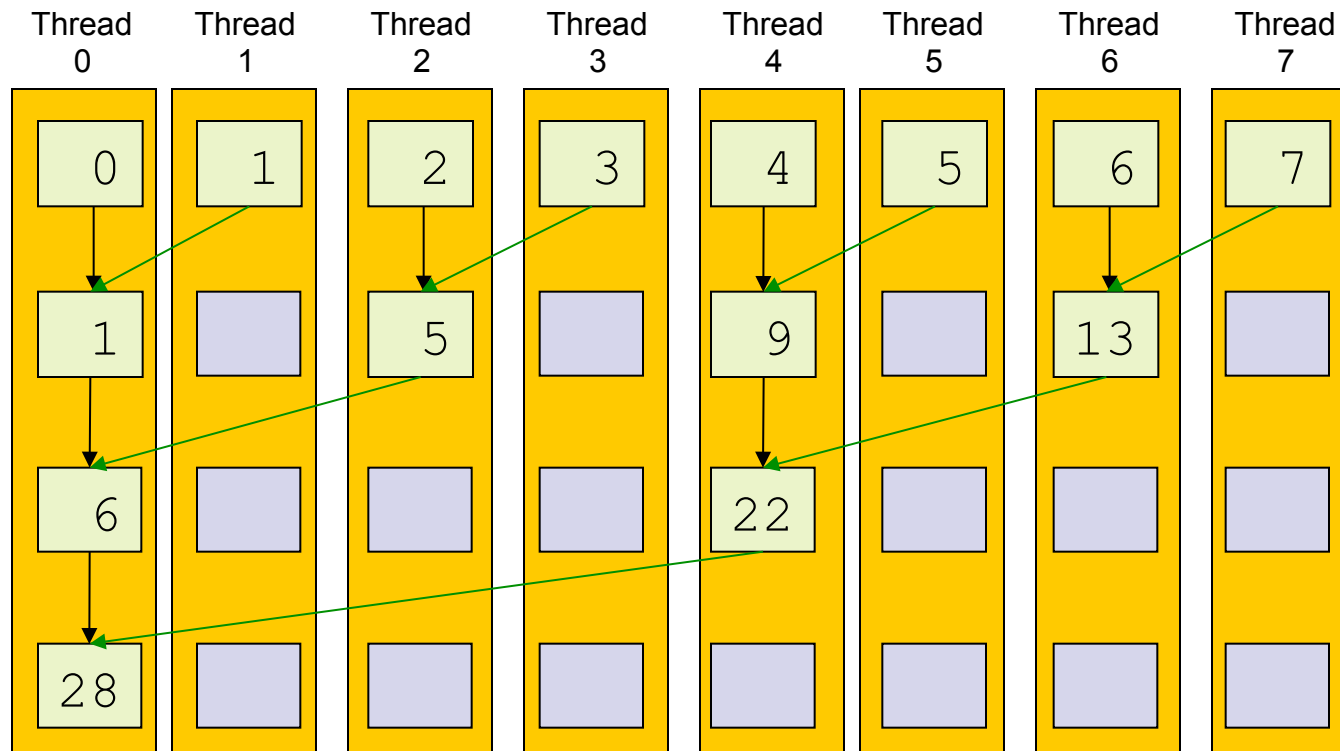
- Shared memory implementation:

```
extern __shared__ float partialSum[];
// ... load values from global into shared memory
int t = threadIdx.x;
for (int stride = 1;
