CSE 599 I Accelerated Computing - Programming GPUS

Warp Shuffle and Warp Vote Instructions

Objective

- Introduce the warp shuffle and warp voting instructions available in CUDA
- See a few examples of warp-synchronous programming in practice

Review: Synchronize-and-Share

`__syncthreads_or(predicate):`

- Synchronizes all threads in a block and returns nonzero if any thread passes a nonzero predicate

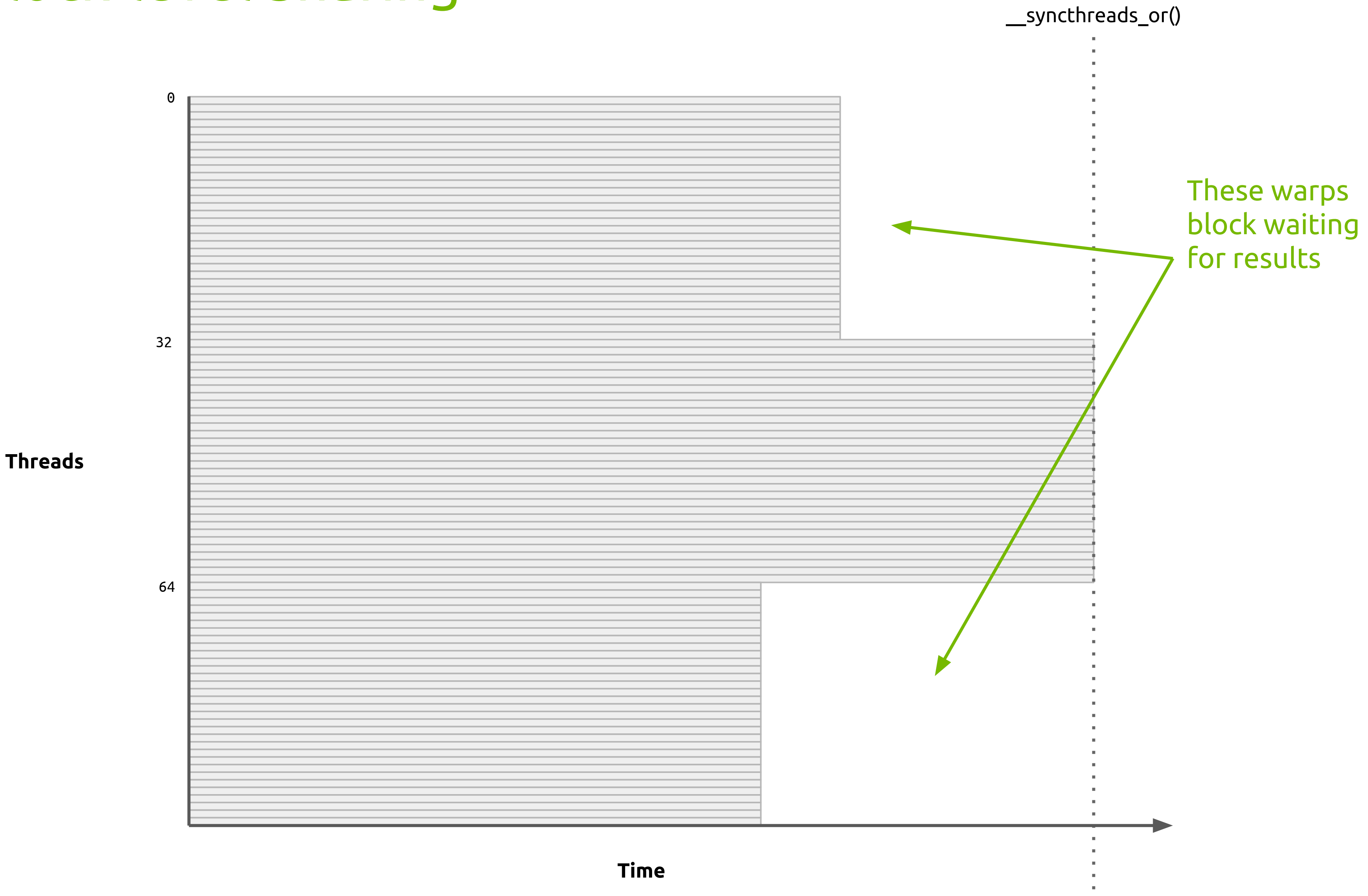
`__syncthreads_and(predicate):`

- Synchronizes all threads in a block and returns nonzero if all threads pass a nonzero predicate

These intrinsic instructions allow threads in a block to efficiently combine results without shared memory or atomic operations

However, it forces all warps in a block to synchronize first

Block-level Sharing



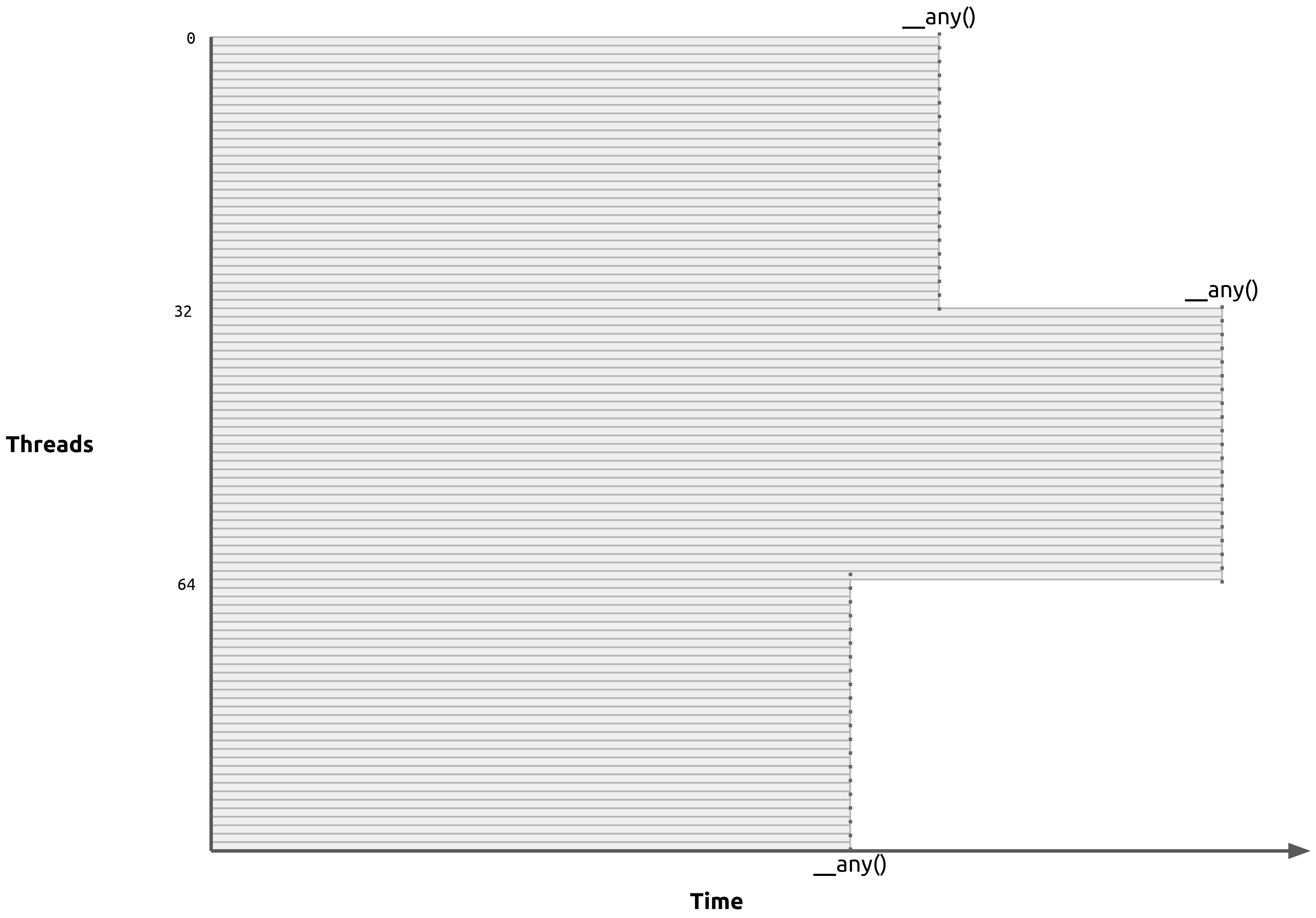
Warp-Synchronous Programming

Predicated `__syncthreads_*` instructions work based on the principle that sharing between threads in a block is possible at a synchronization point

However, threads in a warp are *always* synchronized

Warp voting instructions are like predicated `__syncthreads_*` instructions, except results are only aggregated across a warp, with no extra synchronization

Warp-level Sharing



Warp-vote Instructions

`__any(predicate):`

- Return nonzero if any thread in the warp passes a nonzero predicate

`__all(predicate):`

- Return nonzero if all threads in the warp pass a nonzero predicate

`__ballot(predicate):`

- Returns a 32-bit unsigned integer with each bit set according to the predicate passed by each thread in the warp

This allows threads to share bits. What if we want to share data (i.e. words)?

