UNIVERSITÀ
DEGLI STUDI
FIRENZE

Parallel Programming

Prof. Marco Bertini

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Data parallelism: GPU computing

UNIVERSITÀ
DEGLI STUDI
FIRENZE

CUDA: atomic operations, privatization, algorithms

Atomic operations

- The basics atomic operation in hardware is something like a read-modify-write operation performed by a single hardware instruction on a memory location address
  - Read the old value, calculate a new value, and write the new value to the location
- The hardware ensures that no other threads can perform another read-modify-write operation on the same location until the current atomic operation is complete
  - Any other threads that attempt to perform an atomic operation on the same location will typically be held in a queue
  - All threads perform their atomic operations serially on the same location

Atomic Operations in CUDA

- Performed by calling functions that are translated into single instructions (a.k.a. intrinsic functions or intrinsics)
- Operation on one 32-bit or 64-bit word residing in global or shared memory.
- Atomic functions can only be used in device functions
- Atomic add, sub, inc, dec, min, max, exch (exchange), CAS (compare and swap), and, or, xor

Some examples

- Atomic Add
- `int atomicAdd(int* address, int val);`
  - reads the 32-bit word old from the location pointed to by address in global or shared memory, computes (old + val), and stores the result back to memory at the same address. The function returns old.
- Unsigned 32-bit integer atomic add
- `unsigned int atomicAdd(unsigned int* address, unsigned int val);`
- Unsigned 64-bit integer atomic add
- `unsigned long long int atomicAdd(unsigned long long int* address, unsigned long long int val);`
- Single-precision floating-point atomic add (capability > 2.0)
- `float atomicAdd(float* address, float val);`
- Double precision floating-point atomic add (capability > 6.0)
- `double atomicAdd(double* address, double val);`

atomicCAS

- `int atomicCAS(int* address, int compare, int val);`

- ```
  unsigned int atomicCAS(unsigned int* address,
                         unsigned int compare,
                         unsigned int val);
  ```

- ```
  unsigned long long int atomicCAS(unsigned long long
      int* address,
  unsigned long long int
      compare,
  unsigned long long int
      val);
  ```

- reads the 32-bit or 64-bit word `old` located at the address `address` in global or shared memory, computes (`old == compare ? val : old`) , and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old (Compare And Swap).

atomicCAS

- `int atomicCAS(int* address, int compare, int val);`

- ```
  unsigned int atomicCAS(unsigned int* address,
                         unsigned int compare,
                         unsigned int val);
  ```

- ```
  unsigned long long int atomicCAS(unsigned long long
                                   int* address,
                                   unsigned long long int
                                   compare,
                                   unsigned long long int
                                   val);
  ```

- reads the 32-bit or 64-bit word `old` located at the address `address` in global or shared memory, computes (`old == compare ? val : old`) , and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old (Compare And Swap).

More precisely: `*address=(*address==compare) ? val : *address;`

atomicCAS

- Note that any atomic operation can be implemented based on `atomicCAS()` (Compare And Swap). For example, `atomicAdd()` for double-precision floating-point numbers is not available on devices with compute capability lower than 6.0 but it can be implemented as follows:

```
#if __CUDA_ARCH__ < 600
__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;
    unsigned long long int assumed;
    do {
        assumed = old;
        old = atomicCAS(address_as_ull, assumed,
                __double_as_longlong(val + __longlong_as_double(assumed)));
    // Note: uses integer comparison to avoid hang in case
    //       of NaN (since NaN != NaN)
    } while (assumed != old);
    return __longlong_as_double(old);
}
#endif
```

atomicCAS

- Note that any atomic operation can be implemented based on `atomicCAS()` (Compare And Swap). For example, `atomicAdd()` for double-precision floating-point numbers is not available on devices with compute capability lower than 6.0 but it can be implemented as follows:

```
#if __CUDA_ARCH__ < 600
__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;
    unsigned long long int assumed;
    do {
        assumed = old;
        old = atomicCAS(address_as_ull, assumed,
                __double_as_longlong(val + __longlong_as_double(assumed)));
    // Note: uses integer comparison to avoid hang in case
    //       of NaN (since NaN != NaN)
    } while (assumed != old);
    return __longlong_as_double(old);
}
#endif
```

Reinterpret the bits in the 64-bit signed integer value as a double-precision floating point value.

Critical section

- Using atomic instructions, in particular CAS, it is possible to implement a critical section. We need to use also `atomicExch()` that exchanges a value:

```
__device__ int lock = 0;

__global__ void kernel() {
  // ...
  if (threadIdx.x==0) {
      // set lock
      do {} while(atomicCAS(&lock, 0, 1); // spin...
      // critical code section
      atomicExch(&lock, 0); // release lock
    }
  }
}
```

We use thread 0 of each block for mutual exclusion

Memory fence

- atomicCAS can not pick stale values of the lock, since global atomics bypass L1 and are resolved in L2 cache, which is a device-wide resource

- But no one assures us that a programmer does not read the mutex variable (after all it is just a global variable…)

- To avoid reading stale values in other threads we need to force to read actual values, ie.. using a memory fence. GPUs implement a weak memory ordering…

- Use `__threadfence()`

__threadfence()

- `__threadfence_block();`
- wait until all global and shared memory writes are visible to:
  - all threads in block
  - all threads, for global data

- `__threadfence();`
- wait until all global and shared memory writes are visible to:
  - all threads in block
  - all threads, for global data

Lock structure

```
struct Lock {
    int *mutex;
    Lock( void ) {
        cudaMalloc( (void**)&mutex, sizeof(int) );
        cudaMemset( mutex, 0, sizeof(int) );
    }

    ~Lock( void ) {
        cudaFree( mutex );
    }

    __device__ void lock( void ) {
        while( atomicCAS( mutex, 0, 1 ) != 0 ); // spin lock: cycle until it sees  0
         __threadfence();
    }

    __device__ void unlock( void ) {
         __threadfence();
         atomicExch( mutex, 0 );
    }

};
```

Lock and warps

- Threads within a warp negotiating for a lock can be quite challenging due to the GPU warp-based execution:

```
__device__ int lock;

__global__ void Deadlock() {
    while (atomicCAS(&lock, 0, 1) != 0){}
    // critical section
    atomicExch(&lock, 0);
}
```

Lock and warps

- Threads in a warp execute in lockstep. The threads in a warp entering the while loop must all acquire the lock before any can proceed beyond that while loop. Unfortunately this is impossible, and there is deadlock.

execution:

```
__device__ int lock;

__global__ void Deadlock() {
    while (atomicCAS(&lock, 0, 1) != 0){}
    // critical section
    atomicExch(&lock, 0);
}
```

Lock and warps

- A way to avoid the previous problem;

```
__device__ int lock = 0;

__global__ void kernel() {
  bool blocked = true;
  while(blocked) {
    if (0 == atomicCAS(&lock, 0, 1)) {
      doCriticalJob();
      atomicExch(&lock, 0);
      blocked = false;
    }
  }
}
```

Each thread that acquires the lock has a chance to release it.
All the threads waiting to lock are inside the same while loop, one of them will get the lock and then exit the loop. Once all threads have acquired/ released the lock, the code continues after the loop

Note: Managing mutexes or critical sections, especially when the negotiation is amongst threads in the same warp is notoriously difficult and fragile. *The general advice is to avoid it.*
If you must use mutexes or critical sections, have a single thread in the threadblock negotiate for any thread that needs it, then control behavior within the threadblock using intra-threadblock synchronization mechanisms, such as `__syncthreads()`.

```
__device__ int lock = 0;

__global__ void kernel() {
  bool blocked = true;
  while(blocked) {
    if (0 == atomicCAS(&lock, 0, 1)) {
        doCriticalJob();
        atomicExch(&lock, 0);
        blocked = false;
    }
  }
}
```

Each thread that acquires the lock has a chance to release it.
All the threads waiting to lock are inside the same while loop, one of them will get the lock and then exit the loop. Once all threads have acquired/ released the lock, the code continues after the loop

Atomic operations and caches

- Atomic operations serialize simultaneous updates to a location, thus to improve performance the serialization should be as fast as possible

  - changing locations in global memory is slow: e.g. with an access latency of 200 cycles and 1 Ghz clock the throughput of atomics is 1/400 (atomics/clock) * 1 G (clocks/secs) = 2.5M atomics/secs (vs. the Gflops of modern GPUs)

- Cache memories are the primary tool for reducing memory access latency (e.g. 10s of cycle vs. 100s of cycles).

- Recent GPUs allow atomic operation to be performed in the last level cache, which is shared among all SMs.

Privatization

- The latency for accessing memory can be dramatically reduced by placing data in the shared memory.
  Shared memory is private to each SM and has very short access latency (a few cycles); this directly translates into increase throughput of atomic operations.

- The problem is that due to the private nature of shared memory, the updates by threads in one thread block is no longer visible to threads in other blocks.

Privatization

- The idea of **privatization** is to replicate highly contended data structures into private copies so that each thread (or each subset of threads) can access a private copy.
  The benefit is that the private copies can be accessed with much less contention and often at much lower latency.

- These private copies can dramatically increase the throughput for updating the data structures. The down side is that the private copies need to be merged into the original data structure after the computation completes. One must carefully balance between the level of contention and the merging cost.

Example: histogram computation

```
__global__ void histogram_kernel(const char *input, unsigned int *bins,
                                 unsigned int num_elements,
                                 unsigned int num_bins) {
  unsigned int tid = blockIdx.x * blockDim.x + threadIdx.x;
  // Privatized bins
  extern __shared__ unsigned int bins_s[];
  for (unsigned int binIdx = threadIdx.x; binIdx < num_bins;
       binIdx += blockDim.x) {
    bins_s[binIdx] = 0;
  }
  __syncthreads();


  // Histogram
  for (unsigned int i = tid; i < num_elements; i += blockDim.x * gridDim.x) {
    atomicAdd(&(bins_s[(unsigned int)input[i]]), 1);
  }
  __syncthreads();

  // Commit to global memory
  for (unsigned int binIdx = threadIdx.x; binIdx < num_bins;
         binIdx += blockDim.x) {
    atomicAdd(&(bins[binIdx]), bins_s[binIdx]);
  }
}
```

Example: histogram computation

```
__global__ void histogram_kernel(const char *input, unsigned int *bins,
                                 unsigned int num_elements,
                                 unsigned int num_bins) {
  unsigned int tid = blockIdx.x * blockDim.x + threadIdx.x;
  // Privatized bins
  extern __shared__ unsigned int bins_s[];
  for (unsigned int binIdx = threadIdx.x; binIdx < num_bins;
```

Dynamically allocated shared memory. To allocate it dynamically invoke the kernel with:

```
dim3 blockDim(256), gridDim(30);
histogram_kernel<<<gridDim, blockDim,
                   num_bins * sizeof(unsigned int)>>>
                (input, bins, num_elements, num_bins);
```

```
  }
  __syncthreads();

  // Commit to global memory
  for (unsigned int binIdx = threadIdx.x; binIdx < num_bins;
         binIdx += blockDim.x) {
    atomicAdd(&(bins[binIdx]), bins_s[binIdx]);
  }
}
```

Improving memory access

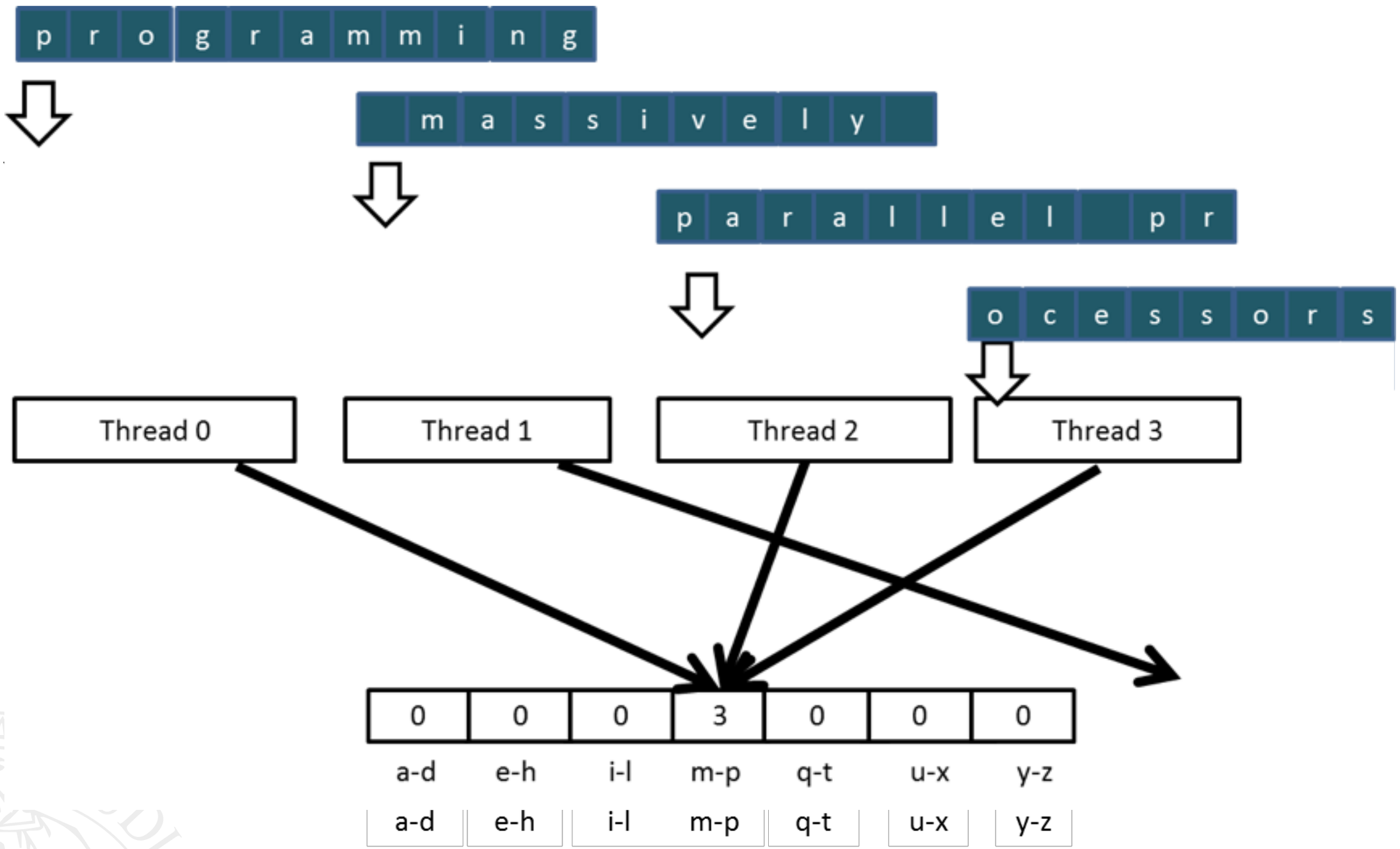
- A simple parallel histogram algorithm partitions the input into sections

- Each section is given to a thread, that iterates through it

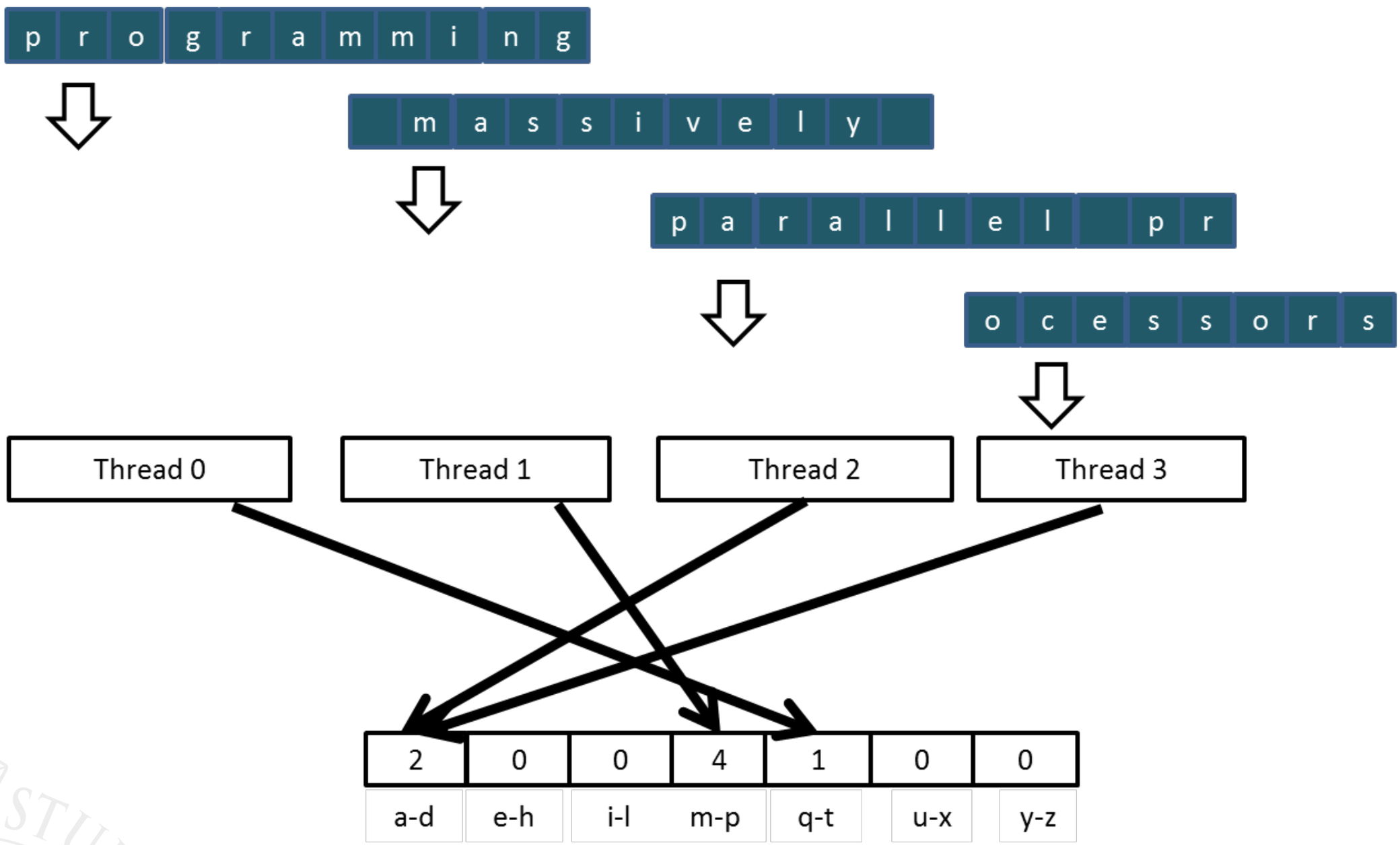
- This makes sense in CPU code, where we have few threads, each of which can efficiently use the cache lines when accessing memory