     stride < blockDim.x;
    stride *= 2)
{

  __syncthreads();
  if (t % (2 * stride) == 0)
    partialSum[t] +=
      partialSum[t + stride];
}
```
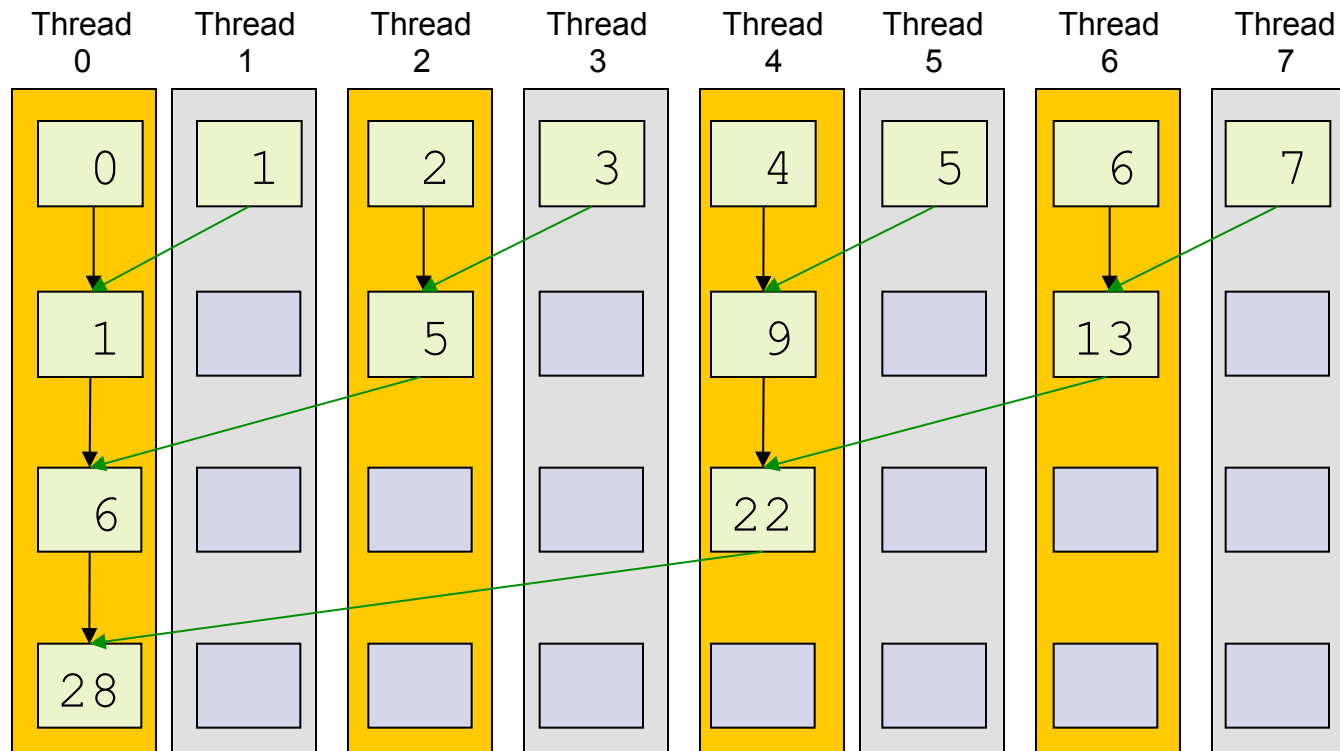
Stride:
  1, 2, 4, …