Review: Latency by Memory Type

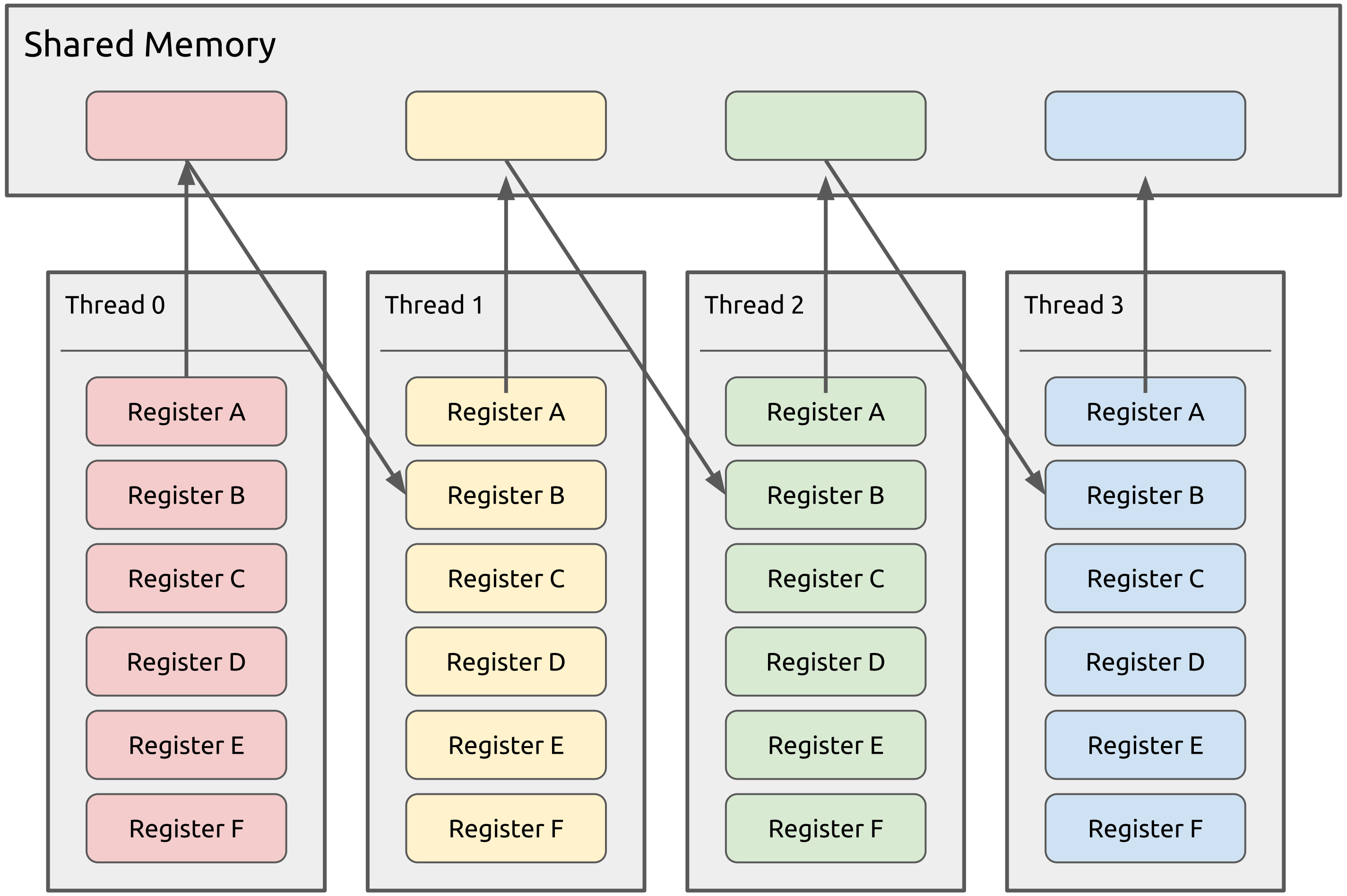
- Global memory: ~500 cycles

- Shared memory / constant memory (after caching): ~5 cycles

- Registers: ~1 cycle

Registers are the fastest memory type, but are specific to a single thread

Register Sharing via Shared Memory



The latency associated with sharing a register with a neighboring thread is twice the shared memory latency

Introducing Warp Shuffle Instructions

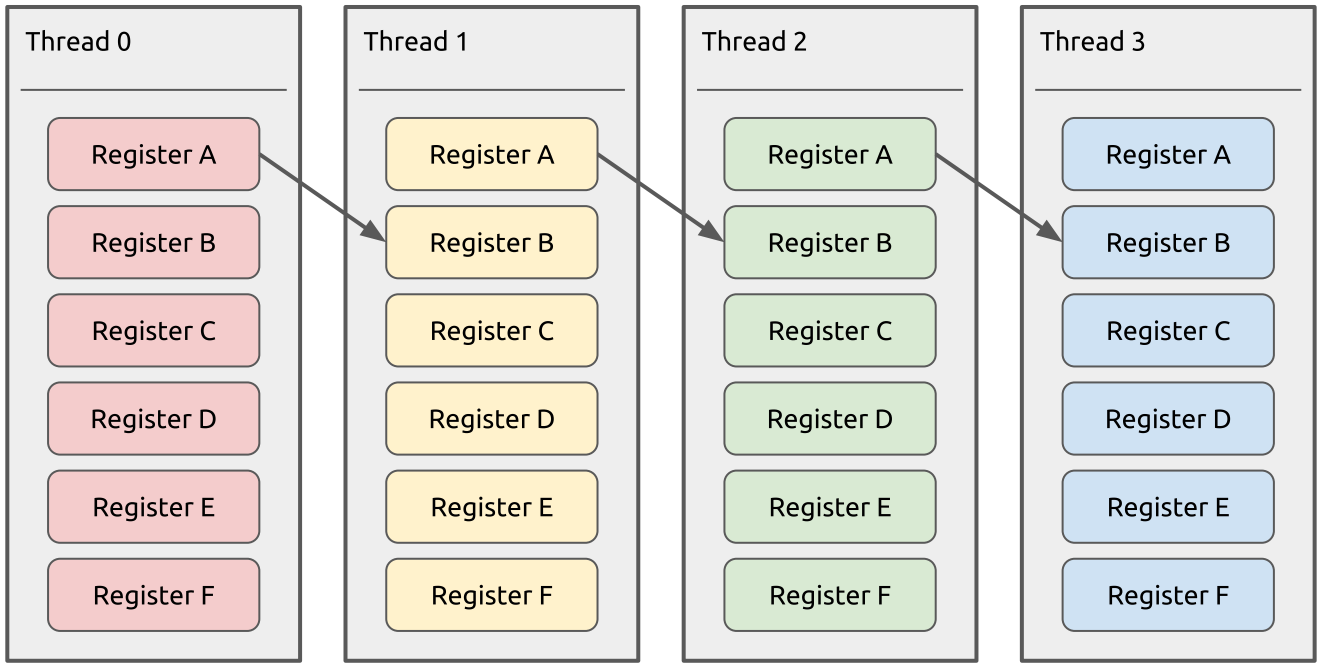
Warp shuffle instructions are intrinsic functions that allow threads to directly access another thread's registers

This results in extremely low-latency data sharing between threads with no extra memory required