- This access is less convenient in GPUs

Sectioned Partitioning (Iteration #1)

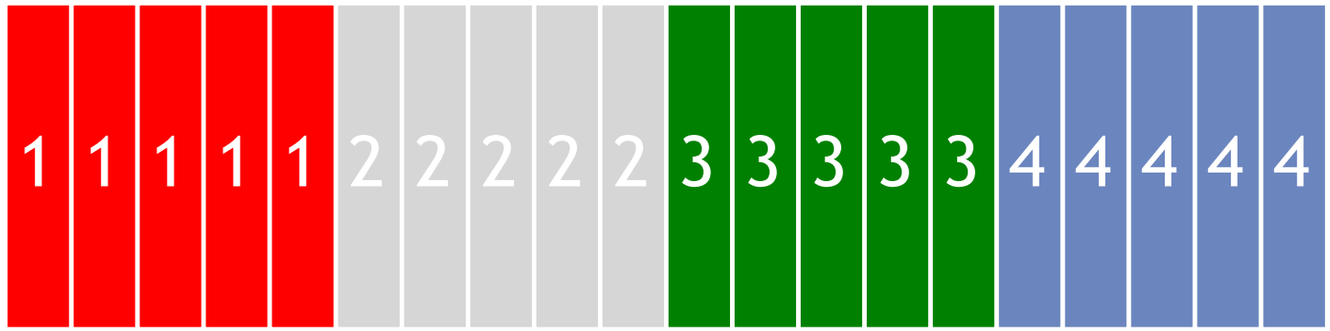


Sectioned Partitioning (Iteration #2)

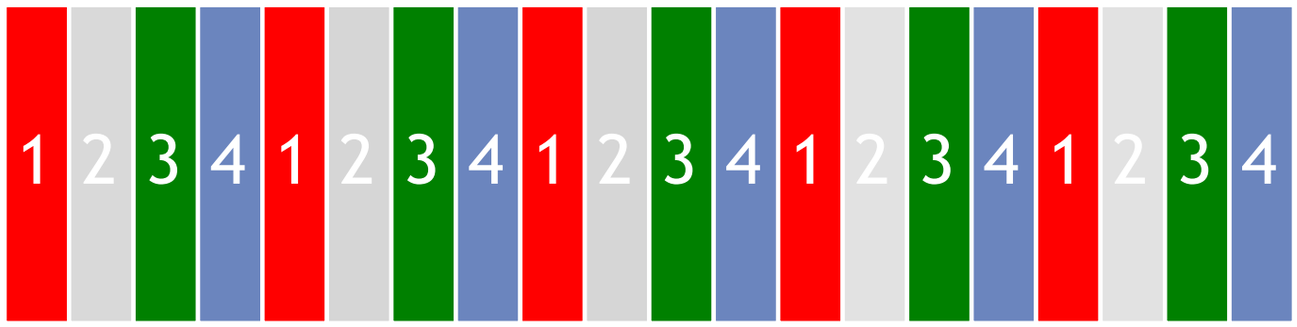


Input Partitioning Affects Memory Access Efficiency

- Sectioned partitioning results in poor memory access efficiency
  - Adjacent threads do not access adjacent memory locations
  - Accesses are not coalesced
  - DRAM bandwidth is poorly utilized

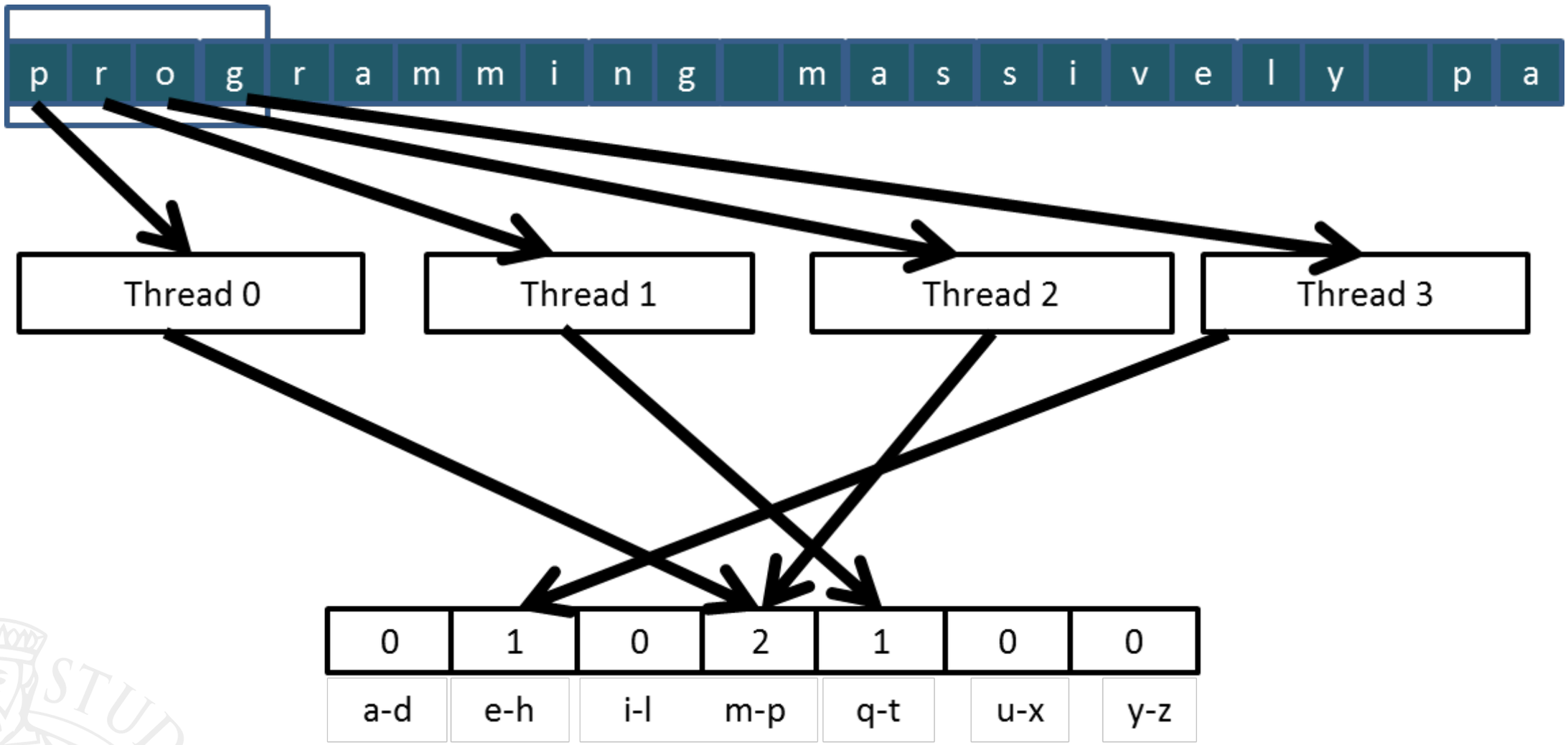


- Change to interleaved partitioning
  - All threads process a contiguous section of elements
  - They all move to the next section and repeat
  - The memory accesses are coalesced

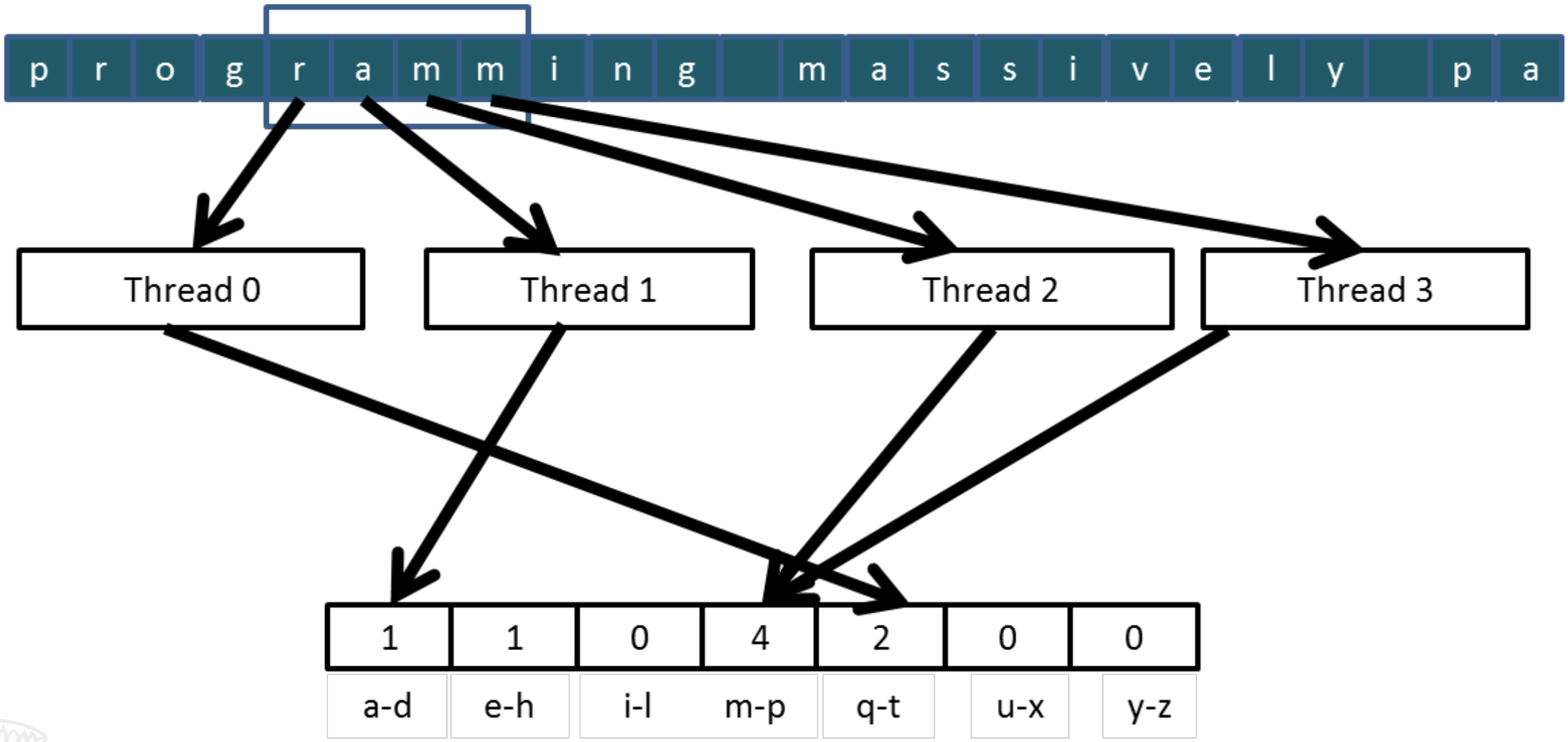


Interleaved Partitioning of Input

- For coalescing and better memory access performance



Interleaved Partitioning (Iteration 2)



A stride algorithm

```
__global__ void histo_kernel(unsigned char *buffer,
                             long size, unsigned int *histo)
{

    int i = threadIdx.x + blockIdx.x * blockDim.x;

    // stride is total number of threads
     int stride = blockDim.x * gridDim.x;

    // All threads handle blockDim.x * gridDim.x
    // consecutive elements
    while (i < size) {
        atomicAdd( &(histo[buffer[i]]), 1);
        i += stride;
    }

}
```

A stride algorithm

Calculates a `stride` value, which is the total number threads launched during kernel invocation (`blockDim.x*gridDim.x`). In the first iteration of the `while` loop, each thread index the input buffer using its global thread index: Thread 0 accesses element 0, Thread 1 accesses element 1, etc. Thus, all threads jointly process the first blockDim.x*gridDim.x elements of the input buffer.

```
int i = threadIdx.x + blockIdx.x * blockDim.x;

 // stride is total number of threads
 int stride = blockDim.x * gridDim.x;

// All threads handle blockDim.x * gridDim.x
// consecutive elements
while (i < size) {
    atomicAdd( &(histo[buffer[i]]), 1);
    i += stride;
}

}
```

UNIVERSITÀ DEGLI STUDI FIRENZE

A stride algorithm

Calculates a `stride` value, which is the total number threads launched during kernel invocation (`blockDim.x*gridDim.x`). In the first iteration of the `while` loop, each thread index the input buffer using its global thread index: Thread 0 accesses element 0, Thread 1 accesses element 1, etc. Thus, all threads jointly process the first blockDim.x*gridDim.x elements of the input buffer.

```
int i = threadIdx.x + blockIdx.x * blockDim.x;

 // stride is total number of threads
 int stride = blockDim.x * gridDim.x;
```

```
// All threads handle blockDim.x * gridDim.x
// consecutive elements
while (i < size) {
    atomicAdd( &(histo[buffer[i]]), 1);
    i += stride;
}
```

The while loop controls the iterations for each thread. When the index of a thread exceeds the valid range of the input buffer (`i` is greater than or equal to `size`), the thread has completed processing its partition and will exit the loop.

UNIVERSITÀ
DEGLI STUDI
FIRENZE

CUDA: parallel patterns - convolution

Convolution (stencil)

- An array operation where each output data element is a weighted sum of a collection of neighboring input elements

- The weights used in the weighted sum calculation are defined by an input mask array, commonly referred to as the convolution kernel

- We will refer to these mask arrays as **convolution masks** to avoid confusion.

- The value pattern of the mask array elements defines the type of filtering done

Convolution (stencil)

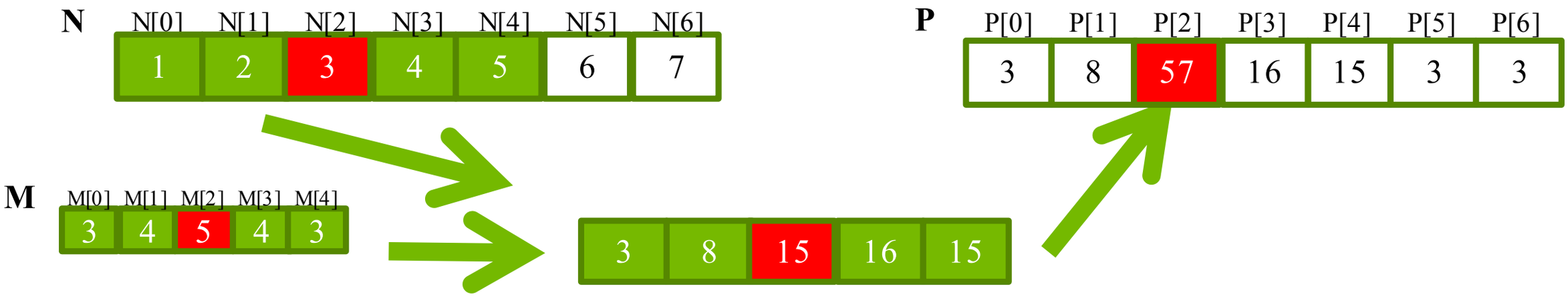
- An array operation where each output data element is a weighted sum of a collection of neighboring input elements

- The weights used in the weighted sum calculation are defined by an input mask array, commonly referred to as the convolution kernel

- We will refer to these mask arrays as **convolution masks** to avoid confusion.

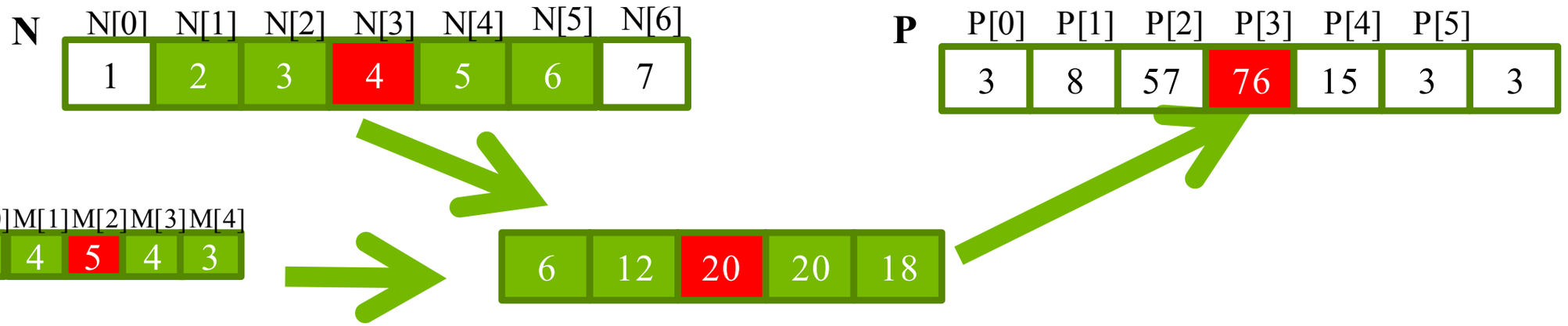
- Often performed as a filter that transforms signal or pixel values into more desirable values.