threads that do work:
  1/2, 1/4, 1/8 …
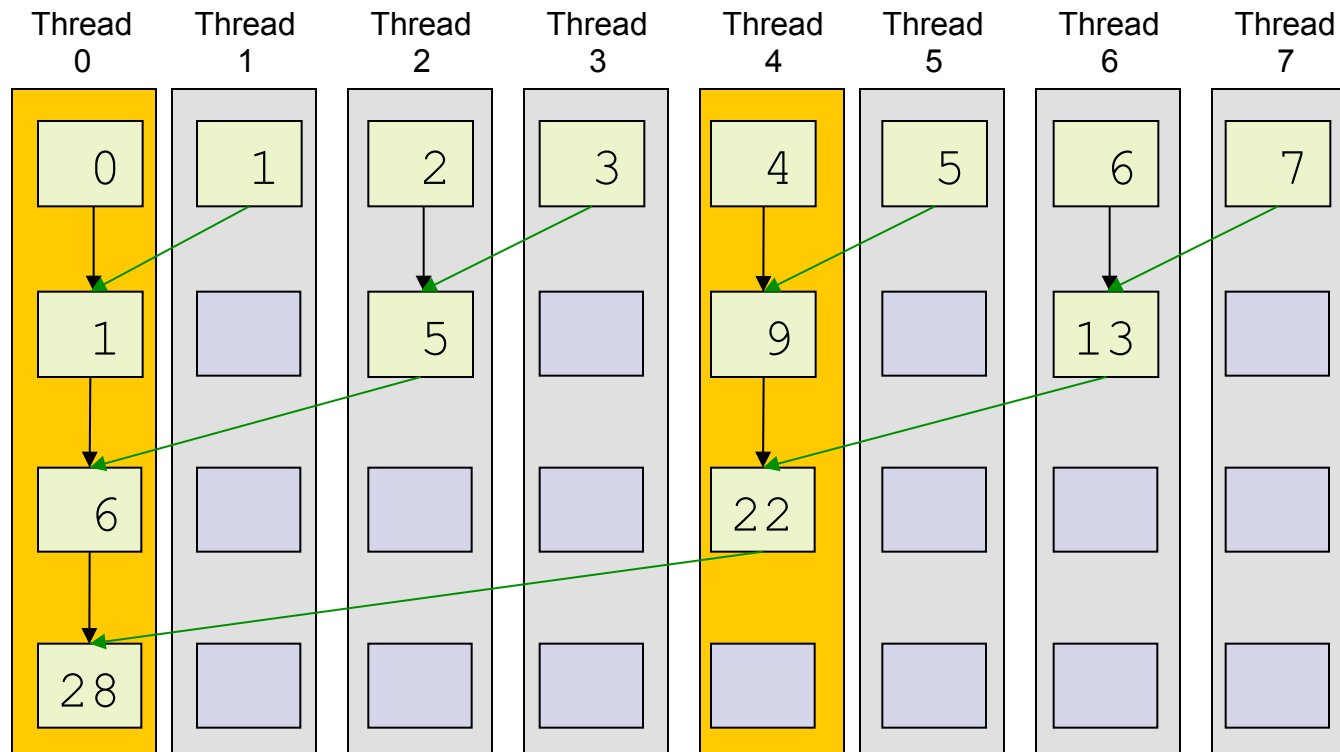
Example: Parallel sum reduction