Only threads within the same warp can share registers

Register Sharing via Warp Shuffle Instructions

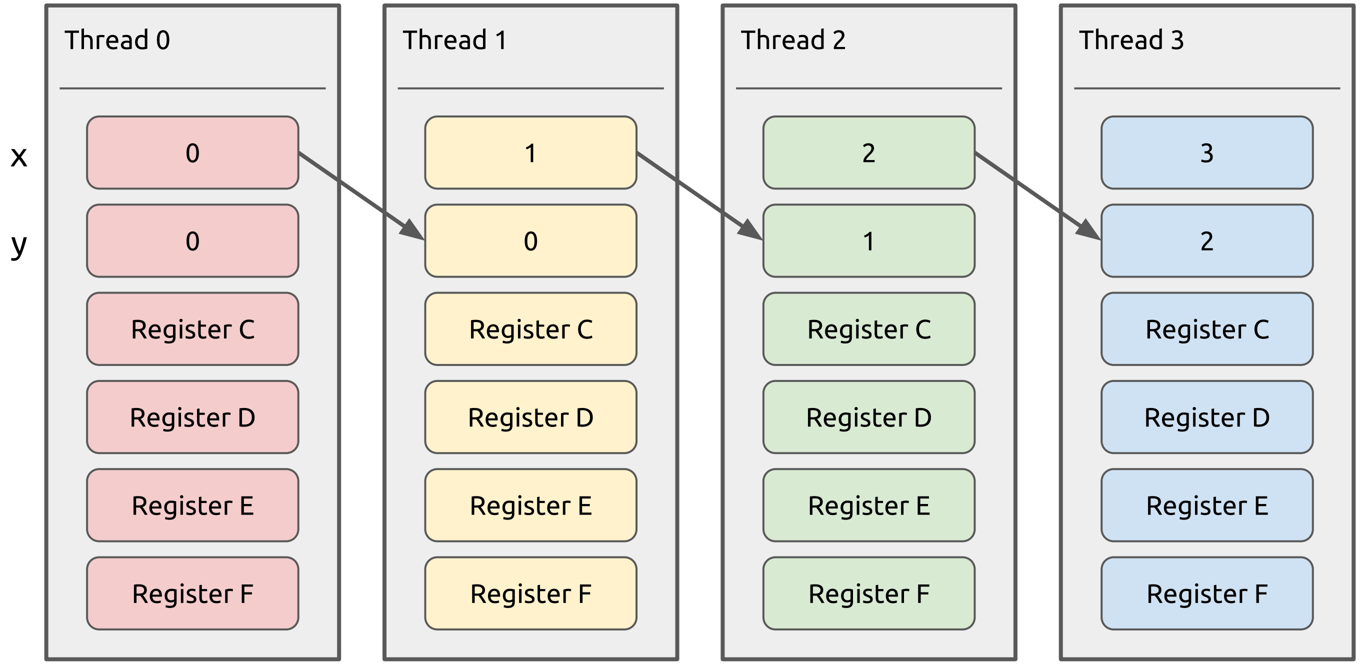
```
int x = threadIdx.x;        // stored in Register A
int y = __shfl_up(x, 1);    // stored in Register B
```



The latency associated with sharing a register with a neighboring thread is one instruction

Register Sharing via Warp Shuffle Instructions

```
int x = threadIdx.x;       // stored in Register A
int y = __shfl_up(x, 1);   // stored in Register B
```



The latency associated with sharing a register with a neighboring thread is one instruction

Introducing Lanes

A thread's **lane index** refers to its position relative to other threads in the warp

Lane and warp index for a 1D block are easily calculated using:

```
const int warpIndex = threadIdx.x / warpSize;

const int laneIndex = threadIdx.x % warpSize;
```

		laneIndex																															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
warpIndex	1	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
	2	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95

Future-Proofing Warp Size

All CUDA devices to date have had warps of size 32

This seems unlikely to change anytime soon, but technically, it could

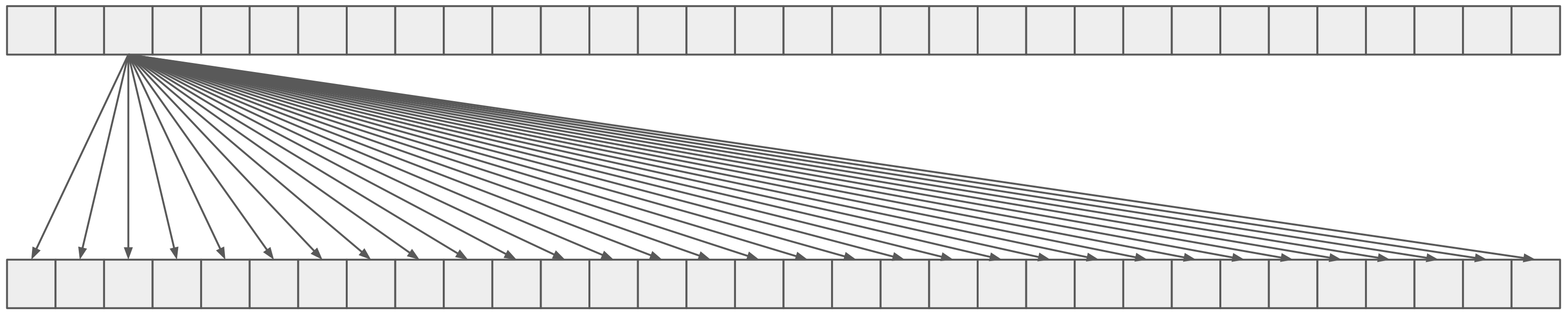
To be safe, the warp size of a CUDA device can be queried dynamically:

```
cudaDeviceProp prop;

cudaGetDeviceProperties(&prop, deviceNum);

printf("warp size is %d\n", prop.warpSize);
```

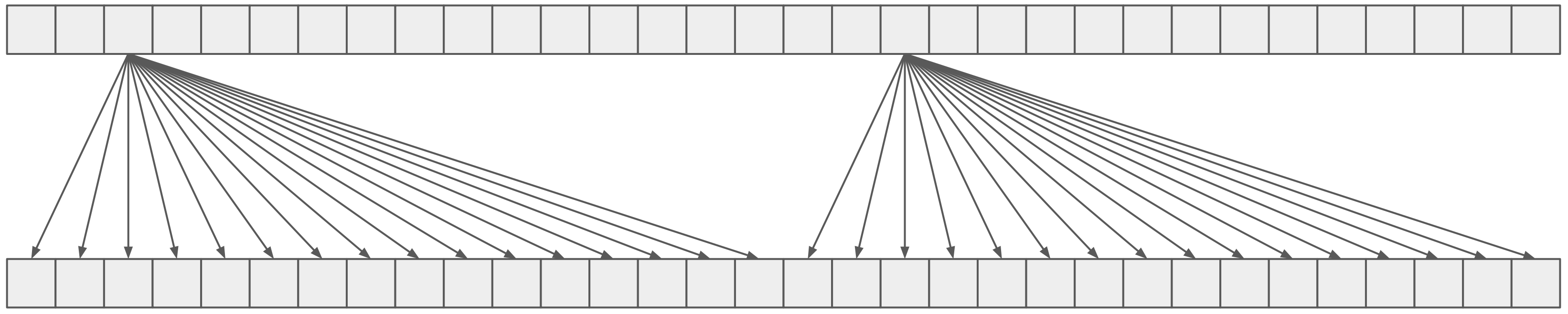
Warp Broadcast

```
int __shfl(int var, int srcLane, int width = warpSize);
```

```
int y = __shfl(x, 2);    // broadcast from threads 2, 34, 66, etc. to their respective warps
```



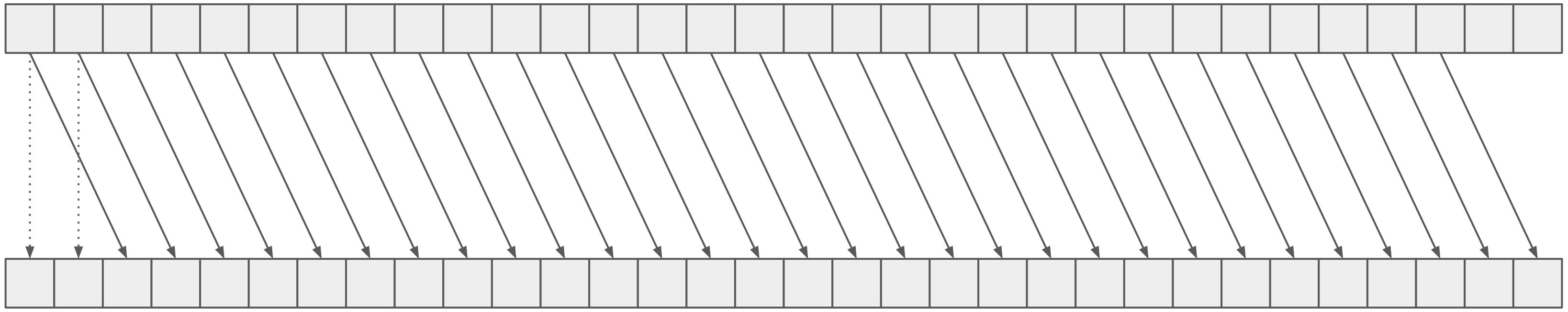
```
int y = __shfl(x, 2, 16); // broadcast from threads 2, 18, 34, etc. to their half-warps
```