1D Convolution Example

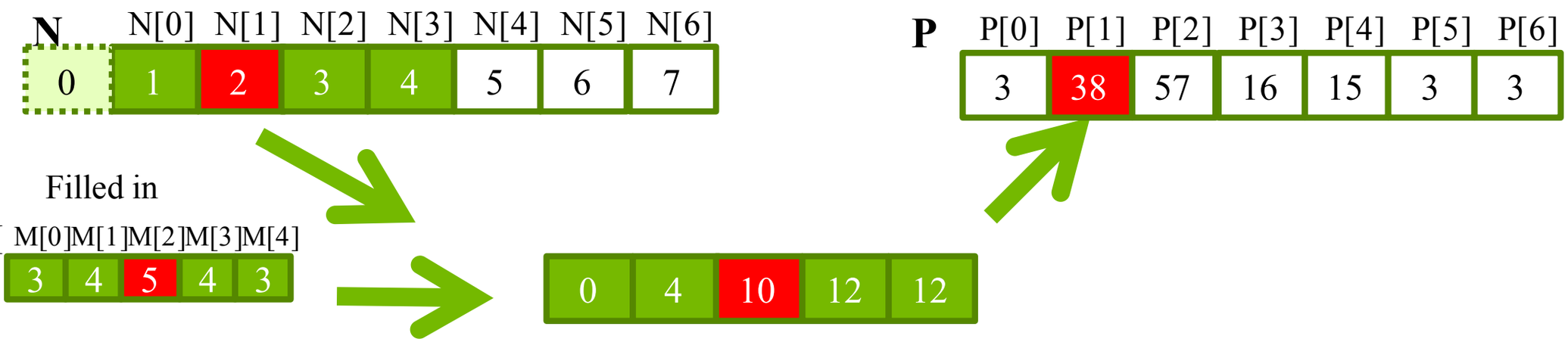


- Commonly used for audio processing
- Mask size is usually an odd number of elements for symmetry (5 in this example)
- The figure shows calculation of P[2]
- `P[2] = N[0]*M[0] + N[1]*M[1] + N[2]*M[2] + N[3]*M[3] + N[4]*M[4]`

Example: calculation of P[3]



Convolution Boundary Condition



- Calculation of output elements near the boundaries (beginning and end) of the array need to deal with “ghost” elements
  - Different policies (0, replicates of boundary values, use of symmetrical values, etc.)

A 1D Convolution Kernel with Boundary Condition Handling

```
__global__ void convolution_1D_basic_kernel(float *N, float *M,
        float *P, int Mask_Width, int Width) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;

    float Pvalue = 0;
    int N_start_point = i - (Mask_Width/2);

    for (int j = 0; j < Mask_Width; j++) {
        if (N_start_point + j >= 0 &&
            N_start_point + j < Width) {
            Pvalue += N[N_start_point + j]*M[j];
        }
    }

    P[i] = Pvalue;
}
```

- This kernel forces all elements outside the valid input range to 0

A 1D Convolution Kernel with Boundary Condition Handling

```
__global__ void convolution_1D_basic_kernel(float *N, float *M,
        float *P, int Mask_Width, int Width) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;

    float Pvalue = 0;
    int N_start_point = i - (Mask_Width/2);
    for (int j = 0; j < Mask_Width; j++) {
        if (N_start_point + j >= 0 &&
            N_start_point + j < Width) {
            Pvalue += N[N_start_point + j]*M[j];
        }
    }

    P[i] = Pvalue;
}
```

Use a register

- This kernel forces all elements outside the valid input range to 0

A 1D Convolution Kernel with Boundary Condition Handling

```
__global__ void convolution_1D_basic_kernel(float *N, float *M,
        float *P, int Mask_Width, int Width) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;

    float Pvalue = 0;
    int N_start_point = i - (Mask_Width/2);
    for (int j = 0; j < Mask_Width; j++) {
        if (N_start_point + j >= 0 &&
            N_start_point + j < Width) {
            Pvalue += N[N_start_point + j]*M[j];
        }
    }

    P[i] = Pvalue;
}
```

Use a register

Source of bad performance: 1 floating-point operation per global memory access

- This kernel forces all elements outside the valid input range to 0

2D Convolution

N

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

M

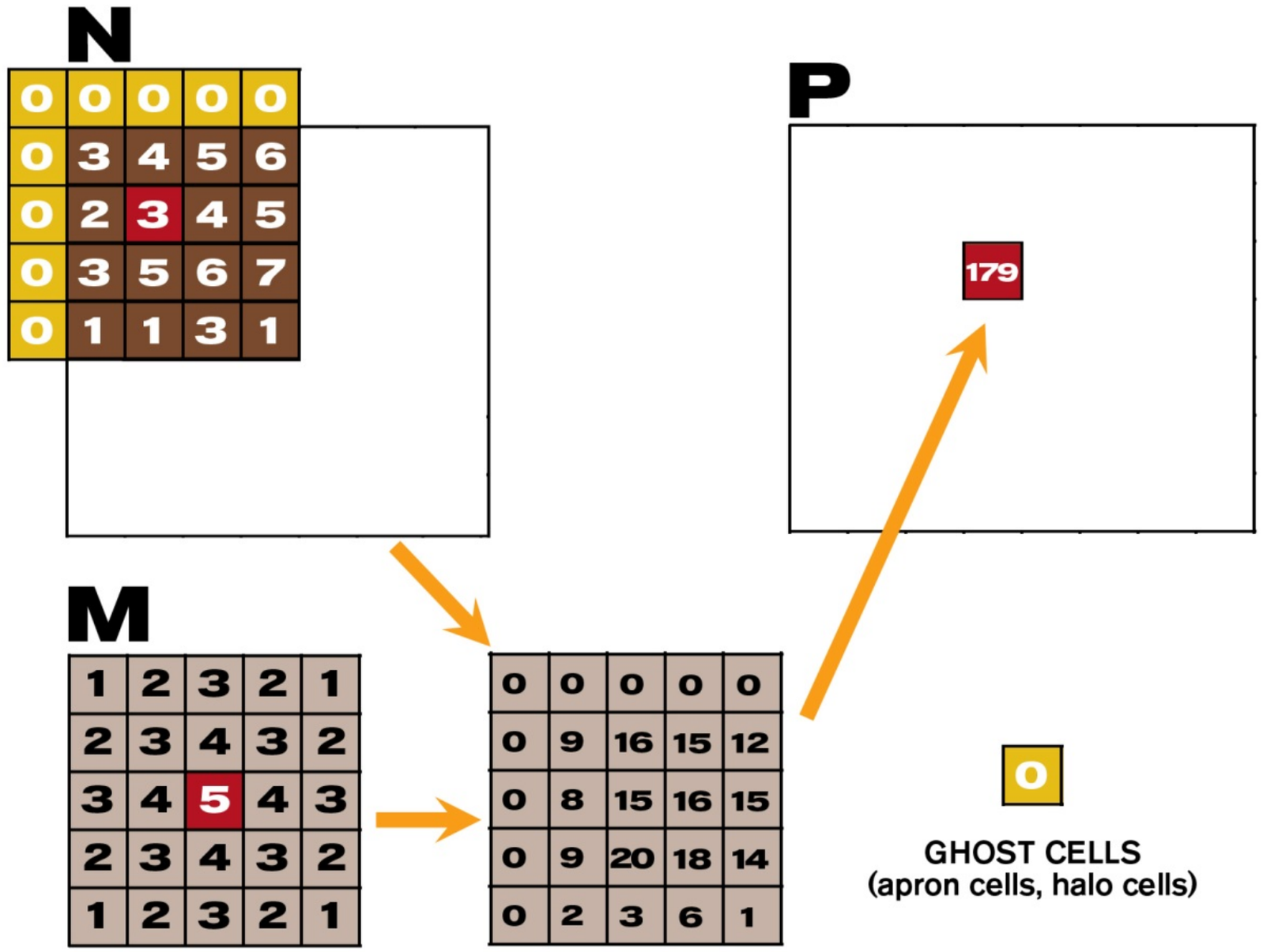
1	2	3	2	1
2	3	4	3	2
3	4	5	4	3
2	3	4	3	2
1	2	3	2	1

1	4	9	8	5
4	9	16	15	12
4	16	25	24	21
8	15	24	21	16
5	12	21	16	5

P

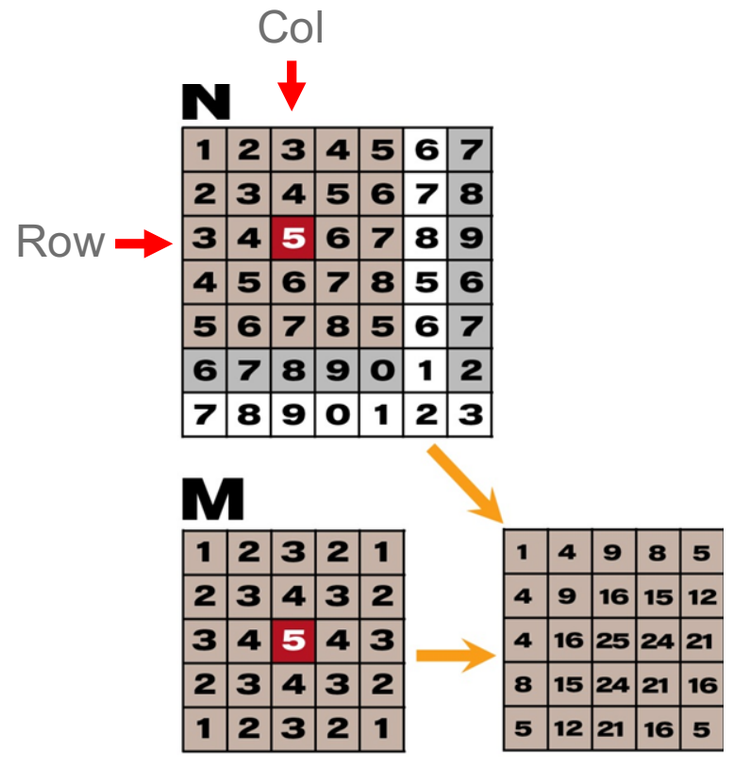
1	2	3	4	5			
2	3	4	5	6			
3	4	321	6	7			
4	5	6	7	8			
5	6	7	8	5			

2D Convolution – Ghost Cells

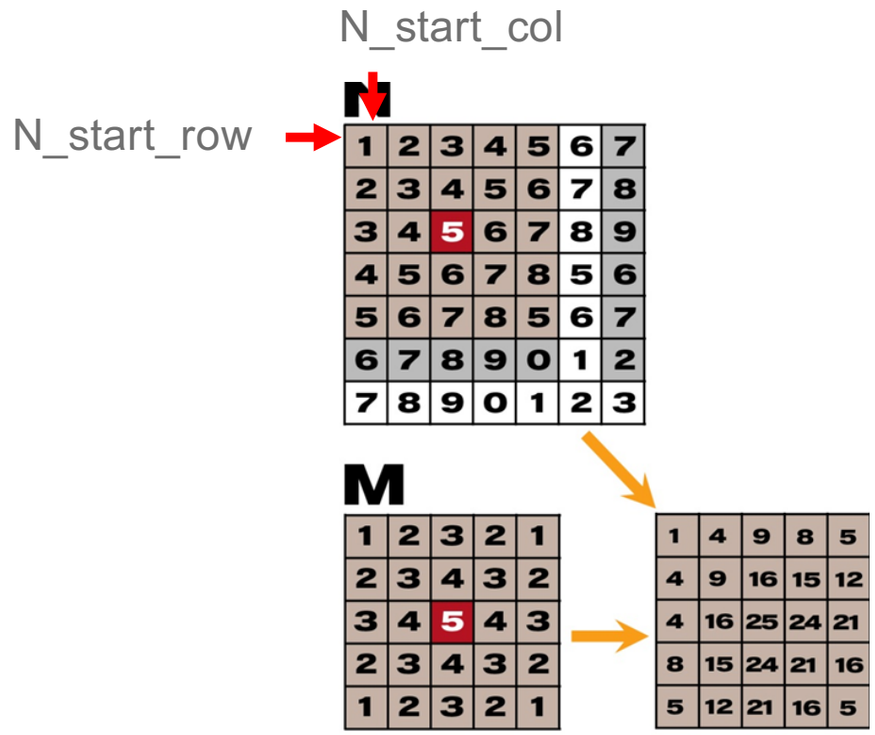


```
__global__
 void convolution_2D_basic_kernel(unsigned char * in, unsigned char * mask, unsigned char * out,
    int maskwidth, int w, int h) {
     int Col  =   blockIdx.x * blockDim.x + threadIdx.x;
     int Row  = blockIdx.y * blockDim.y + threadIdx.y;

     if (Col < w && Row < h) {
         int pixVal = 0;

         N_start_col  = Col -   (maskwidth/2);
         N_start_row = Row - (maskwidth/2);

         // Get the of the surrounding box
         for(int j = 0; j < maskwidth; ++j) {
             for(int k = 0; k < maskwidth; ++k) {
                 int curRow = N_Start_row + j;
                 int curCol = N_start_col + k;
                 // Verify we have a valid image pixel
                 if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
                     pixVal += in[curRow * w + curCol] * mask[j*maskwidth+k];
                 }
             }
         }

         // Write our new pixel value out
         out[Row * w + Col] = (unsigned char)(pixVal);
     }
}
```

```
__global__
 void convolution_2D_basic_kernel(unsigned char * in, unsigned char * mask, unsigned char * out,
    int maskwidth, int w, int h) {
     int Col  =   blockIdx.x * blockDim.x + threadIdx.x;
     int Row  = blockIdx.y * blockDim.y + threadIdx.y;

     if (Col < w && Row < h) {
         int pixVal = 0;

         N_start_col  = Col -   (maskwidth/2);
         N_start_row = Row - (maskwidth/2);

         // Get the of the surrounding box
         for(int j = 0; j < maskwidth; ++j) {
             for(int k = 0; k < maskwidth; ++k) {

                 int curRow = N_Start_row + j;
                 int curCol = N_start_col + k;
                 // Verify we have a valid image pixel
                 if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
                     pixVal += in[curRow * w + curCol] * mask[j*maskwidth+k];
                 }
             }
         }

         // Write our new pixel value out
         out[Row * w + Col] = (unsigned char)(pixVal);
     }
}
```



```
__global__
 void convolution_2D_basic_kernel(unsigned char * in, unsigned char * mask, unsigned char * out,
    int maskwidth, int w, int h) {
     int Col  =   blockIdx.x * blockDim.x + threadIdx.x;
     int Row  = blockIdx.y * blockDim.y + threadIdx.y;

     if (Col < w && Row < h) {
         int pixVal = 0;

         N_start_col  = Col -   (maskwidth/2);
         N_start_row = Row - (maskwidth/2);

         // Get the of the surrounding box
         for(int j = 0; j < maskwidth; ++j) {
             for(int k = 0; k < maskwidth; ++k) {

                 int curRow = N_Start_row + j;
                 int curCol = N_start_col + k;
                 // Verify we have a valid image pixel
                 if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
                     pixVal += in[curRow * w + curCol] * mask[j*maskwidth+k];
                 }
             }
         }

         // Write our new pixel value out
         out[Row * w + Col] = (unsigned char)(pixVal);
     }
}
```



```
__global__
 void convolution_2D_basic_kernel(unsigned char * in, unsigned char * mask, unsigned char * out,
    int maskwidth, int w, int h) {
     int Col  =   blockIdx.x * blockDim.x + threadIdx.x;
     int Row  = blockIdx.y * blockDim.y + threadIdx.y;

     if (Col < w && Row < h) {
         int pixVal = 0;

         N_start_col  = Col -   (maskwidth/2);
         N_start_row = Row - (maskwidth/2);

         // Get the of the surrounding box
         for(int j = 0; j < maskwidth; ++j) {
             for(int k = 0; k < maskwidth; ++k) {
                 int curRow = N_Start_row + j;
                 int curCol = N_start_col + k;
                 // Verify we have a valid image pixel
                 if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
                     pixVal += in[curRow * w + curCol] * mask[j*maskwidth+k];
                 }
             }
         }

         // Write our new pixel value out
         out[Row * w + Col] = (unsigned char)(pixVal);
     }
}
```

```
__global__
 void convolution_2D_basic_kernel(unsigned char * in, unsigned char * mask, unsigned char * out,
    int maskwidth, int w, int h) {
     int Col  =   blockIdx.x * blockDim.x + threadIdx.x;
     int Row  = blockIdx.y * blockDim.y + threadIdx.y;

     if (Col < w && Row < h) {
         int pixVal = 0;

         N_start_col  = Col -   (maskwidth/2);
         N_start_row = Row - (maskwidth/2);

         // Get the of the surrounding box
         for(int j = 0; j < maskwidth; ++j) {
             for(int k = 0; k < maskwidth; ++k) {

                 int curRow = N_Start_row + j;
                 int curCol = N_start_col + k;
                 // Verify we have a valid image pixel
                 if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
                     pixVal += in[curRow * w + curCol] * mask[j*maskwidth+k];
                 }
             }
         }

         // Write our new pixel value out
         out[Row * w + Col] = (unsigned char)(pixVal);
     }
}
```

```
__global__
 void convolution_2D_basic_kernel(unsigned char * in, unsigned char * mask, unsigned char * out,
    int maskwidth, int w, int h) {
     int Col  =   blockIdx.x * blockDim.x + threadIdx.x;
     int Row  = blockIdx.y * blockDim.y + threadIdx.y;

     if (Col < w && Row < h) {
         int pixVal = 0;

         N_start_col  = Col -   (maskwidth/2);
         N_start_row = Row - (maskwidth/2);

         // Get the of the surrounding box
         for(int j = 0; j < maskwidth; ++j) {
             for(int k = 0; k < maskwidth; ++k) {

                 int curRow = N_Start_row + j;
                 int curCol = N_start_col + k;
                 // Verify we have a valid image pixel
                 if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
                     pixVal += in[curRow * w + curCol] * mask[j*maskwidth+k];
                 }
             }
         }

         // Write our new pixel value out
         out[Row * w + Col] = (unsigned char)(pixVal);
     }
}
```