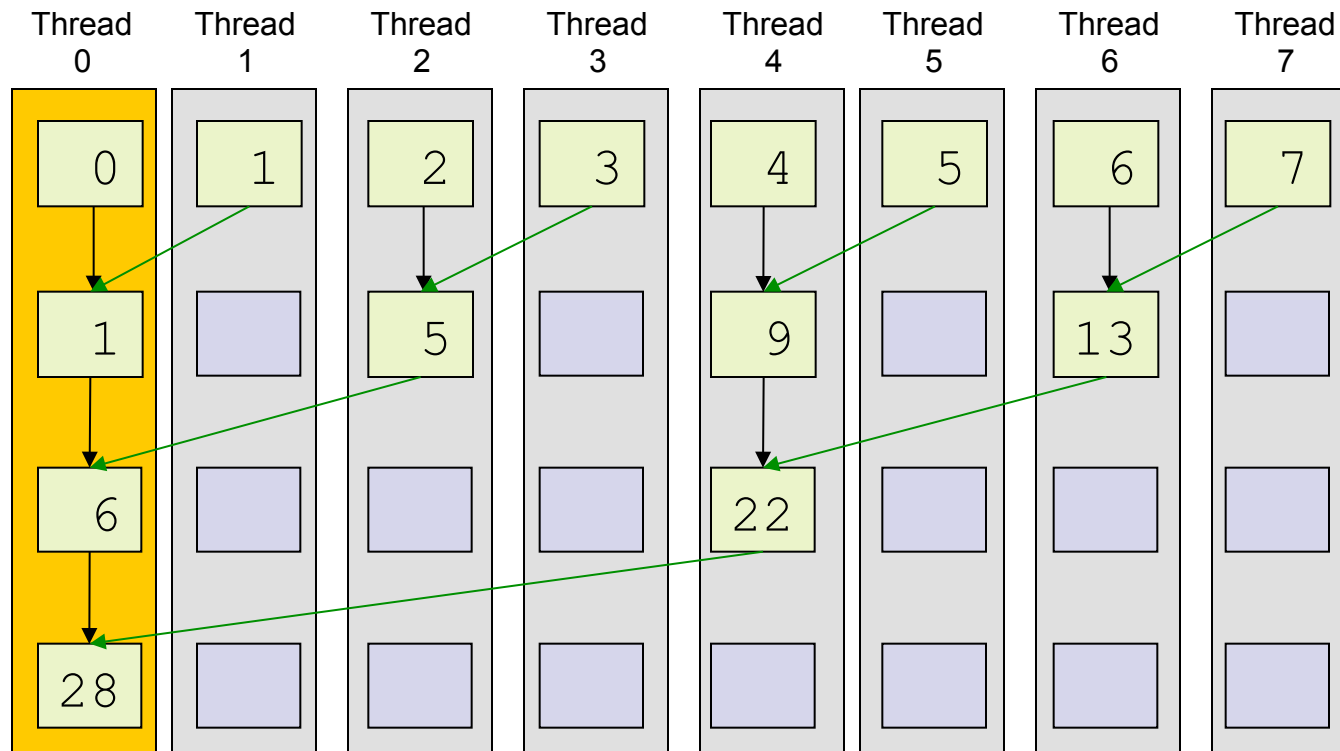# Example: Parallel sum reduction

■ 1st pass: threads 1, 3, 5, and 7 don't do anything

  ❑ Really only need `n/2` threads for `n` elements
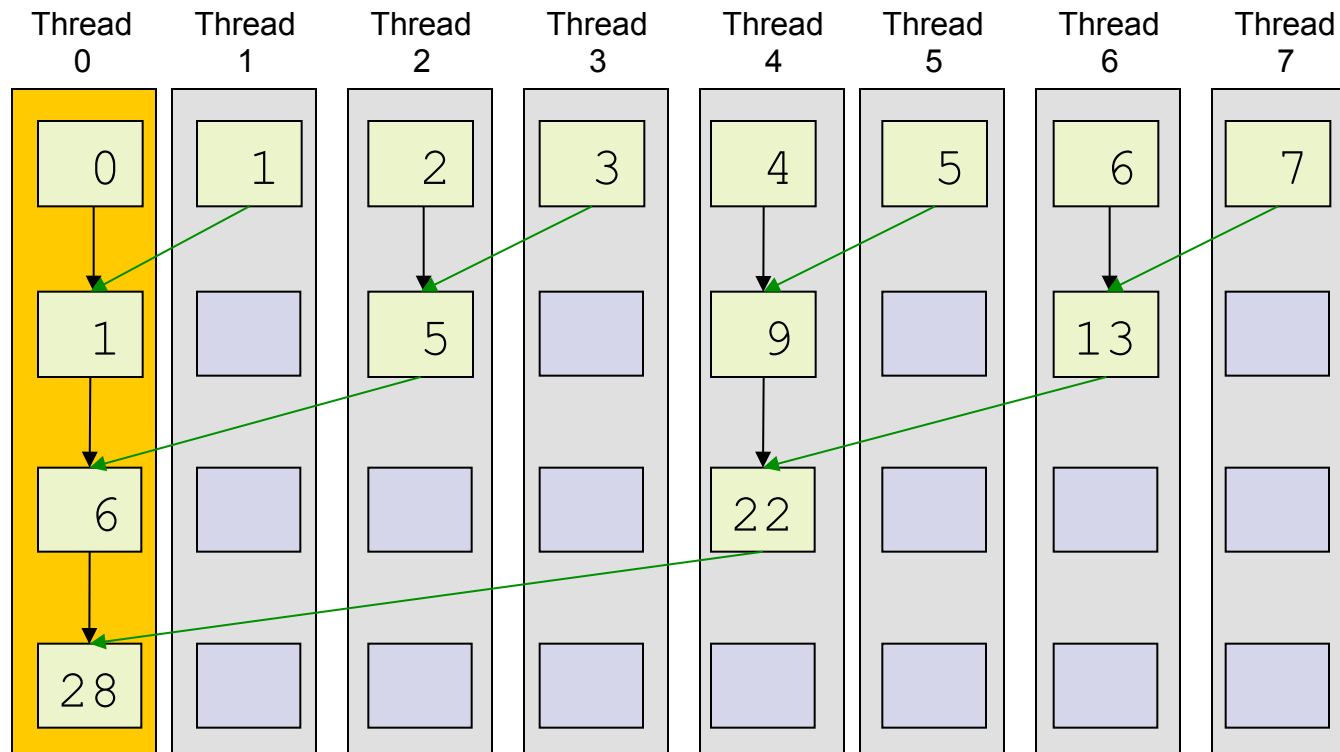
Example: Parallel sum reduction