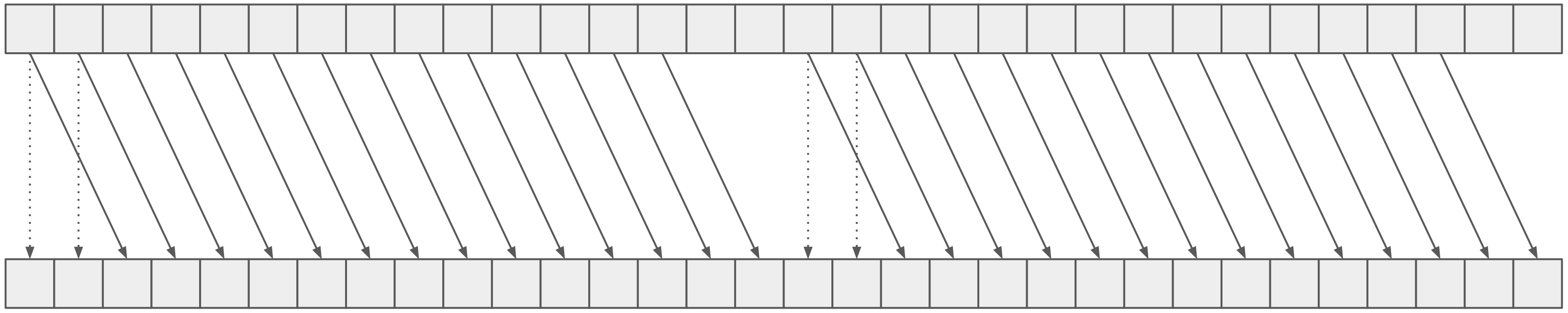
Warp Shuffle Up

```
int __shfl_up(int var, unsigned int delta, int width = warpSize);
```

```
int y = __shfl_up(x, 2);    // shift values two lanes to the right within a warp
```



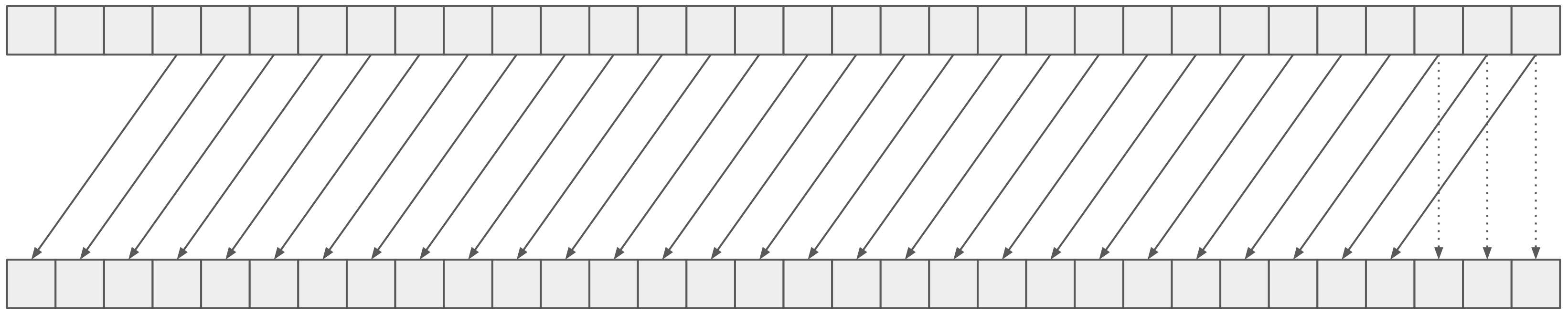
```
int y = __shfl_up(x, 2, 16);  // shift values two lanes to the right within a half-warp
```



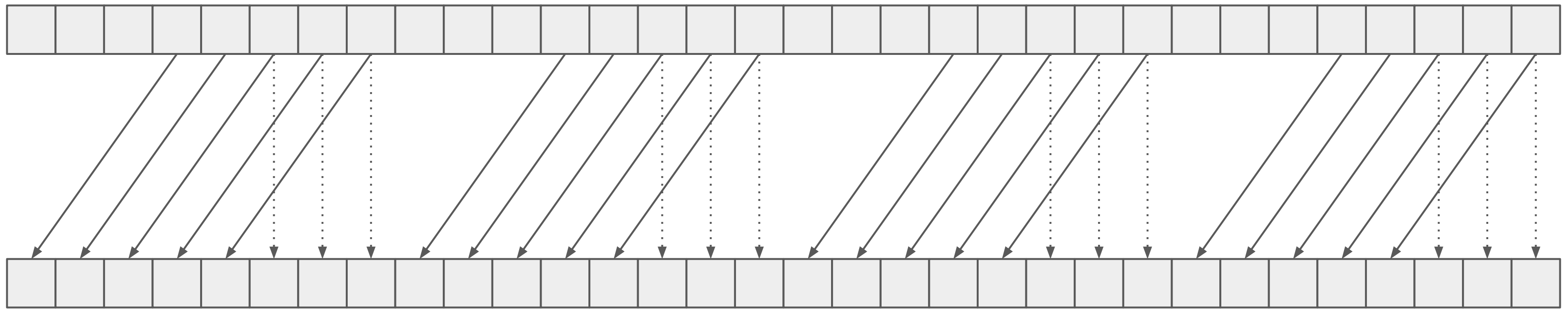
Warp Shuffle Down

```
int __shfl_down(int var, unsigned int delta, int width = warpSize);
```

```
int y = __shfl_down(x, 3);   // shift values two lanes to the right within a warp
```



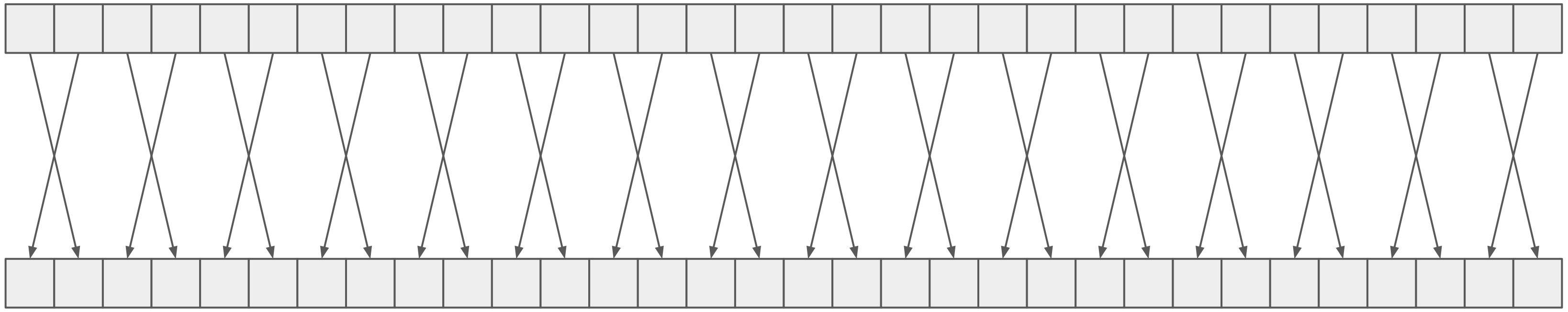
```
int y = __shfl_down(x, 3, 8);  // shift values two lanes to the right within a quarter-warp
```



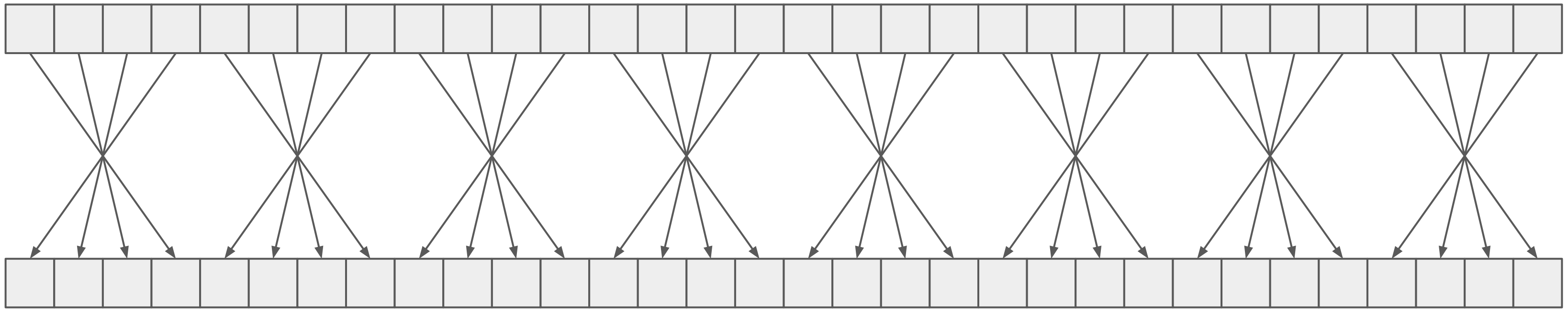
Masked Warp Shuffle (Butterfly Exchange)

```
int __shfl_xor(int var, int laneMask, int width = warpSize);
```

```
int y = __shfl_xor(x, 1);    // trade values with a neighbor
```



```
int y = __shfl_xor(x, 3);  // invert order of groups of four elements
```



Notes on Shuffle Instructions

- There is also a variant of each of the four shuffle instructions that takes a float / half2 as the first parameter and returns a float / half2
- The width parameter to all four shuffle instructions can be at most the size of a warp and must be a power of 2
- The srcLane parameter of __shfl may be larger than the warpSize
    - Data will come from srcLane % width in this case

Watch out for Control-flow Divergence!