Source of bad performance: 1 floating-point operation per global memory access

Improving convolution kernel

- Use tiling for the N array element
- Use constant memory for the M mask
  - it’s typically small and is not changed
  - can be read by all threads of the grid

```
#define MAX_MASK_WIDTH 10
__constant__ float M[MAX_MASK_WIDTH];

cudaMemcpyToSymbol(M, M_h, Mask_Width*sizeof(float));
```

Improving convolution kernel

- Use tiling for the N array element
- Use constant memory for the M mask
  - it’s typically small and is not changed
  - can be read by all threads of the grid

```
#define MAX_MASK_WIDTH 10
__constant__ float M[MAX_MASK_WIDTH];
```

global variable

```
cudaMemcpyToSymbol(M, M_h, Mask_Width*sizeof(float));
```

Convolution with constant memory

```
__global__ void convolution_1D_basic_kernel(float *N, float *P,
                                            int Mask_Width, int Width) {

  int i = blockIdx.x*blockDim.x + threadIdx.x;

  float Pvalue = 0;

  int N_start_point = i - (Mask_Width/2);

  for (int j = 0; j < Mask_Width; j++) {

    if (N_start_point + j >= 0 && N_start_point + j < Width) {

      Pvalue += N[N_start_point + j]*M[j];

    }

  }

  P[i] = Pvalue;

}
```

Convolution with constant memory

```
__global__ void convolution_1D_basic_kernel(float *N, float *P,
                                            int Mask_Width, int Width) {

  int i = blockIdx.x*blockDim.x + threadIdx.x;

  float Pvalue = 0;

  int N_start_point = i - (Mask_Width/2);

  for (int j = 0; j < Mask_Width; j++) {

    if (N_start_point + j >= 0 && N_start_point + j < Width) {

      Pvalue += N[N_start_point + j]*M[j];

    }
  }

  P[i] = Pvalue;

}
```

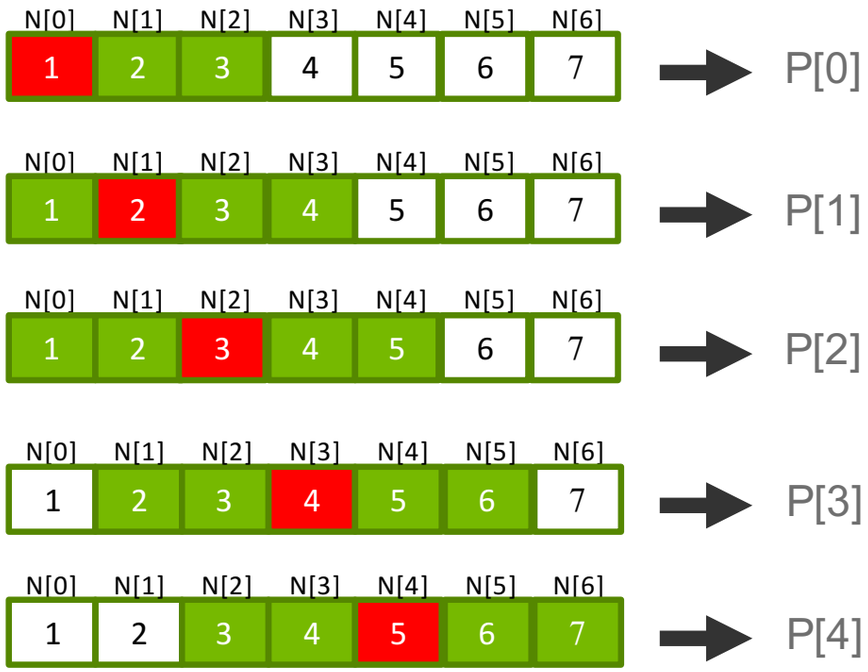
2 floating-point operations per global memory access (**N**)

UNIVERSITÀ
DEGLI STUDI
FIRENZE

CUDA: parallel patterns - convolution & tiling

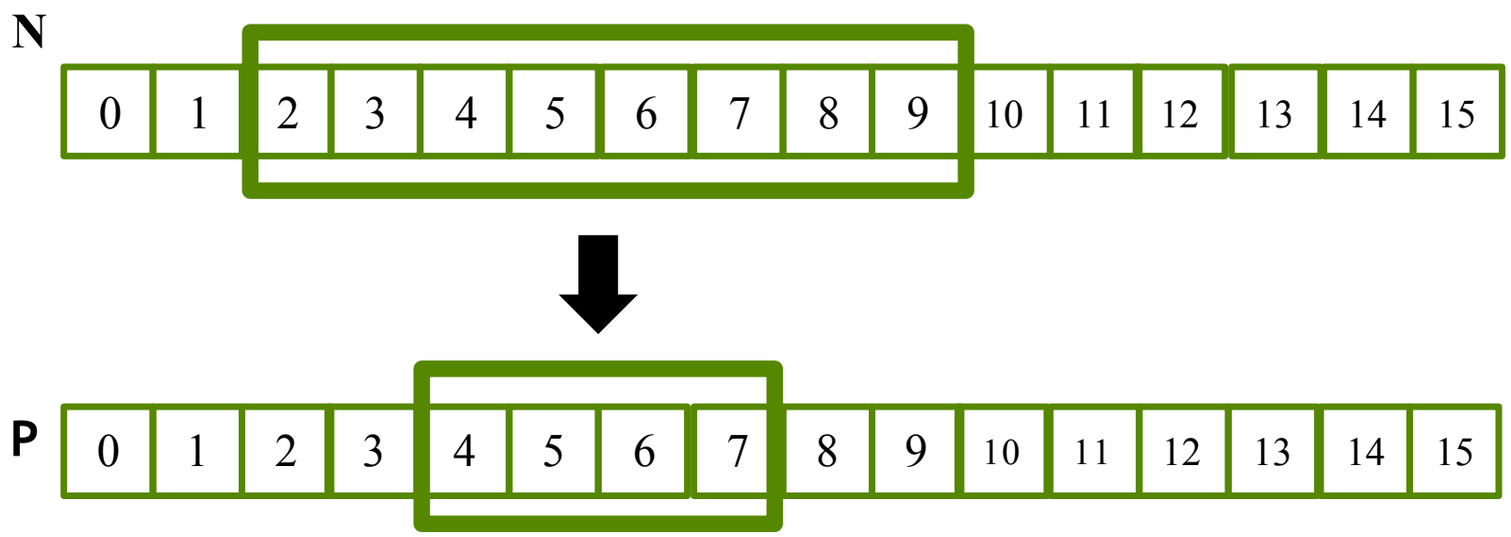
Tiling & convolution

- Calculation of adjacent output elements involve shared input elements
  - E.g., N[2] is used in calculation of P[0], P[1], P[2]. P[3 and P[5] assuming a 1D convolution Mask_Width of width 5
- We can load all the input elements required by all threads in a block into the shared memory to reduce global memory accesses

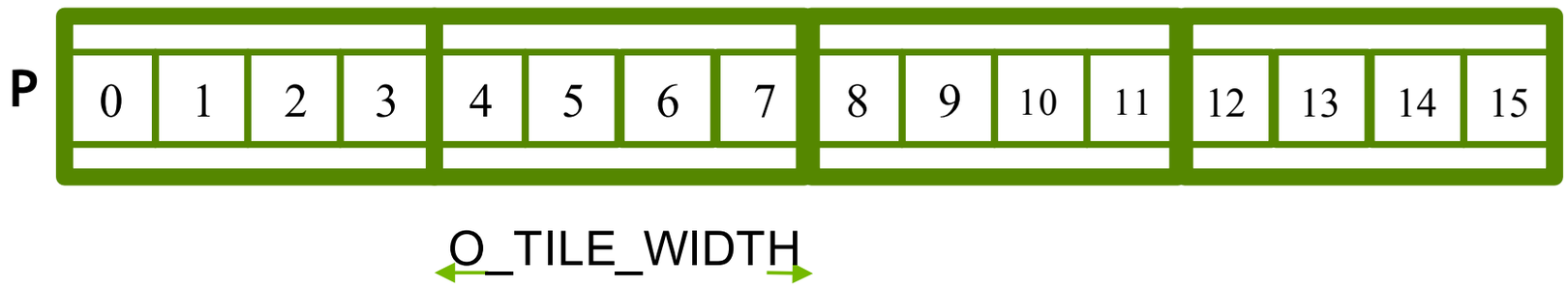


Input Data Needs

- Assume that we want to have each block to calculate T output elements
  - T + Mask_Width -1 input elements are needed to calculate T output elements
  - T + Mask_Width -1 is usually not a multiple of T, except for small T values
  - T is usually significantly larger than Mask_Width

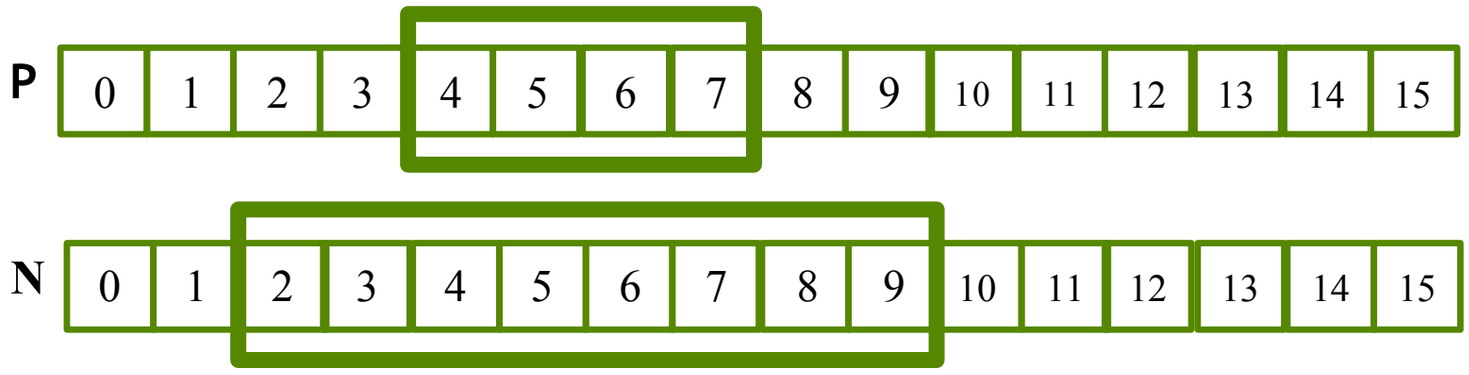


Definition – output tile



- Each thread block calculates an output tile
- Each output tile width is O_TILE_WIDTH
- For each thread:
  - O_TILE_WIDTH is 4 in this example

Definition - Input Tiles

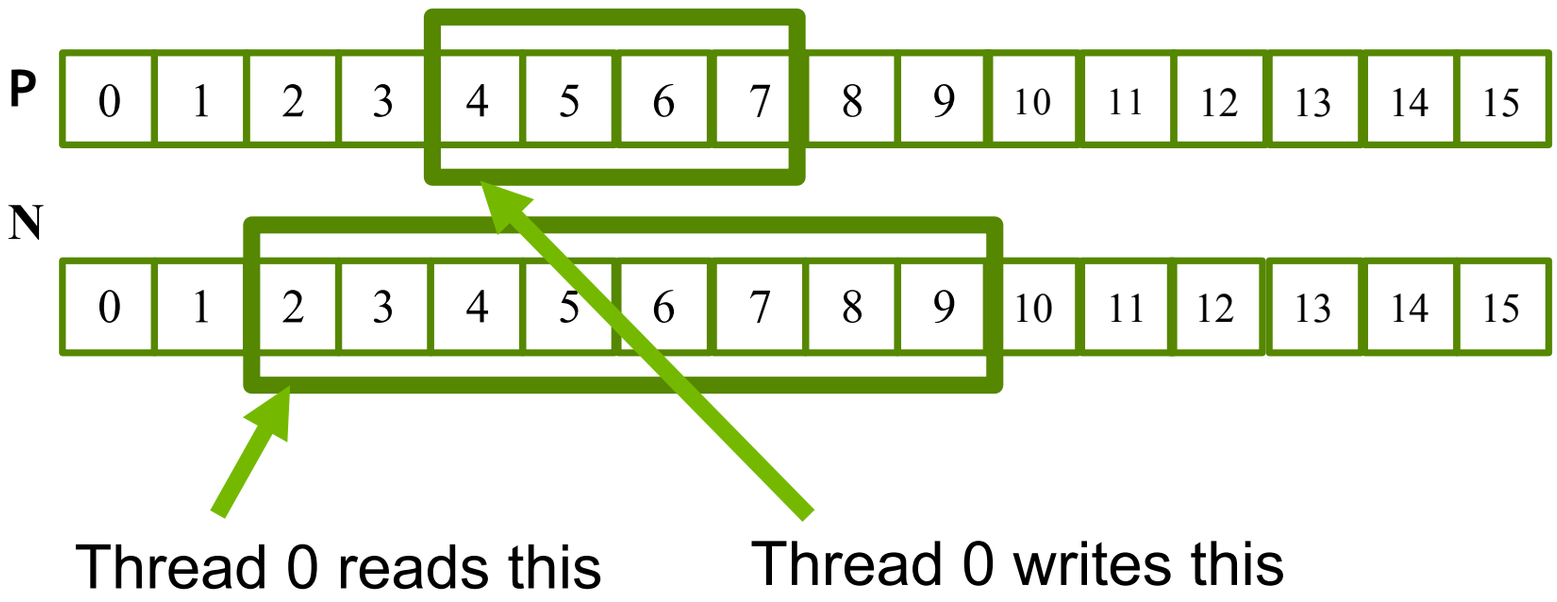


- Each input tile has all values needed to calculate the corresponding output tile.

Two Design Options

- Design 1: The size of each thread block matches the size of an output tile
  - All threads participate in calculating output elements
  - blockDim.x would be 4 in our example
  - Some threads need to load more than one input element into the shared memory
- Design 2: The size of each thread block matches the size of an input tile
  - Some threads will not participate in calculating output elements
  - blockDim.x would be 8 in our example
  - Each thread loads one input element into the shared memory

Thread to Input and Output Data Mapping



- For each thread:
  - index_i = index_o – n
- where n is Mask_Width /2
  - n is 2 in this example

Loading input tiles

All threads participate:

```
float output = 0.0f;

if((index_i >= 0) && (index_i < Width)) {
    Ns[tx] = N[index_i];
} else {
    Ns[tx] = 0.0f;
}
```

Calculating output

- Some threads do not participate: Only Threads `0` through `O_TILE_WIDTH-1` participate in calculation of output.

```
index_o = blockIdx.x*O_TILE_WIDTH + threadIdx.x

if (threadIdx.x < O_TILE_WIDTH){
    output = 0.0f;
    for(j = 0; j < Mask_Width; j++) {
        output += M[j] * Ns[j+threadIdx.x];
    }
    P[index_o] = output;
}
```

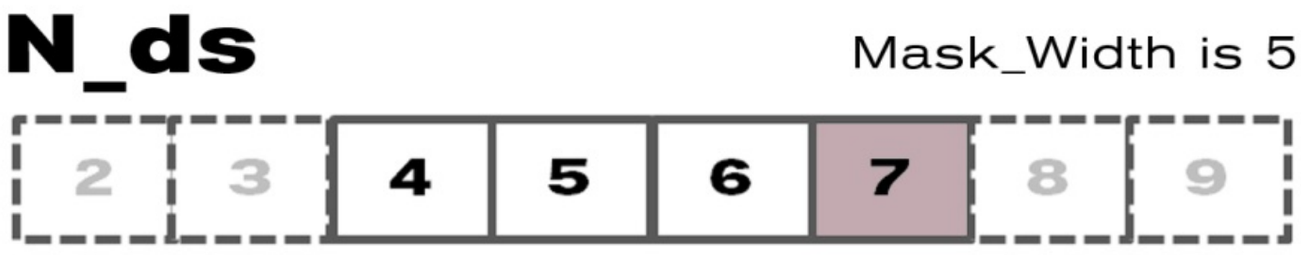
Setting Block Size

```
#define O_TILE_WIDTH 1020

#define BLOCK_WIDTH (O_TILE_WIDTH + 4)
```

```
dim3 dimBlock(BLOCK_WIDTH,1, 1);

dim3 dimGrid((Width-1)/O_TILE_WIDTH+1, 1, 1)
```

- The Mask_Width is 5 in this example
- In general, block width should be
  - output tile width + (mask width-1)

Shared Memory Data Reuse



Element 2 is used by thread 4 (1X)

Element 3 is used by threads 4, 5 (2X)

Element 4 is used by threads 4, 5, 6 (3X)

Element 5 is used by threads 4, 5, 6, 7 (4X)