- 2nd pass: threads 2 and 6 also don't do anything

Example: Parallel sum reduction

■ 3rd pass: thread 4 also doesn't do anything

Example: Parallel sum reduction

Thread 0	Thread 1	Thread 2	Thread 3	Thread 4	Thread 5	Thread 6	Thread 7
0	1	2	3	4	5	6	7
1		5		9		13	
6				22			
28							

- In general, number of required threads cuts in half after each pass

Example: Parallel sum reduction

- An alternative algorithm:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Example: Parallel sum reduction

- An alternative algorithm:

0	1	2	3	4	5	6	7

4	6	8	10				
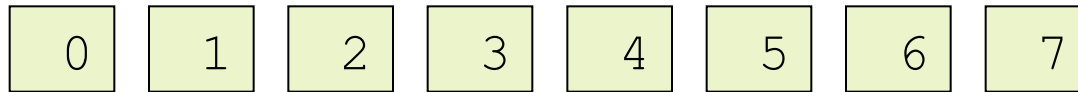
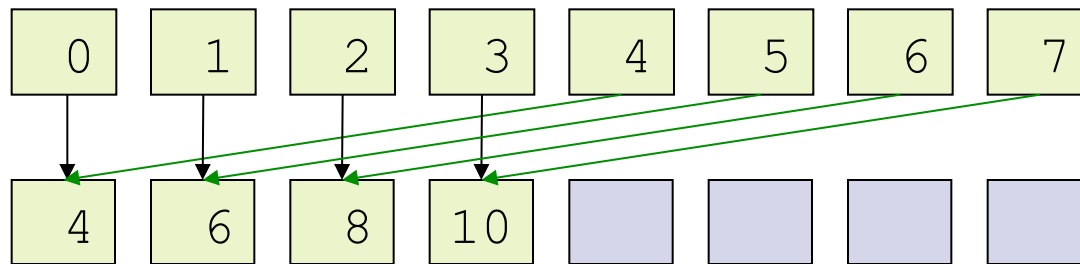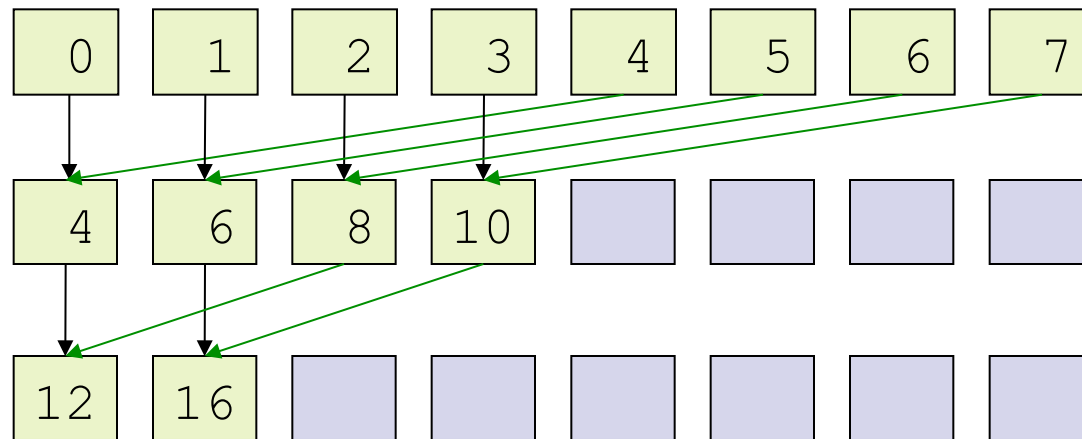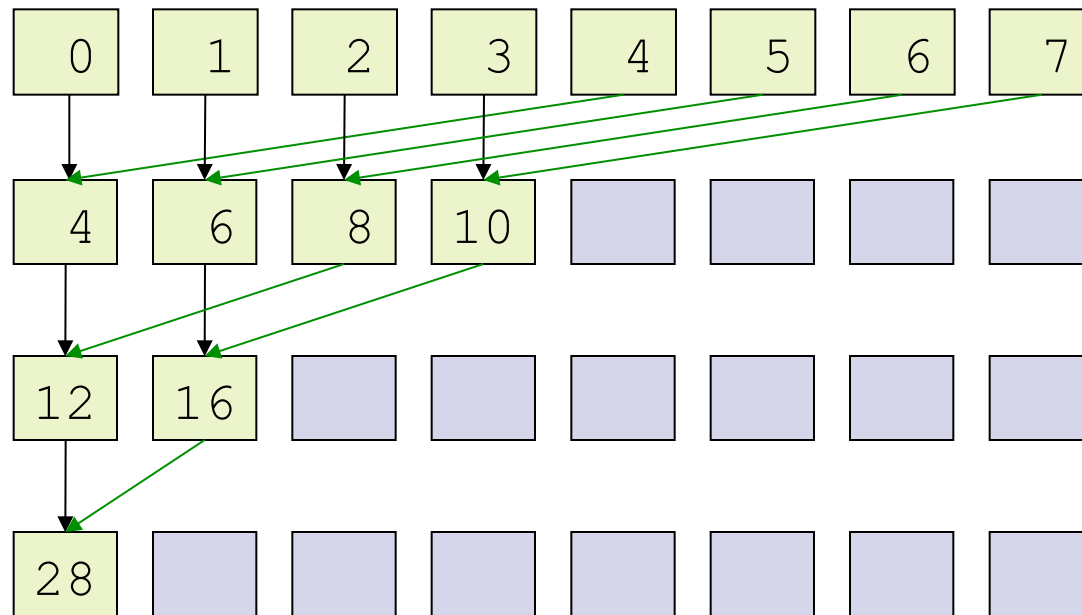# Example: Parallel sum reduction