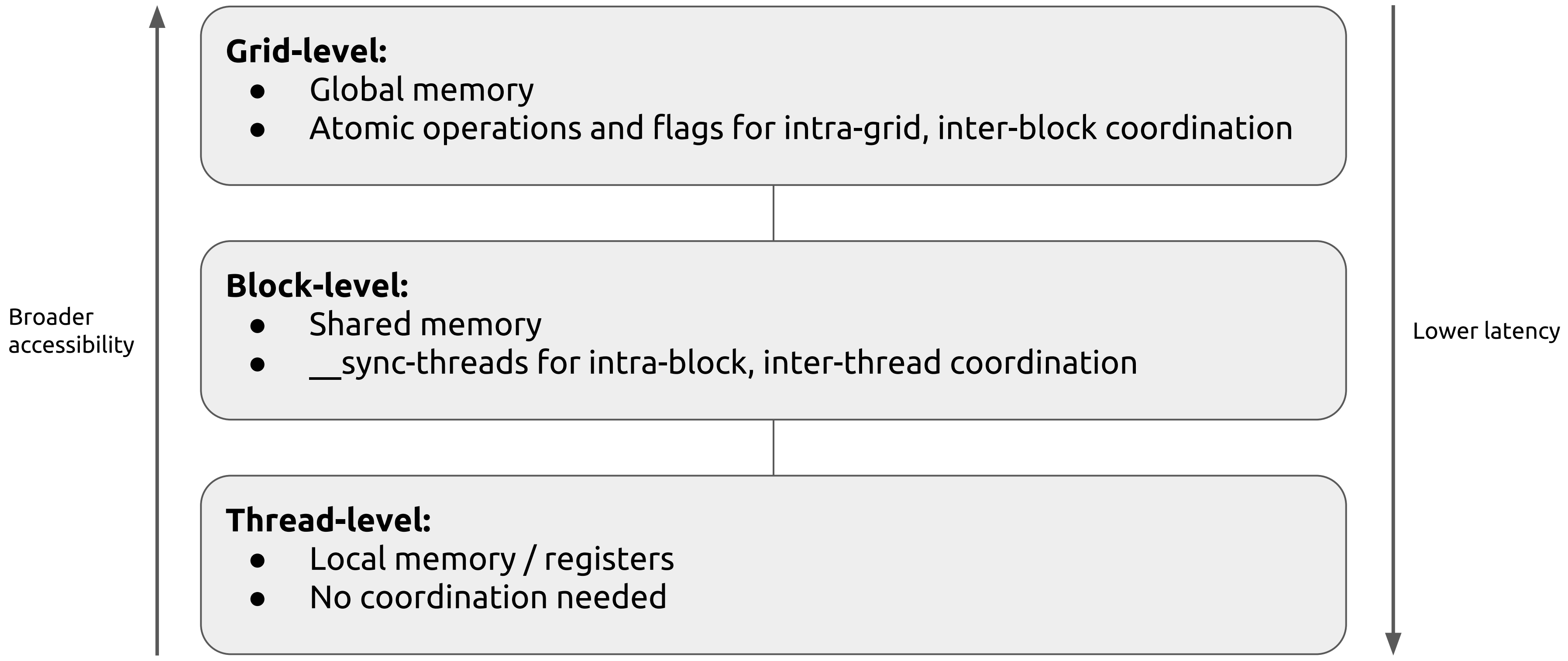
If warp-level instructions are nested in control-flow statements, it is possible that not all threads are active when the instruction is executed

```
if (threadIdx.x % 2) {
      int votes = __ballot(predicate); // Only every other thread participates in the vote
}
```

For voting instructions, the predicate of inactive threads is assumed to be 0

For shuffle instructions, values sourced from inactive threads will be **undefined**

Three-Level Processing Hierarchy

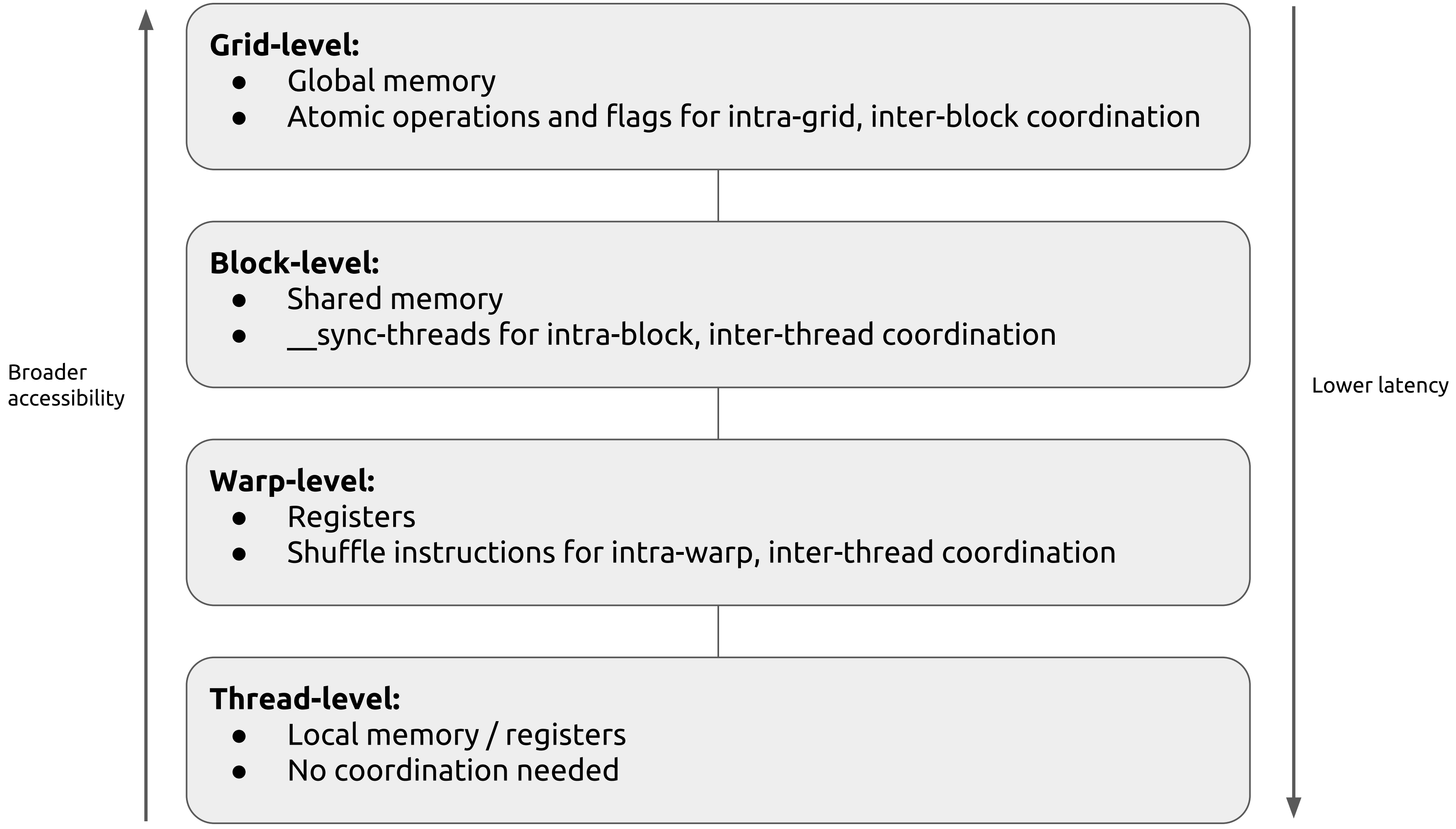


Grid-level:
- Global memory
- Atomic operations and flags for intra-grid, inter-block coordination

Block-level:
- Shared memory
- __sync-threads for intra-block, inter-thread coordination

Thread-level:
- Local memory / registers
- No coordination needed

Four-Level Processing Hierarchy



Grid-level:
- Global memory
- Atomic operations and flags for intra-grid, inter-block coordination

Block-level:
- Shared memory
- __sync-threads for intra-block, inter-thread coordination

Warp-level:
- Registers
- Shuffle instructions for intra-warp, inter-thread coordination

Thread-level:
- Local memory / registers
- No coordination needed

Reduction Revisited

```
__shared__ float partialSums[BLOCK_SIZE];

partialSums[threadIdx.x] = input[threadIdx.x + 2*BLOCK_SIZE*blockIdx.x];
partialSums[threadIdx.x + BLOCK_SIZE] = input[threadIdx.x + BLOCK_SIZE]

for (int stride = BLOCK_SIZE; stride > 0;  stride >>= 1) {

     __syncthreads();

     if (t < stride) {
          partialSums[t] += partialSums[t+stride];
     }

}
```