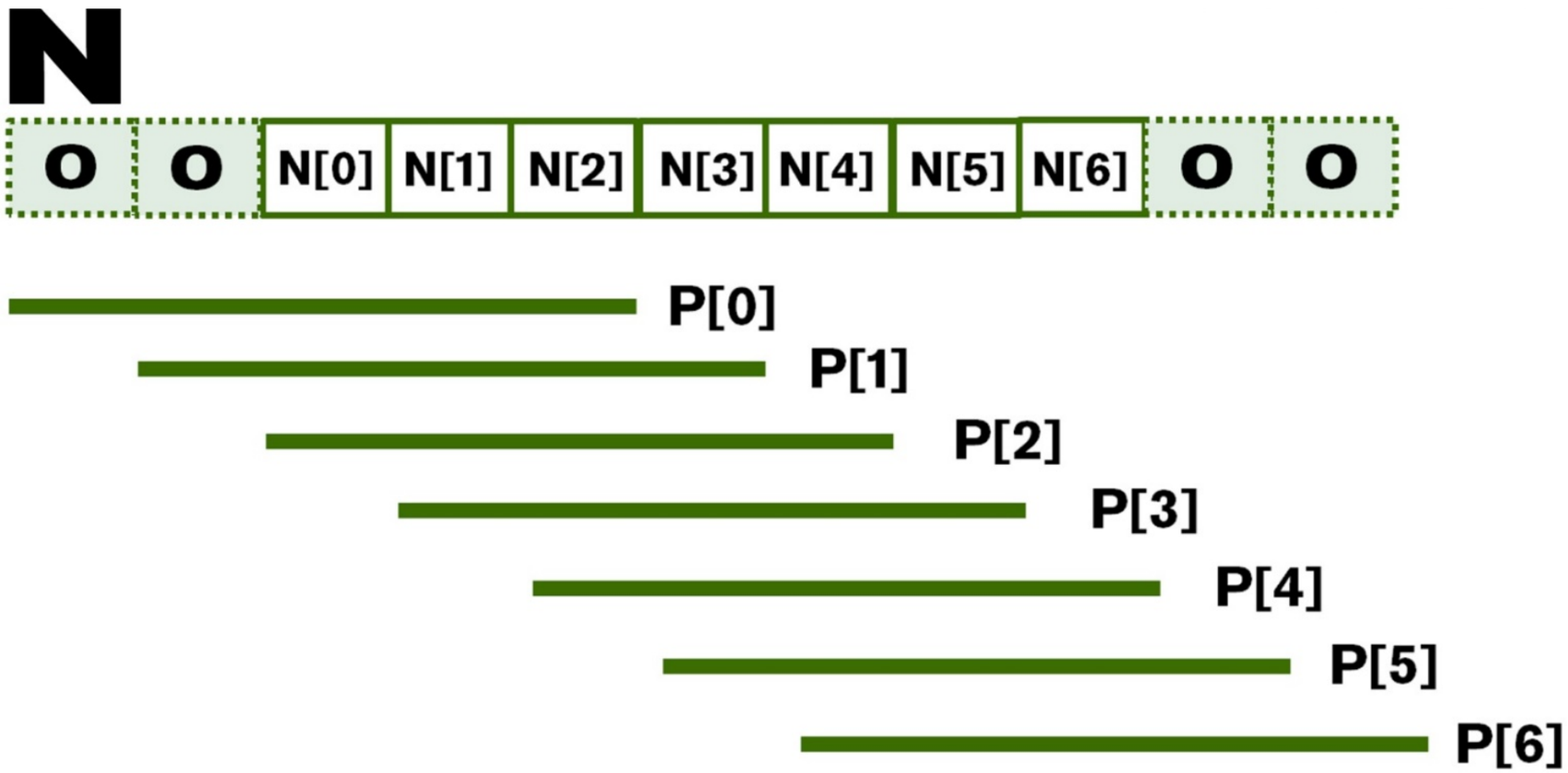
Element 6 is used by threads 4, 5, 6, 7 (4X)

Element 7 is used by threads 5, 6, 7 (3X)

Element 8 is used by threads 6, 7 (2X)

Element 9 is used by thread 7 (1X)

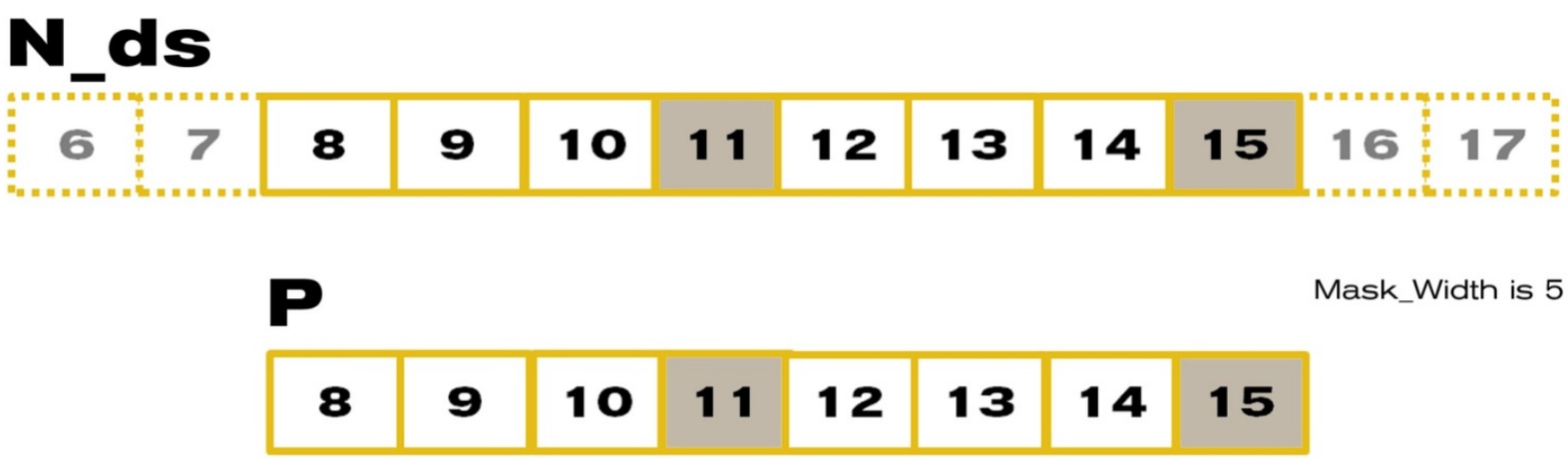
Ghost/Halo cells



UNIVERSITÀ
DEGLI STUDI
FIRENZE

Evaluating tiling

An 8-element Convolution Tile

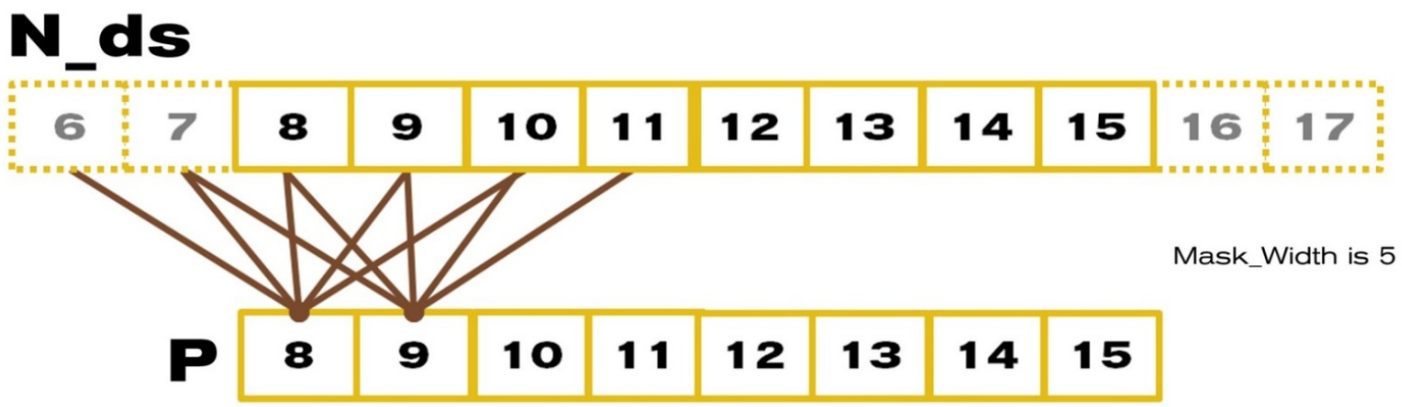


- For Mask_Width=5, we load 8+5-1=12 elements (12 memory loads)

Evaluating reuse

- Each output P element uses 5 N elements:

  P[8] uses N[6], N[7], N[8], N[9], N[10]
  P[9] uses N[7], N[8], N[9], N[10], N[11]
  P[10] use N[8], N[9], N[10], N[11], N[12]
  …
  P[14] uses N[12], N[13], N[14], N[15], N[16]
  P[15] uses N[13], N[14], N[15], N[16], N[17]

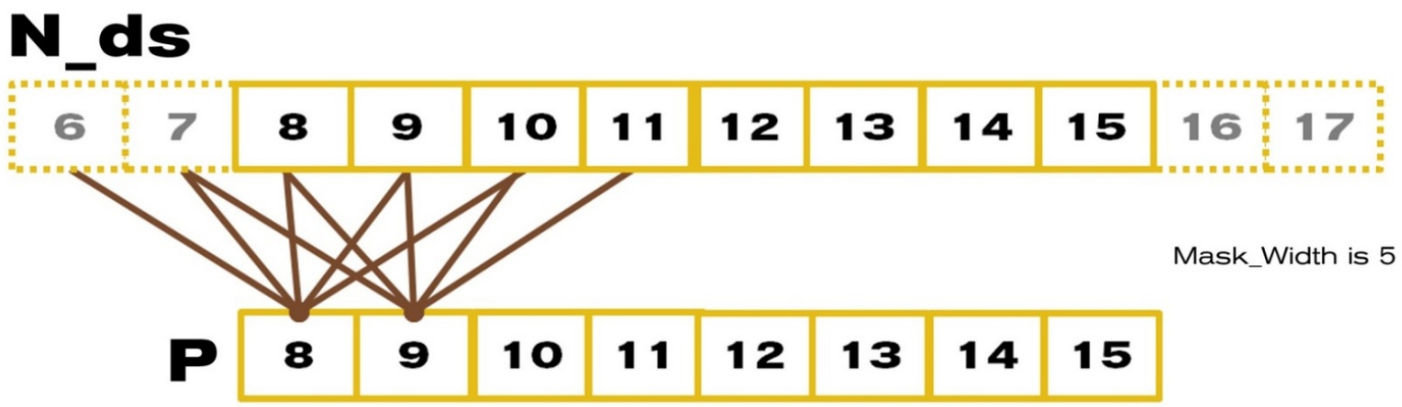


Evaluating reuse

- Each output P element uses 5 N elements:

  P[8] uses N[6], N[7], N[8], N[9], N[10]
  P[9] uses N[7], N[8], N[9], N[10], N[11]
  P[10] use N[8], N[9], N[10], N[11], N[12]

(8+5-1)=12 elements loaded
8*5 global memory accesses replaced by shared memory accesses
This gives a bandwidth reduction of 40/12=3.3



General 1D tiled convolution

- O_TILE_WIDTH+MASK_WIDTH -1 elements loaded for each input tile

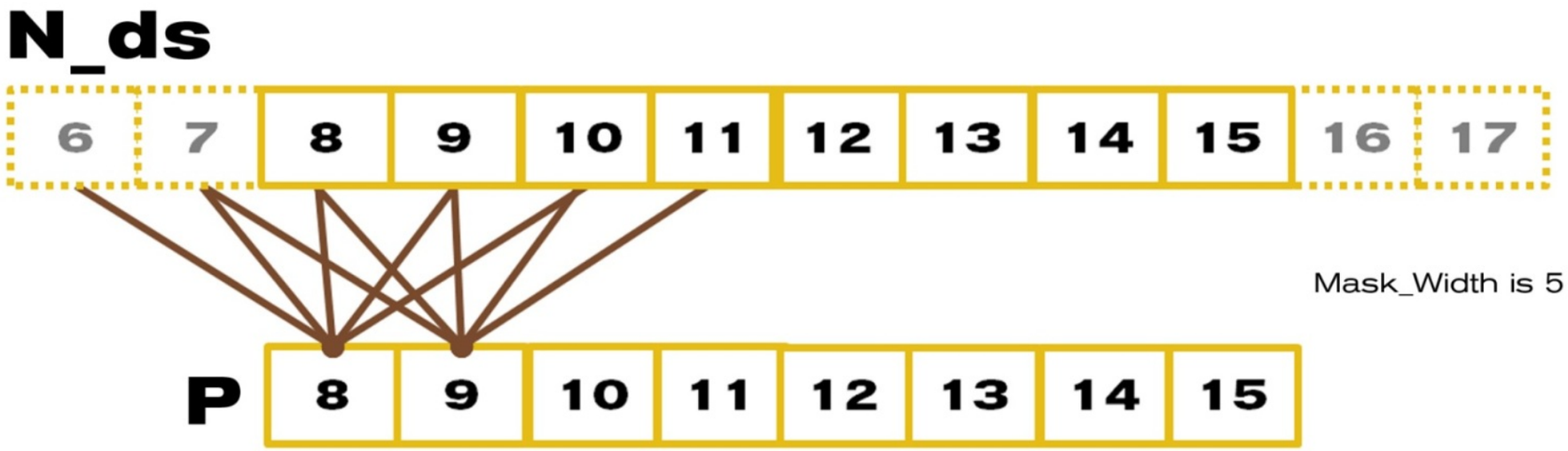
- O_TILE_WIDTH*MASK_WIDTH global memory accesses replaced by shared memory accesses

- This gives a reduction factor of

  (O_TILE_WIDTH*MASK_WIDTH)/
  (O_TILE_WIDTH+MASK_WIDTH-1)

- This ignores ghost elements in edge tiles.

Another Way to Look at Reuse



- N[6] is used by P[8] (1X)
  N[7] is used by P[8], P[9] (2X)
  N[8] is used by P[8], P[9], P[10] (3X)
  N[9] is used by P[8], P[9], P[10], P[11] (4X)
  N10 is used by P[8], P[9], P[10], P[11], P[12] (5X)
  … (5X)
  N[14] is used by P[12], P[13], P[14], P[15] (4X)
  N[15] is used by P[13], P[14], P[15] (3X)

Another Way to Look at Reuse

- The total number of global memory accesses to the (8+5-1)=12 elements of N that is replaced by shared memory accesses is:

  1+2+3+4+5*(8-5+1)+4+3+2+1 = 10+20+10 = 40

- So the reduction is 40/12 = 3.3

General 1D tiling

- The total number of global memory accesses to the input tile can be calculated as

- 1 + 2+…+ MASK_WIDTH-1 + MASK_WIDTH*(O_TILE_WIDTH-MASK_WIDTH+1) + MASK_WIDTH-1 + …+ 2 + 1
- = MASK_WIDTH * (MASK_WIDTH-1) + MASK_WIDTH *
- (O_TILE_WIDTH-MASK_WIDTH+1)
- = MASK_WIDTH * O_TILE_WIDTH

- For a total of O_TILE_WIDTH + MASK_WIDTH -1 input tile elements

Examples of Bandwidth Reduction for 1D

- The reduction ratio is:
- MASK_WIDTH * O_TILE_WIDTH / (O_TILE_WIDTH+MASK_WIDTH-1)

O_TILE_WIDTH	16	32	64	128	256
MASK_WIDTH= 5	4.0	4.4	4.7	4.9	4.9
MASK_WIDTH = 9	6.0	7.2	8.0	8.5	8.7

2D convolution tiles

- (O_TILE_WIDTH+MASK_WIDTH-1)^2 input elements need to be loaded into shared memory

- The calculation of each output element needs to access MASK_WIDTH^2 input elements

- O_TILE_WIDTH^2 * MASK_WIDTH^2 global memory accesses are converted into shared memory accesses

- The reduction ratio is

  - O_TILE_WIDTH^2 * MASK_WIDTH^2 / (O_TILE_WIDTH+MASK_WIDTH-1)^2

Bandwidth Reduction for 2D

- The reduction ratio is:

  O_TILE_WIDTH^2 * MASK_WIDTH^2 /
  (O_TILE_WIDTH+MASK_WIDTH-1)^2

O_TILE_WIDTH	8	16	32	64
MASK_WIDTH = 5	11.1	16	19.7	22.1
MASK_WIDTH = 9	20.3	36	51.8	64

- Tile size has significant effect on of the memory bandwidth reduction ratio.

- This often argues for larger shared memory size

UNIVERSITÀ
DEGLI STUDI
FIRENZE

CUDA: parallel patterns - map/ gather/scatter

Map

- Takes an input list *i*
- Applies a function *f*
- Writes the results in a list *o*, by applying *f* to all the members of *i*
- A CUDA kernel where *i* and *o* are memory locations determined by `threadIdx`... implements this pattern

Gather

- Multiple inputs and single coalesced output
- Might have sequential loading or random access
  - Affect memory performance, e.g. if thread 1 gets input 0 and 1, thread 2 gets input 2 and 3, and so on… we have coalesced memory access. With random access no.
- Differs to map due to multiple inputs

Scatter

- Reads from a single input and writes to one or many
- Can be implemented in CUDA using atomics
- Write pattern will determine performance
  - e.g. do we have write collisions ? Random access write ?