- An alternative algorithm:

Example: Parallel sum reduction

- An alternative algorithm:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

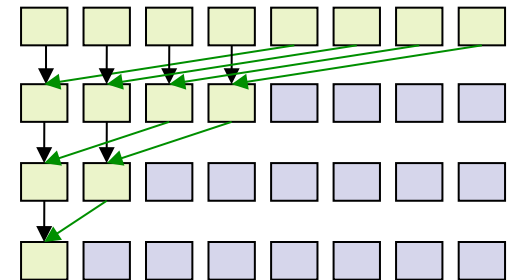| 4 | 6 | 8 | 10 | | | | |

| 12 | 16 | | | | | | |

| 28 | | | | | | | |

- Still log2(n) passes for n elements

Example: Parallel sum reduction

■ Alternative shared memory implementation:

```
extern __shared__ float partialSum[];
// ... load values from global into shared memory
int t = threadIdx.x;
for (int stride = blockDim.x / 2;
        stride > 0;
        stride /= 2)
{
    __syncthreads();
    if (t < stride)
        partialSum[t] +=
            partialSum[t + stride];
}
```
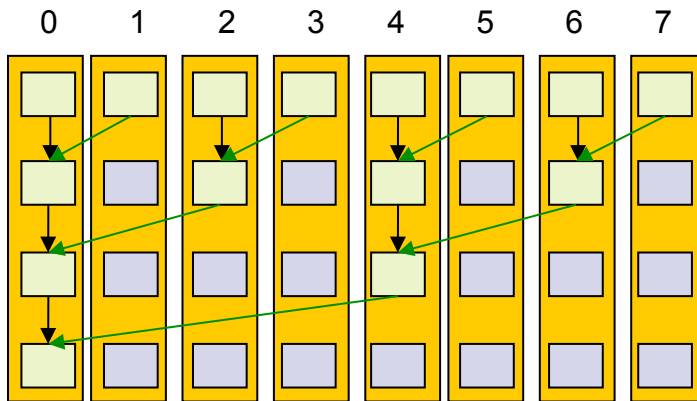
Stride:
    …, 4, 2, 1

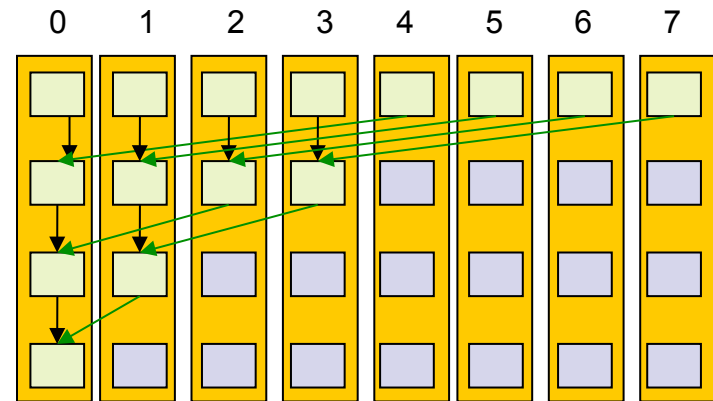# threads that do work:
    1/2, 1/4, 1/8 …