Once stride is equal to the warp size,
only one warp executes the sum

Reduction Revisited

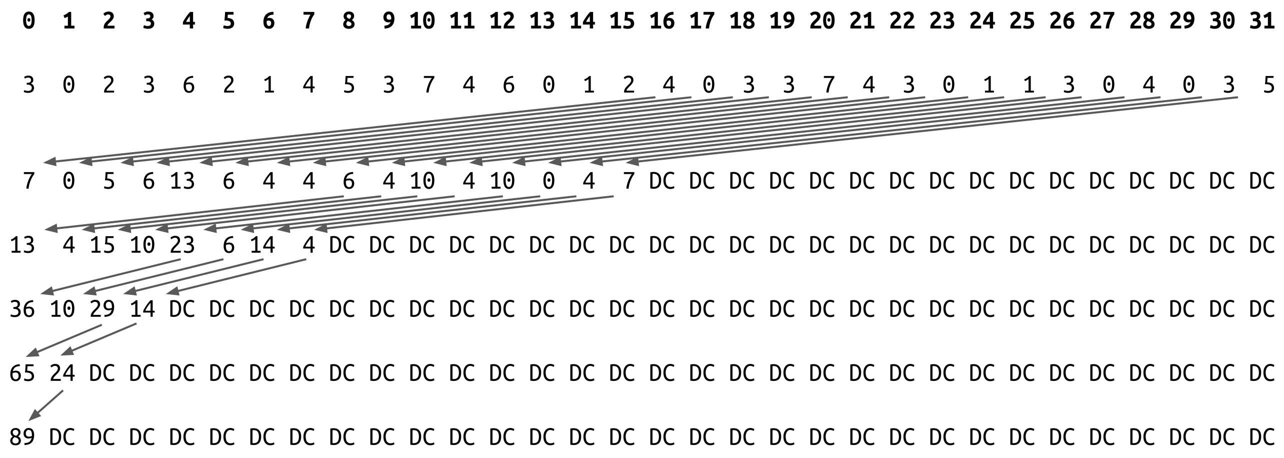
```
__shared__ volatile float partialSums[BLOCK_SIZE];

partialSums[threadIdx.x] = input[threadIdx.x + 2*BLOCK_SIZE*blockIdx.x];
partialSums[threadIdx.x + BLOCK_SIZE] = input[threadIdx.x + BLOCK_SIZE]

for (int stride = BLOCK_SIZE; stride > WARP_SIZE;  stride >>= 1) {

     __syncthreads();

     if (t < stride) {
          partialSums[t] += partialSums[t+stride];
     }

}

for (int stride = WARP_SIZE; stride > 0; stride >>= 1) {

     __threadfence_block();

     if (t < stride) {
          partialSums[t] += partialSums[t+stride];
     }

}
```

Only one warp, but we still need to make sure memory written by one thread is visible to all

Warp-level Reduction

```
__device__ __forceinline__ int warpReduceSum(int sum) {
        sum += __shfl_down(sum, 16);
        sum += __shfl_down(sum, 8);
        sum += __shfl_down(sum, 4);
        sum += __shfl_down(sum, 2);
        sum += __shfl_down(sum, 1);
        return sum;
}
```

Only the thread in lane 0 has a meaningful result!

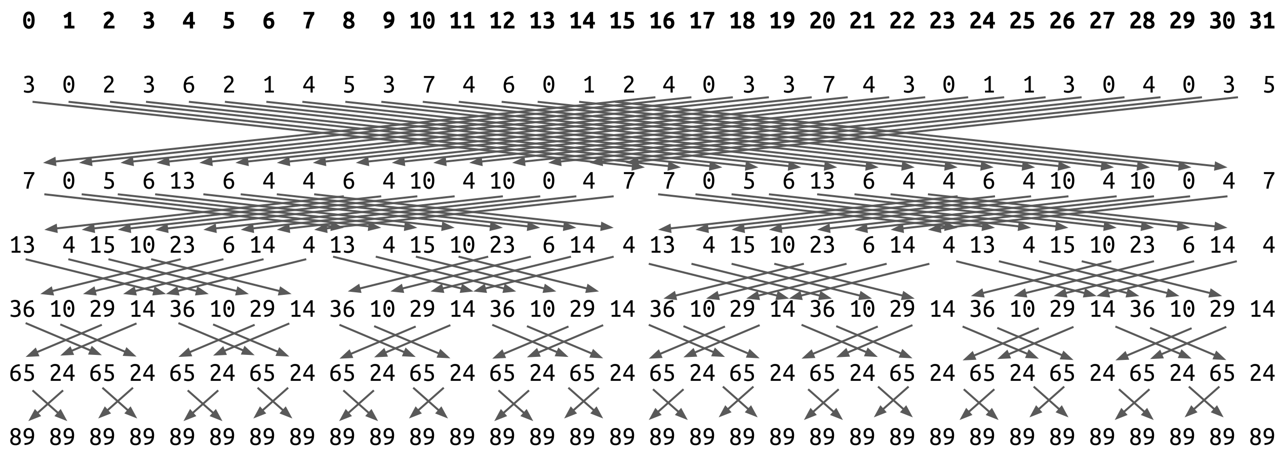


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
3 0 2 3 6 2 1 4 5 3 7 4 6 0 1 2 4 0 3 3 7 4 3 0 1 1 3 0 4 0 3 5
7 0 5 6 13 6 4 4 6 4 10 4 10 0 4 7 DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC
13 4 15 10 23 6 14 4 DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC
36 10 29 14 DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC
65 24 DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC
89 DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC DC

Warp-level Reduction

```
__device__ __forceinline__ int warpReduceSum(int sum) {
      sum += __shfl_xor(sum, 16);   // 0-16, 1-17, 2-18, etc.
      sum += __shfl_xor(sum, 8);// 0-8, 1-9, 2-10, etc.
      sum += __shfl_xor(sum, 4);// 0-4, 1-5, 2-6, etc.
      sum += __shfl_xor(sum, 2);// 0-2, 1-3, 4-6, 5-7, etc.
      sum += __shfl_xor(sum, 1);// 0-1, 2-3, 4-5, etc.
      return sum;
}
```

All threads return the sum



Block-Level Reduction

```
__device__ __forceinline__ int laneId() {
        return threadIdx.x % WARP_SIZE;
}

__device__ __forceinline__ int warpId() {
        return threadIdx.x / WARP_SIZE;
}

__inline__ __device__ int blockReduceSum(int val) {

        static __shared__ int warpLevelSums[WARP_SIZE]; // Shared mem for partial sums (one per warp in the block)
        const int laneId = laneId();
        const int warpId = warpId();

        val = warpReduceSum(val);     // Each warp performs warp-level reduction

        if (laneId == 0) warpLevelSums[warpId] = val; // Write reduced value to shared memory

        __syncthreads();              // Wait for all partial reductions

        // read from shared memory only if that warp existed
        val = (threadIdx.x < blockDim.x / WARP_SIZE) ? warpLevelSums[laneId] : 0;

        if (warpId == 0) val = warpReduceSum(val); // Final reduce using first warp

        return val; // NB: all warps enter, only first warp returns the block-level sum
}
```

Full Four-Level Reduction

```
__global__ void reductionKernel(const int * in, int * out, const int N) {
        int sum = 0;
        for (int i = blockIdx.x * blockDim.x + threadIdx.x;
              i < N;
              i += blockDim.x * gridDim.x) {
                sum += in[i];// thread-level reduction
        }
        sum = blockReduceSum(sum); // block- / warp-level reduction
        if (threadIdx.x == 0)
                atomicAdd(out, sum); // grid-level reduction
}
```

Knobs to tune:

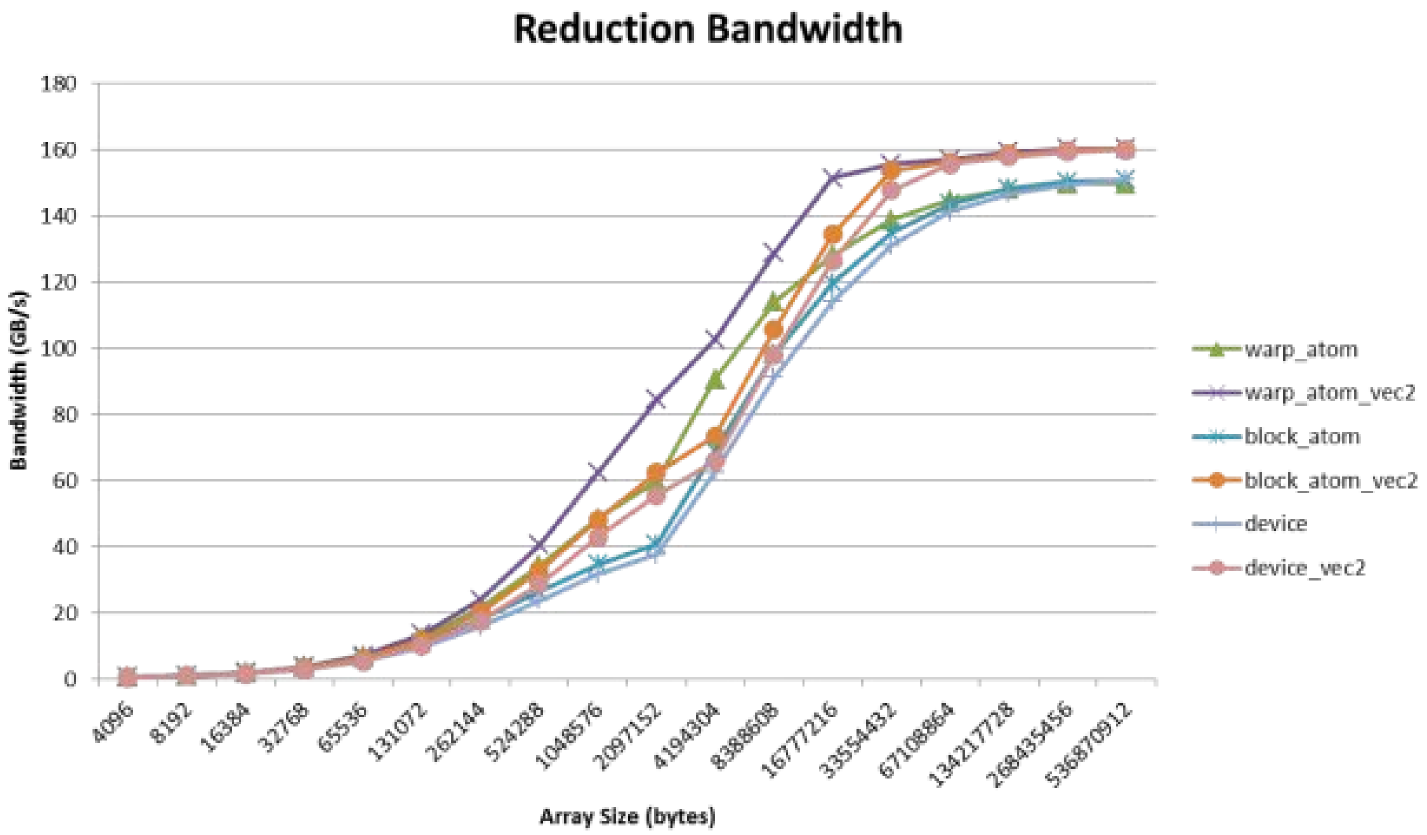
`blockDim.x * gridDim.x`	↑ Less thread-level processing	↓ More thread-level processing
`blockDim.x`	↑ More warp-level processing	↓ Less warp-level processing
`gridDim.x`	↑ More grid- and block-level processing	↓ Less grid- and block-level processing

Skipping the Block

Note that, depending on the problem and the hardware, it is sometimes faster to skip block-level processing and go straight from warp-level to grid-level

```
__global__ void reductionKernel(const int * in, int * out, const int N) {
       int sum = 0;
       for (int i = blockIdx.x * blockDim.x + threadIdx.x;
            i < N;
            i += blockDim.x * gridDim.x) {
              sum += in[i];// thread-level reduction
       }
       sum = warpReduceSum(sum); // block- / warp-level reduction
       if (threadIdx.x & (WARP_SIZE - 1) == 0)
              atomicAdd(out, sum); // grid-level reduction
}
```

Reduction Analysis



[https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/]

Other Warp-level Reductions

```
__device__ __forceinline__ int warpReduceMin(int minVal) {
      minVal = min(minVal,__shfl_xor(minVal, 16));
      minVal = min(minVal,__shfl_xor(minVal,  8));
      minVal = min(minVal,__shfl_xor(minVal,  4));
      minVal = min(minVal,__shfl_xor(minVal,  2));
      minVal = min(minVal,__shfl_xor(minVal,  1));
      return min;
}

__device__ __forceinline__ int warpReduceProduct(int prod) {
      prod *= __shfl_xor(prod, 16));
      prod *= __shfl_xor(prod,  8));
      prod *= __shfl_xor(prod,  4));
      prod *= __shfl_xor(prod,  2));
      prod *= __shfl_xor(prod,  1));
      return min;
}
```

Warp Scan

Shuffle instructions can also be used to implement a scan

```
__device__ __forceinline__ float warpScan(float value) {
#pragma unroll
    for (int i = 1; i < WARP_SIZE; i <<= 1) {
        float otherValue = __shfl_up(value, i);
        if (threadIdx.x >= i)
            value += otherValue;
    }
    return value;
}
```

Using PTX Instructions In CUDA C

You can write inline intermediate assembly instructions in CUDA C code!

This can be useful for:

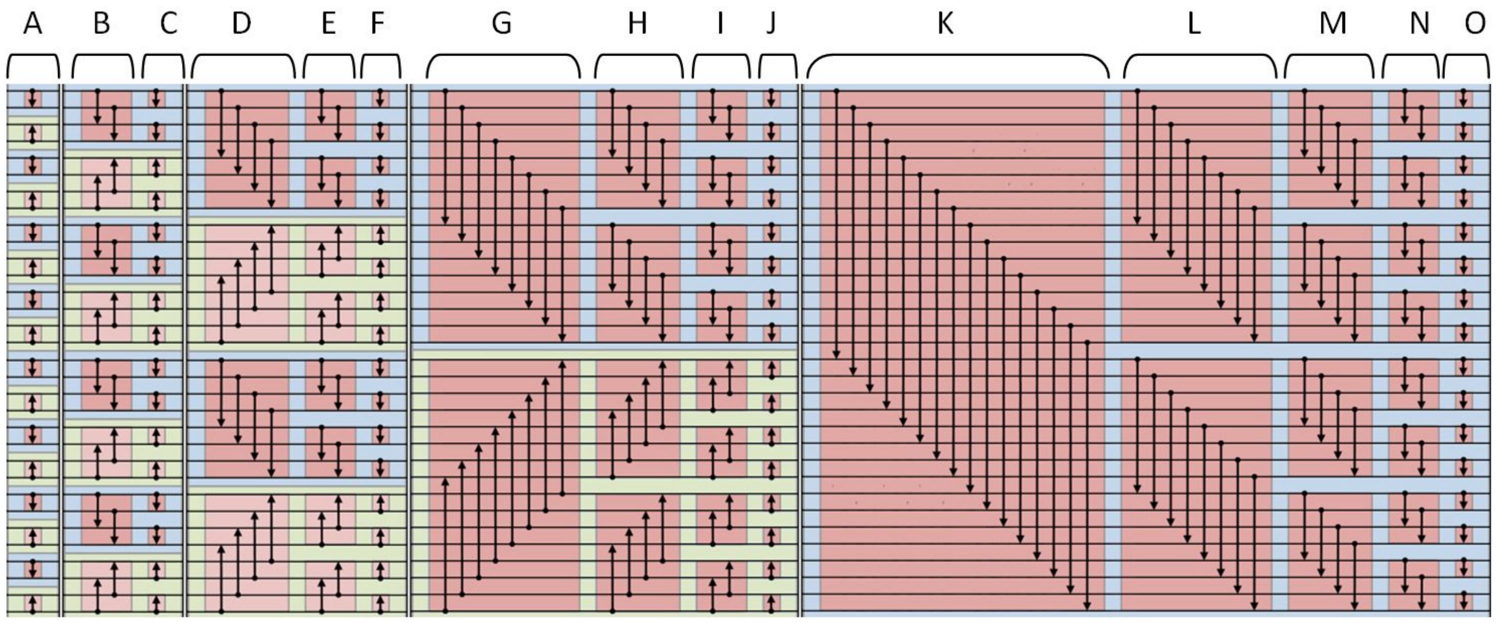
- Manually-optimized assembly (for the pros)
- Access to special registers not exposed in CUDA C
- Access to special instructions not exposed in CUDA C

Example:

```
__device__ __forceinline__ unsigned int laneId() {
        unsigned int laneId;
        asm volatile("mov.u32 %0, %%laneid;" : "=r"(laneId));
        return laneId;
}
```