UNIVERSITÀ
DEGLI STUDI
FIRENZE

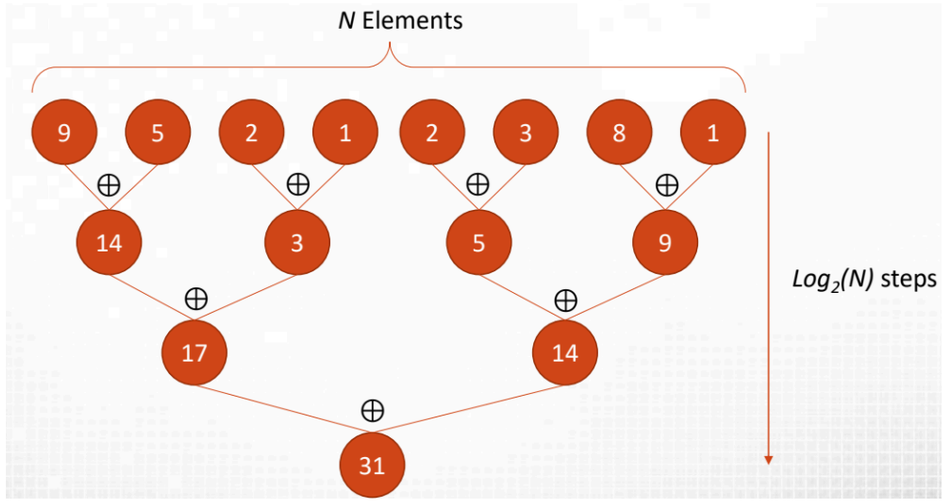
CUDA: parallel patterns - reduction

Reduction

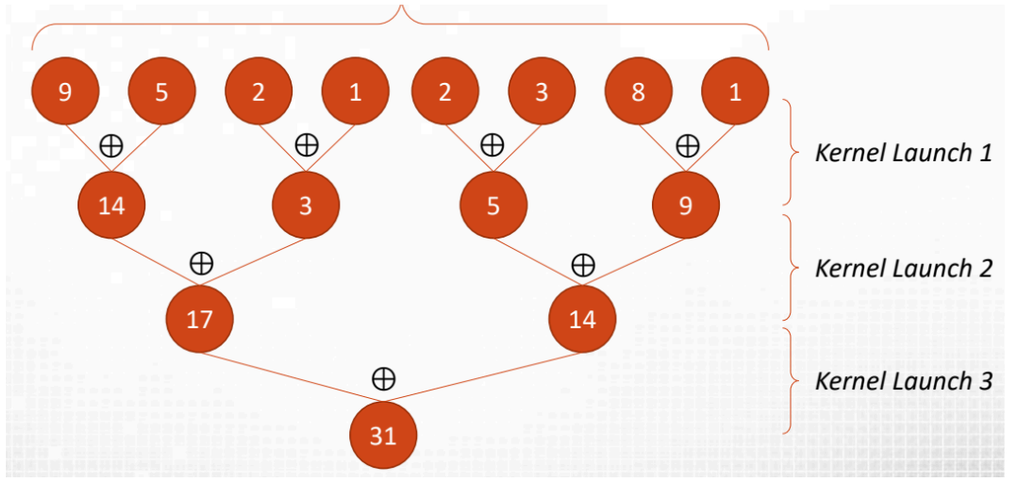
- A reduction is where all elements of a set have a common binary associative operator ($\oplus$) applied to them to “reduce” the set to a single value
- Binary associative = order in which operations is performed on set does not matter
  - Most obvious example is addition (Summation)
  - Other examples, Maximum, Minimum, product

Tree based

- At each step data is reduced by a factor of 2

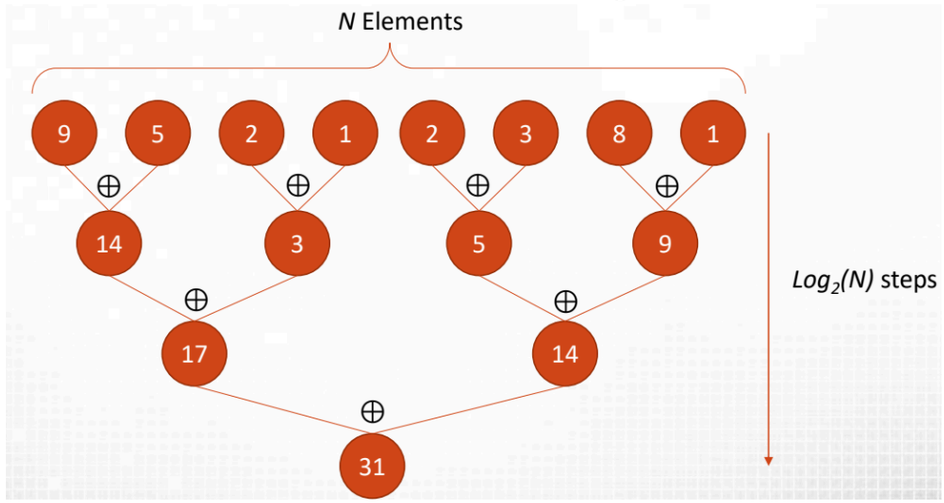


- In CUDA there is no global synchronisation so we need to split the execution into multiple stages

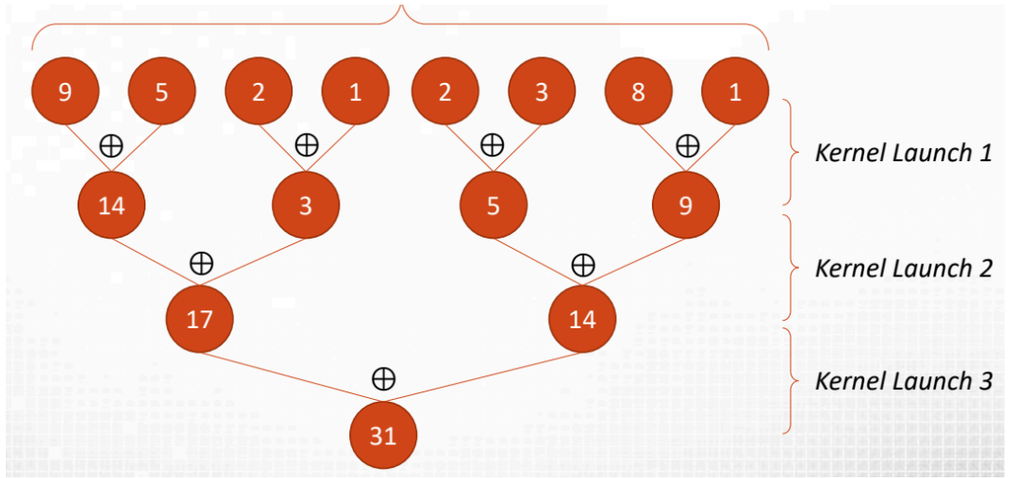


N-1 operations in $\log_2(N)$ steps
Avg. parallelism: $(N-1)/\log_2(N)$
For N=1000000 avg. parallelism is 50000, but peak resource req. is 500000

- At each step data is reduced by a factor of 2



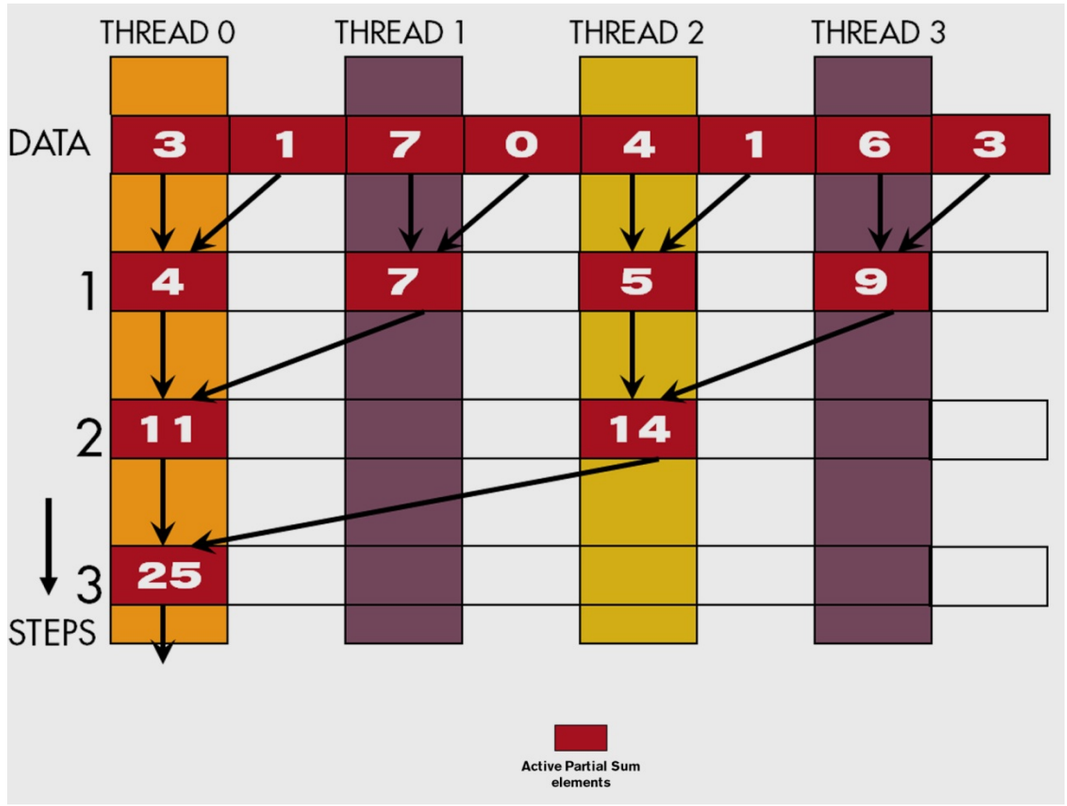
- In CUDA there is no global synchronisation so we need to split the execution into multiple stages



Parallel sum reduction

- Parallel implementation
  - Each thread adds two values in each step
  - Recursively halve # of threads
  - 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 is used to hold a partial sum vector
  - Initially, the partial sum vector is simply the original vector
  - Each step brings the partial sum vector closer to the sum
  - The final sum will be in element 0 of the partial sum vector
  - Reduces global memory traffic due to reading and writing partial sum values
  - Thread block size limits *n* to be less than or equal to 2,048

A Parallel Sum Reduction Example



- Step 1 - Stride 1
- Step 2 - Stride 2
- Step 3 - Stride 4

- Naïve thread-to-data mapping
  - Each thread is responsible for an even-index location of the partial sum vector (location of responsibility)
  - After each step, half of the threads are no longer needed
  - One of the inputs is always from the location of responsibility
  - In each step, one of the inputs comes from an increasing distance away

A Simple Thread Block Design

- Each thread block takes `2*BlockDim.x` input elements

- Each thread loads 2 elements into shared memory

```
__shared__ float partialSum[2*BLOCK_SIZE];

unsigned int t = threadIdx.x;
unsigned int start = 2*blockIdx.x*blockDim.x;
partialSum[t] = input[start + t];
partialSum[blockDim+t] = input[start + blockDim.x +t];
```

Reduction steps

```
for (unsigned int stride = 1; stride <= blockDim.x;
                 stride *= 2) {
  __syncthreads();
  if (t % stride == 0)
    partialSum[2*t] += partialSum[2*t+stride];
}
```

- `__syncthreads()` is needed to ensure that all elements of each version of partial sums have been generated before we proceed to the next step

Reduction steps

```
for (unsigned int stride = 1; stride <= blockDim.x;
                 stride *= 2) {
  __syncthreads();
  if (t % stride == 0)
      partialSum[2*t] += partialSum[2*t+stride];
 }
```

- At the end of the kernel, Thread 0 in each block writes the sum of the thread block in `partialSum[0]` into a vector indexed by the `blockIdx.x`
- There can be a large number of such sums if the original vector is very large
  - The host code may iterate and launch another kernel
- If there are only a small number of sums, the host can simply transfer the data back and add them together
- Alternatively, Thread 0 of each block could use atomic operations to accumulate into a global sum variable.

Reduction steps

```
if (t==0) output[blockIdx.x]=partialSum[0];
```

```
for (unsigned int stride = 1; stride <= blockDim.x;
                 stride *= 2) {
  __syncthreads();
  if (t % stride == 0)
    partialSum[2*t] += partialSum[2*t+stride];
}
```

- At the end of the kernel, Thread 0 in each block writes the sum of the thread block in `partialSum[0]` into a vector indexed by the `blockIdx.x`
- There can be a large number of such sums if the original vector is very large
  - The host code may iterate and launch another kernel
- If there are only a small number of sums, the host can simply transfer the data back and add them together
- Alternatively, Thread 0 of each block could use atomic operations to accumulate into a global sum variable.

Reduction steps

```
if (t==0) output[blockIdx.x]=partialSum[0];
```

```
for (unsigned int stride = 1; stride <= blockDim.x;
                 stride *= 2) {
  __syncthreads();
  if (t % stride == 0)
    partialSum[2*t] += partialSum[2*t+stride];
}
```

Problem: highly divergent branching: very poor performance

- At the end of the kernel, Thread 0 in each block writes the sum of the thread block in `partialSum[0]` into a vector indexed by the `blockIdx.x`
- There can be a large number of such sums if the original vector is very large
  - The host code may iterate and launch another kernel
- If there are only a small number of sums, the host can simply transfer the data back and add them together
- Alternatively, Thread 0 of each block could use atomic operations to accumulate into a global sum variable.

Problems

- In each iteration, 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 still consume execution resources
- Half or fewer of threads will be executing after the first step
  - All odd-index threads are disabled after first step
  - After the 5th step, entire warps in each block will fail the if test, poor resource utilization but no divergence
    - This can go on for a while, up to 6 more steps (stride = 32, 64, 128, 256, 512, 1024), where each active warp only has one productive thread until all warps in a block retire

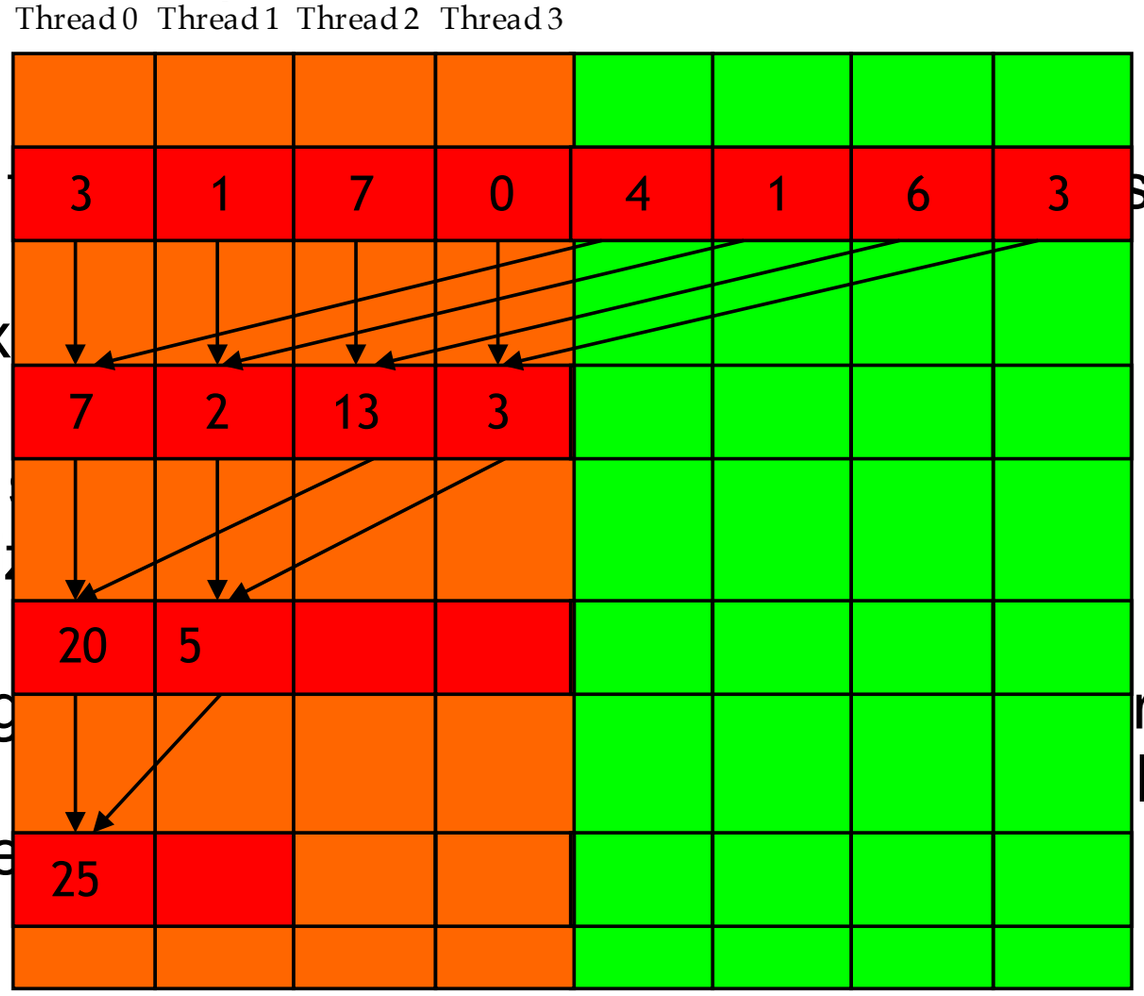
Problems

- In e d for eac
  - Solutions:
    - Change index usage to improve divergence behavior
    - Compact partial sums into front locations of `partialSum[]` array
    - Keep the active threads consecutive
  - Threads that do not perform addition still consume execution resources

- Half or fewer of threads will be executing after the first step
  - All odd-index threads are disabled after first step
  - After the 5th step, entire warps in each block will fail the if test, poor resource utilization but no divergence
    - This can go on for a while, up to 6 more steps (stride = 32, 64, 128, 256, 512, 1024), where each active warp only has one productive thread until all warps in a block retire

Problems

- In e d for eac
  - Solutions:
    - Change index usage to improve divergence behavior
    - Compact partial sums into front locations of `partialSum[]` array
    - Keep the active threads consecutive
  - Threads that do not perform addition still consume execution resources
- Half or fewer of t step
  - All odd-index
  - After the 5th s the if test, poor resource utiliz
    - This can g ride = 32, 64, 128, 256, ly has one productive



Better reduction kernel

```
for (unsigned int stride = blockDim.x; stride > 0; stride /= 2)
{
  __syncthreads();
  if (t < stride)
    partialSum[t] += partialSum[t+stride];
}
```

- For a 1024 thread block
  - No divergence in the first 5 steps
    - 1024, 512, 256, 128, 64, 32 consecutive threads are active in each step
    - All threads in each warp either all active or all inactive
  - The final 5 steps will still have divergence

Better reduction kernel

```
for (unsigned int stride = blockDim.x; stride > 0; stride /= 2)
{
  __syncthreads();
  if (t < stride)
    partialSum[t] += partialSum[t+stride];
}
```