Example: Parallel sum reduction

■ What is the difference?


```
if (t % (2 * stride) == 0)
  partialSum[t] +=
    partialSum[t + stride];
```
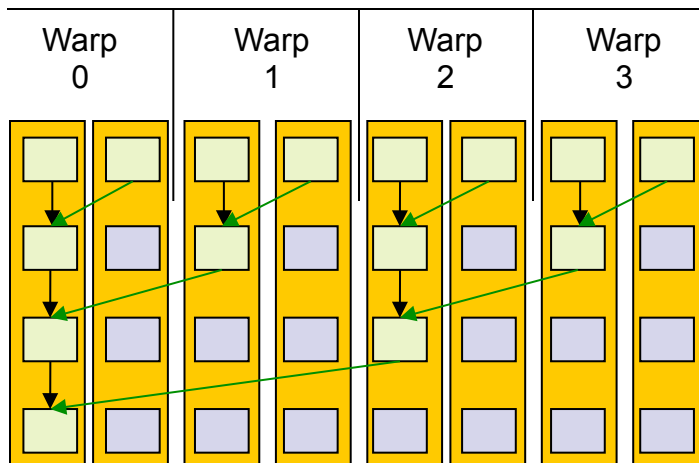
stride = 1, 2, 4, ...

```
if (t < stride)
  partialSum[t] +=
    partialSum[t + stride];
```
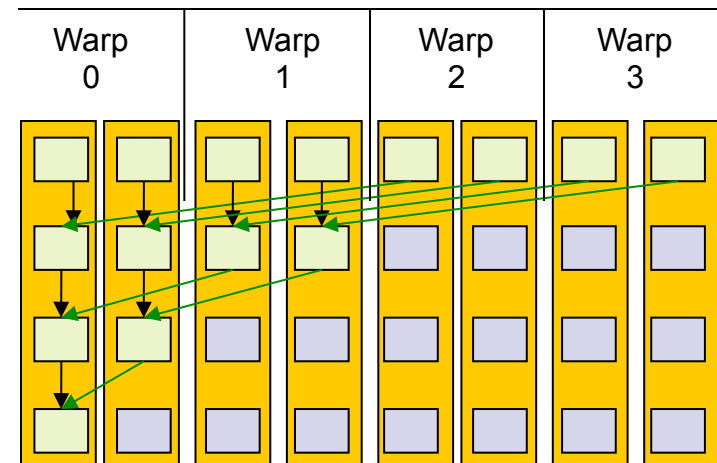
stride = 4, 2, 1, ...

Example: Parallel sum reduction

- Pretend `warpSize == 2`

stride = 1, 2, 4, ...   stride = 4, 2, 1, ...

Example: Parallel sum reduction

- 1st Pass

| Warp 0 | Warp 1 | Warp 2 | Warp 3 |

4 divergent branches

stride = 1, 2, 4, ...

| Warp 0 | Warp 1 | Warp 2 | Warp 3 |

0 divergent branches

stride = 4, 2, 1, ...

Example: Parallel sum reduction

- 2nd Pass

2 divergent branches

stride = 1, 2, 4, ...

0 divergent branches

stride = 4, 2, 1, ...

Example: Parallel sum reduction

- 3rd Pass

1 divergent branch

1 divergent branch

`stride` = 1, 2, 4, ...

`stride` = 4, 2, 1, ...