- For a 1024 thread block
  - No divergence in the first 5 steps
    - 1024, 512, 256, 128, 64, 32 consecutive threads are active in each step
    - All threads in each warp either all active or all inactive
  - The final 5 steps will still have divergence

Further improvements:

- Loop unrolling
- Increasing parallelism
- Perform an initial add during data load

UNIVERSITÀ
DEGLI STUDI
FIRENZE

CUDA: parallel patterns - scan

Scan (prefix sum)

- Scan: computes all partial reduction of a collection
- For every output in a collection, a reduction of the input up to that point is computed
- If the function being used is associative, the scan can be parallelized
- Parallelizing a scan is not obvious at first, because of dependencies to previous iterations in the serial loop
- A parallel scan will require more operations than a serial version

Scan (prefix sum)

- Scan: computes all partial reduction of a collection
- For every output in a collection, a reduction of the input up to that point is computed
- If the function being used is associative, the scan can be parallelized

More formally:

The scan operation takes a binary associative operator $\oplus$, and an array of n elements

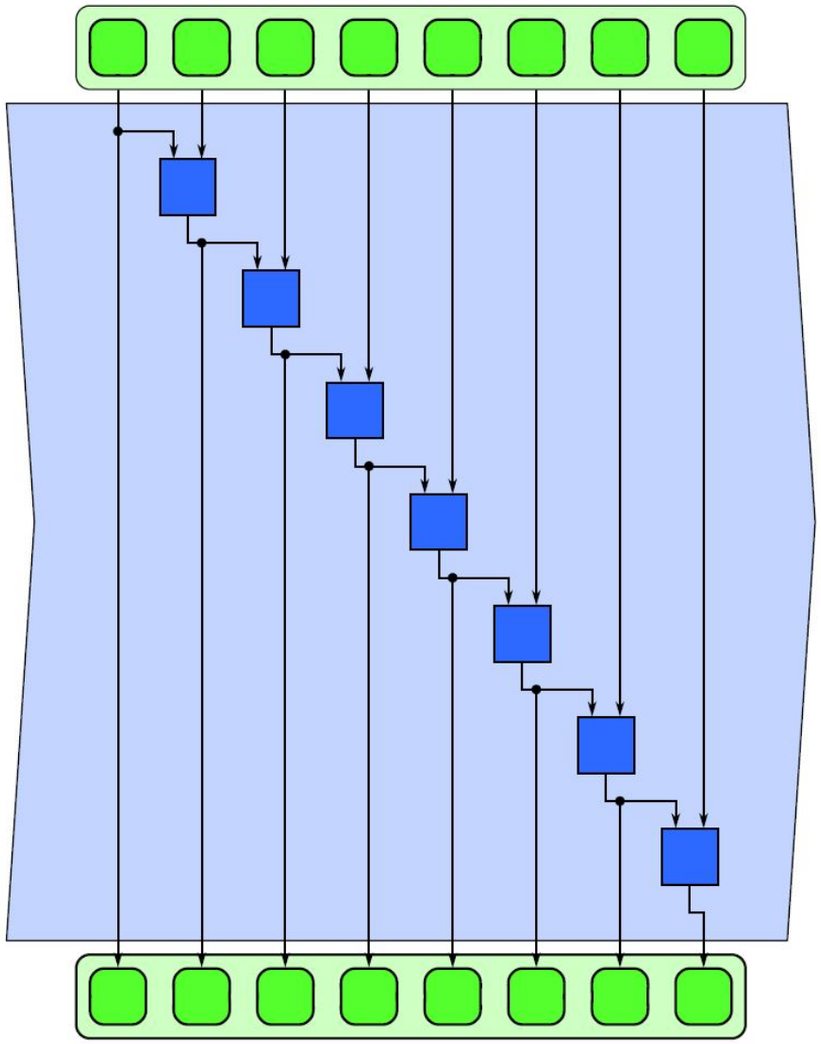
$[x_0, x_1, \ldots, x_{n-1}]$,

and returns the array

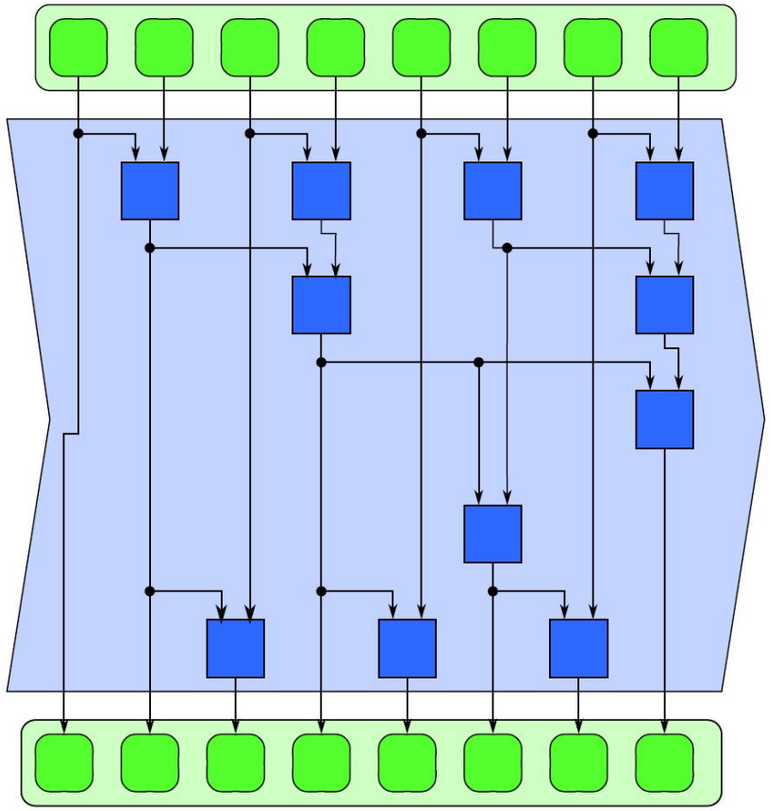
$[x_0, (x_0 \oplus x_1), \ldots, (x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1})]$.

Scan

Serial scan



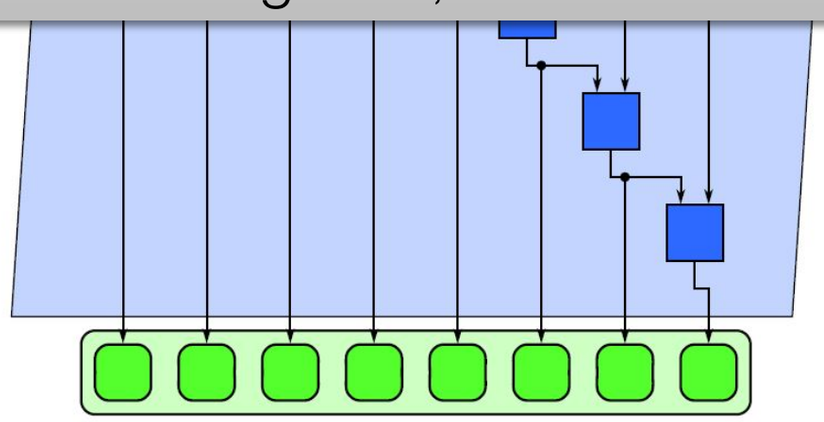
Parallel scan

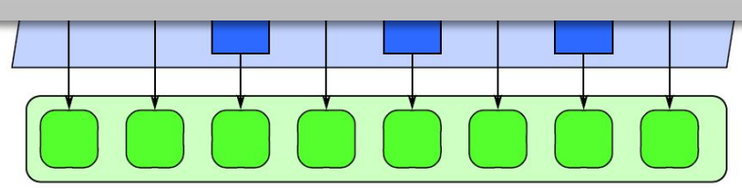


Scan

- Frequently used for parallel work assignment and resource allocation
- A key primitive in many parallel algorithms to convert serial computation into parallel computation
- A foundational parallel computation pattern, useful for many algorithms like:

  - Radix sort
  - Quicksort
  - String comparison
  - Lexical analysis
  - Stream compaction
  - Polynomial evaluation
  - Solving recurrences
  - Tree operations
  - Histograms, ….





Naïve solutions

- A serial version:

```
y[0] = x[0];
for (i = 1; i < Max_i; i++)
  y[i] = y [i-1] + x[i];
```

- is computationally efficient: N additions needed for N elements - O(N)!

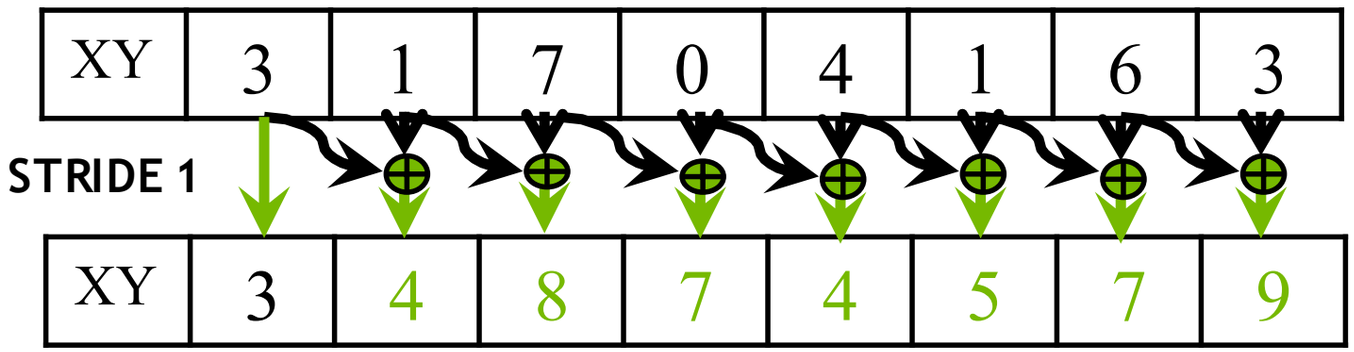
- A parallel version may assign a thread to each element:

```
y0 = x0
y1 = x0 + x1
y2 = x0 + x1 + x2
y3 = x0 + x1 + x2 + x3
```

- is not efficient !

A Better Parallel Scan Algorithm

1. Read input from device global memory to shared memory
2. Iterate log(n) times; stride from 1 to n-1: double stride each iteration
3. Write output from shared memory to device memory



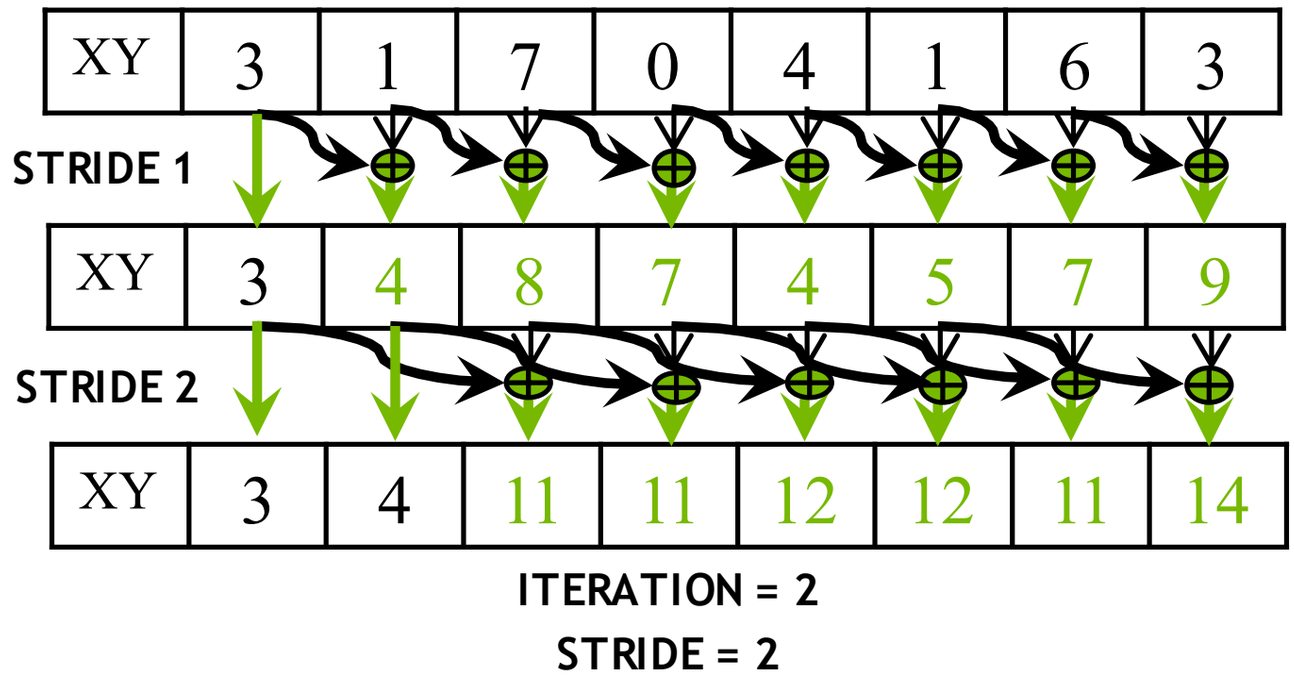
ITERATION = 1

STRIDE = 1

- Active threads stride to n-1 (n-stride threads)
- Thread j adds elements j and j-stride from shared memory and writes result into element j in shared memory
- Requires barrier synchronization, once before read and once before write

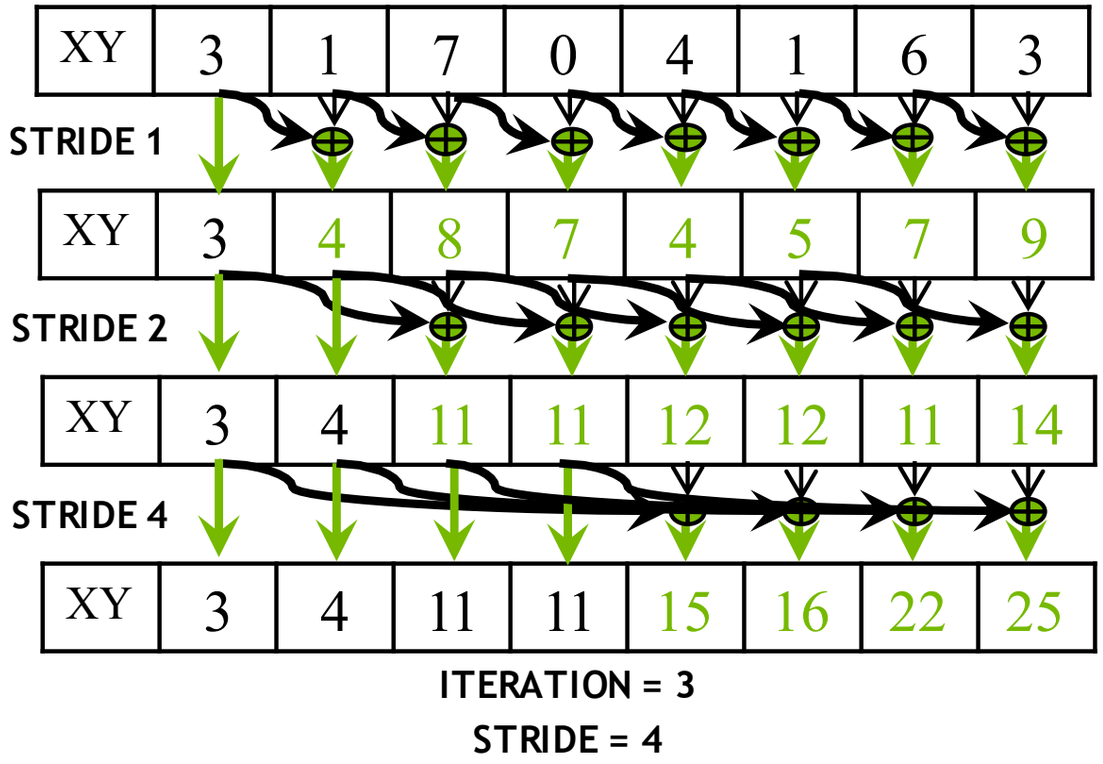
A Better Parallel Scan Algorithm

1. Read input from device global memory to shared memory
2. Iterate log(n) times; stride from 1 to n-1: double stride each iteration
3. Write output from shared memory to device memory



A Better Parallel Scan Algorithm

1. Read input from device global memory to shared memory
2. Iterate log(n) times; stride from 1 to n-1: double stride each iteration
3. Write output from shared memory to device memory



Handling Dependencies

- During every iteration, each thread can overwrite the input of another thread
  - Barrier synchronization to ensure all inputs have been properly generated (i.e. `__syncthreads()`)
  - All threads secure input operand that can be overwritten by another thread
  - Barrier synchronization is required to ensure that all threads have secured their inputs
  - All threads perform addition and write output

```
__global__ void work_inefficient_scan_kernel(float *X, float *Y, int
InputSize) {
    __shared__ float XY[SECTION_SIZE];

    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < InputSize) {
        XY[threadIdx.x] = X[i];
    }

    // the code below performs iterative scan on XY
    for (unsigned int stride = 1; stride < blockDim.x; stride *= 2) {
        __syncthreads();
        if(threadIdx.x >= stride)
            XY[threadIdx.x] += XY[threadIdx.x - stride];
    }

    __syncthreads();
    if (i < InputSize) {
        Y[i] = XY[threadIdx.x];
    }

}
```

SECTION_SIZE = block size

```
__global__ void work_inefficient_scan_kernel(float *X, float *Y, int
InputSize) {
    __shared__ float XY[SECTION_SIZE];

    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < InputSize) {
        XY[threadIdx.x] = X[i];
    }

    // the code below performs iterative scan on XY
    for (unsigned int stride = 1; stride < blockDim.x; stride *= 2) {
        __syncthreads();
        if(threadIdx.x >= stride)
            XY[threadIdx.x] +=
    }

    __syncthreads();
    if (i < InputSize) {
        Y[i] = XY[threadIdx.x]
    }
}
```

SECTION_SIZE = block size

- This Scan executes $\log_2(n)$ parallel iterations with n=SECTION_SIZE
  - The iterations do (n-1), (n-2), (n-4),..(n- n/2) adds each
  - Total adds: n * $\log_2(n)$ - (n-1) → O(n*$\log_2(n)$) work
- This scan algorithm is not work efficient
  - Sequential scan algorithm does n adds
  - A factor of $\log_2(n)$ can hurt: 10× for 1024 elements!

A parallel algorithm can be slower than a sequential one when execution resources are saturated from low work efficiency

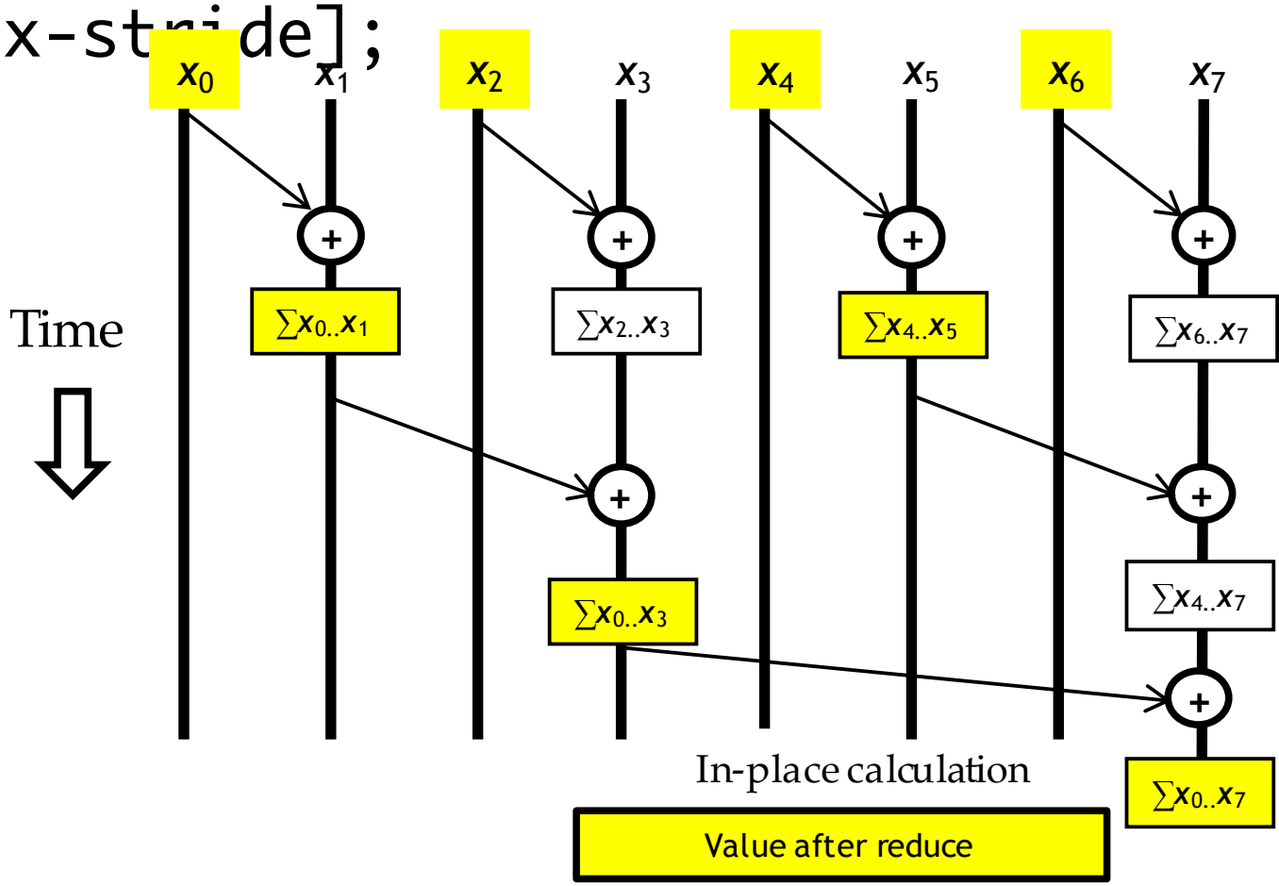
Improving efficiency

- Balanced Trees
  - Form a balanced binary tree on the input data and sweep it to and from the root
  - Tree is not an actual data structure, but a concept to determine what each thread does at each step
- For scan:
  - Traverse down from leaves to the root building partial sums at internal nodes in the tree
    - The root holds the sum of all leaves
  - Traverse back up the tree building the output from the partial sums

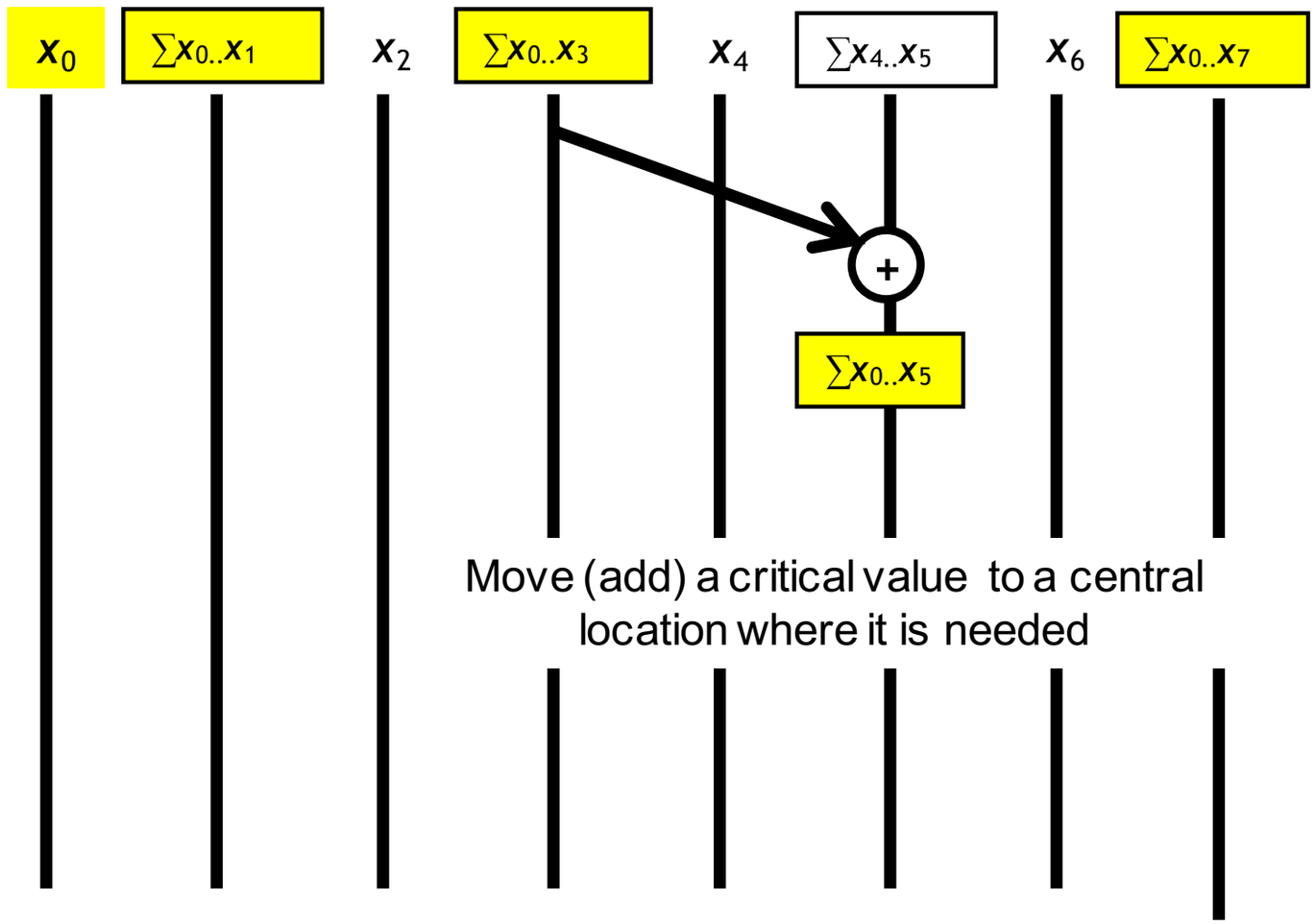
Parallel Scan - Reduction Phase

```
// XY[2*BLOCK_SIZE] is in shared memory

for (unsigned int stride = 1; stride <= BLOCK_SIZE;
        stride *= 2) {
    __syncthreads();
    int index = (threadIdx.x+1)*stride*2 - 1;
    if(index < 2*BLOCK_SIZE)
        XY[index] += XY[index-stride];
}
```

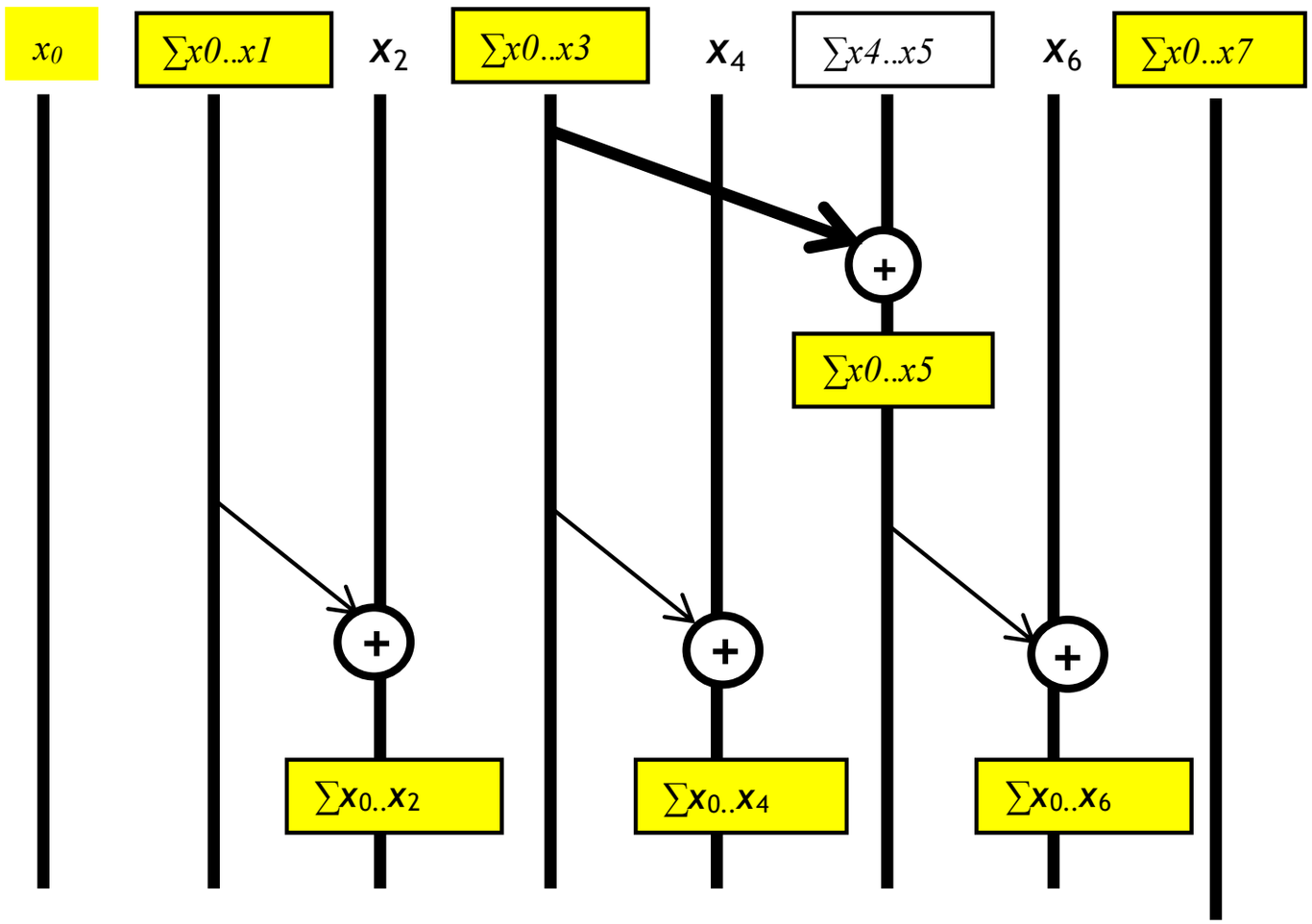
- threadIdx.x+1 = 1,2,3,4…
- stride = 1,
- index = 1,3,5,7, …



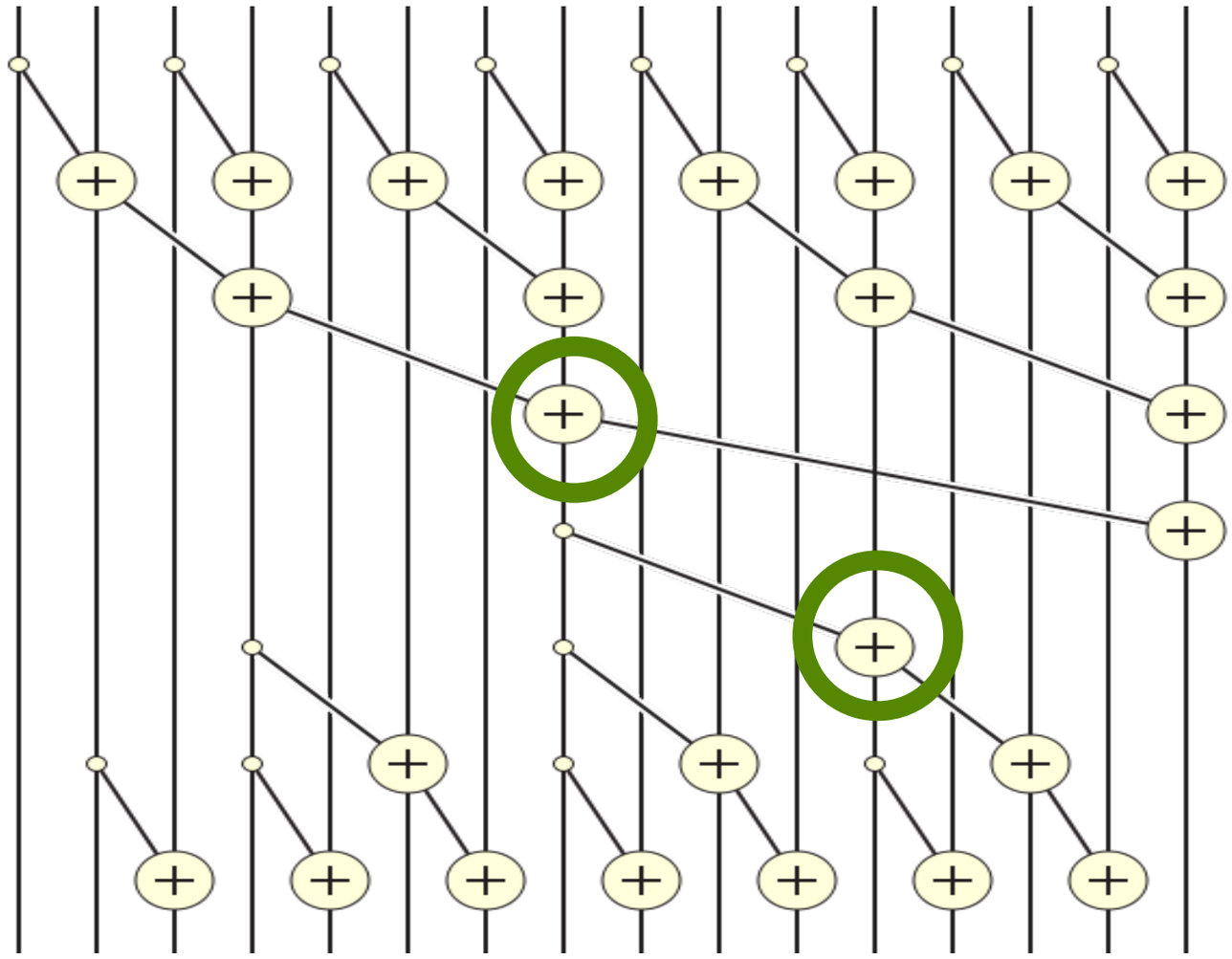
Parallel Scan - Post Reduction Reverse Phase



Parallel Scan - Post Reduction Reverse Phase



Parallel Scan - Post Reduction Reverse Phase



Putting together

Post Reduction Reverse Phase Kernel Code

```
for (unsigned int stride = BLOCK_SIZE/2; stride > 0; stride /= 2) {
     __syncthreads();
     int index = (threadIdx.x+1) * stride * 2 - 1;
     if(index+stride < 2*BLOCK_SIZE) {
        XY[index + stride] += XY[index];
     }
     }

     __syncthreads();
     if (i < InputSize)
         Y[i] = XY[threadIdx.x];
```

- First iteration for 16-element section
- threadIdx.x = 0
- stride = BLOCK_SIZE/2 = 8/2 = 4
- index = 8-1 = 7

Credits

- These slides report material from:
  - NVIDIA GPU Teaching Kit
  - Prof. Paul Richmond (Univ. Sheffield)

Books

- Programming Massively Parallel Processors: A Hands-on Approach, D. B. Kirk and W-M. W. Hwu, Morgan Kaufman - 2nd edition - Chapt. 5, 8 and 9

or

Programming Massively Parallel Processors: A Hands-on Approach, D. B. Kirk and W-M. W. Hwu, Morgan Kaufman - 3rd edition - Chapt. 4, 7, 8 and 9