Still diverge when number of elements left is <= `warpSize`
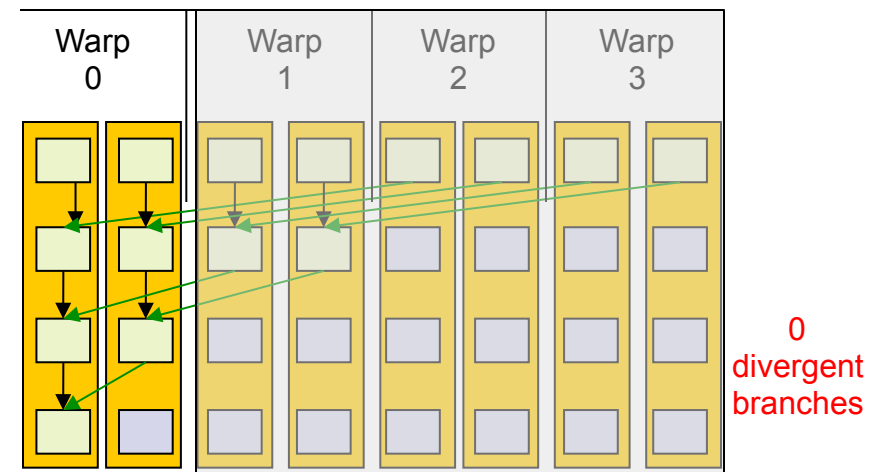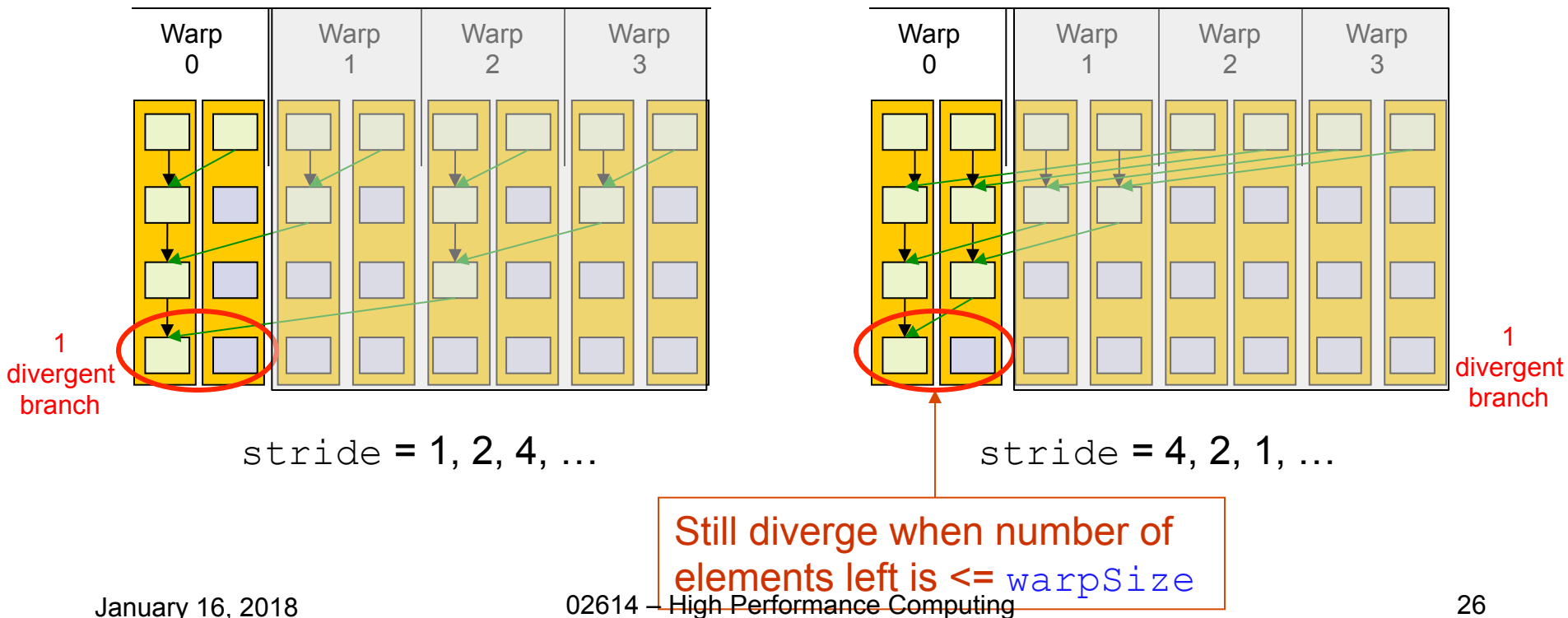
Instruction optimizations

- **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)`

End of lecture