Warp-level Bitonic Sort

```
__device__ __forceinline__ float warpSort(float & threadValue, const unsigned int laneId) {
      threadValue = comparator(x,  1, bfe(laneId, 1) ^ bfe(laneId, 0));          // A, sorted sequences of length 2
      threadValue = comparator(x,  2, bfe(laneId, 2) ^ bfe(laneId, 1));          // B
      threadValue = comparator(x,  1, bfe(laneId, 2) ^ bfe(laneId, 0));          // C, sorted sequences of length 4
      threadValue = comparator(x,  4, bfe(laneId, 3) ^ bfe(laneId, 2));          // D
      threadValue = comparator(x,  2, bfe(laneId, 3) ^ bfe(laneId, 1));          // E
      threadValue = comparator(x,  1, bfe(laneId, 3) ^ bfe(laneId, 0));          // F, sorted sequences of length 8
      threadValue = comparator(x,  8, bfe(laneId, 4) ^ bfe(laneId, 3));          // G
      threadValue = comparator(x,  4, bfe(laneId, 4) ^ bfe(laneId, 2));          // H
      threadValue = comparator(x,  2, bfe(laneId, 4) ^ bfe(laneId, 1));          // I
      threadValue = comparator(x,  1, bfe(laneId, 4) ^ bfe(laneId, 0));          // J, sorted sequences of length 16
      threadValue = comparator(x, 16, bfe(laneId, 4));                     // K
      threadValue = comparator(x,  8, bfe(laneId, 3));                     // L
      threadValue = comparator(x,  4, bfe(laneId, 2));                     // M
      threadValue = comparator(x,  2, bfe(laneId, 1));                     // N
      threadValue = comparator(x,  1, bfe(laneId, 0));                     // O, sorted sequences of length 32
}
```



Warp-level Bitonic Sort

```
// bit field extract
__device__ __forceinline__ bfe(unsigned int source, unsigned int bitIndex) {
       unsigned int bit;
       asm volatile("bfe.u32 %0, %1, %2, %3;" : "=r"(bit) : "r"((unsigned int) source), "r"(bitIndex), "r"(1));
       return bit;
}

__device__ __forceinline__ comparator(const float value, const int stride, const int direction) {
       const float otherValue = __shfl_xor(value, stride);
       return value < otherValue == direction ? y : x;
}

__device__ __forceinline__ float warpSort(float & threadValue, const unsigned int laneId) {
       threadValue = comparator(x,  1, bfe(laneId, 1) ^ bfe(laneId, 0));            // A, sorted sequences of length 2
       threadValue = comparator(x,  2, bfe(laneId, 2) ^ bfe(laneId, 1));            // B
       threadValue = comparator(x,  1, bfe(laneId, 2) ^ bfe(laneId, 0));            // C, sorted sequences of length 4
       threadValue = comparator(x,  4, bfe(laneId, 3) ^ bfe(laneId, 2));            // D
       threadValue = comparator(x,  2, bfe(laneId, 3) ^ bfe(laneId, 1));            // E
       threadValue = comparator(x,  1, bfe(laneId, 3) ^ bfe(laneId, 0));            // F, sorted sequences of length 8
       threadValue = comparator(x,  8, bfe(laneId, 4) ^ bfe(laneId, 3));            // G
       threadValue = comparator(x,  4, bfe(laneId, 4) ^ bfe(laneId, 2));            // H
       threadValue = comparator(x,  2, bfe(laneId, 4) ^ bfe(laneId, 1));            // I
       threadValue = comparator(x,  1, bfe(laneId, 4) ^ bfe(laneId, 0));            // J, sorted sequences of length 16
       threadValue = comparator(x, 16, bfe(laneId, 4));                     // K
       threadValue = comparator(x,  8, bfe(laneId, 3));                     // L
       threadValue = comparator(x,  4, bfe(laneId, 2));                     // M
       threadValue = comparator(x,  2, bfe(laneId, 1));                     // N
       threadValue = comparator(x,  1, bfe(laneId, 0));                     // O, sorted sequences of length 32
}
```

Voting Example: Stream Compaction

Problem: Copy all input elements for which a predicate returns nonzero into contiguous locations in an output array

Stream Compaction: Grid-Level Only

```
__global__ void compact(int * dst, int * nValid, const int * src, const int N) {
     int i = threadIdx.x + blockIdx.x * blockDim.x;
     if (i < N && src[i] > 0) {
          dst[atomicAdd(nValid, 1)] = src[i];
     }
}
```

- Very simple code
- Very slow if there are many valid elements

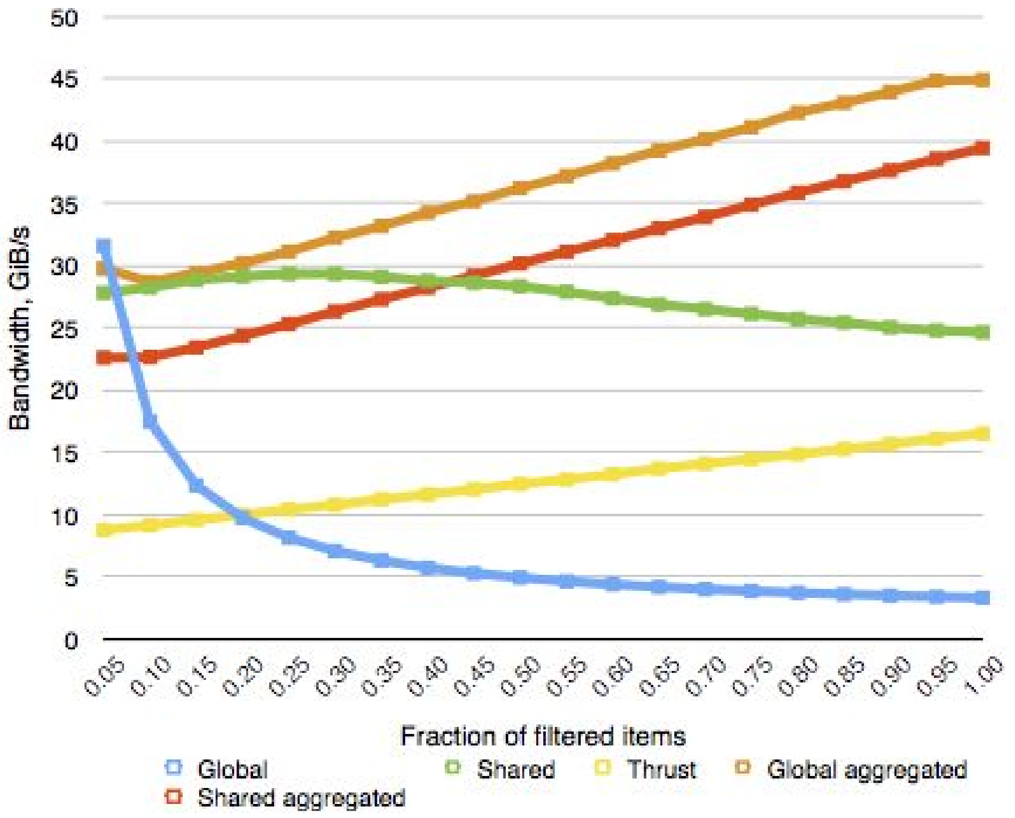
Stream Compaction: Grid- and Block-Level

```
__global__ void compactShared(int * dst, int * nValid, const int * src, const int N) {
        __shared__ int nLocalValid;
        int i = blockIdx.x * (NPER_THREAD * BS) + threadIdx.x;

        for (int iter = 0; iter < NPER_THREAD; iter++) {
                if (threadIdx.x == 0)           // zero the counter
                        nLocalValid = 0;
                __syncthreads();

                // get the value, evaluate the predicate, and increment the counter if needed
                int d, pos;
                if (i < N) {
                        d = src[i];
                        if (d > 0)  pos = atomicAdd(&nLocalValid, 1);
                }
                __syncthreads();

                if (threadIdx.x == 0)   // leader increments the global counter
                        nLocalValid = atomicAdd(nValid, nLocalValid);
                __syncthreads();

                if (i < N && d > 0) {   // threads with true predicates write their elements
                        pos += nLocalValid; // increment local pos by global counter
                        dst[pos] = d;
                }
                __syncthreads();

                i += BS;
        }
}
```

Warp-Aggregated Atomics

```
__device__ inline int warpBroadcast(const int value, const int source) {
      return __shfl(value, source);
}

__device__ inline int laneId() {
      return threadIdx.x % WARP_SIZE;
}

// warp-aggregated atomic increment
__device__ int warpAggregatedAtomicAdd(int * numValid) {
      const int activeThreadMask = __ballot(1);
      // select the leader
      const int leader = __ffs(activeThreadMask) - 1;
      // leader does the update
      int startingIndex;
      if (laneId() == leader)
            startingIndex = atomicAdd(numValid, __popc(activeThreadMask));
      // broadcast result
      startingIndex = warpBroadcast(startingIndex, leader);
      // each thread computes its own value
      return startingIndex + __popc(activeThreadMask & ((1 << laneId()) - 1));
}

__global__ void compact(int * dst, int * nValid, const int * src, const int N) {
      int i = threadIdx.x + blockIdx.x * blockDim.x;
      if (i < N && src[i] > 0) {
            dst[warpAggregatedAtomicAdd(nValid)] = src[i];
      }
}
```

Warp-Aggregated Atomic Analysis



[https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/]

Conclusion / Takeaways

- Warp shuffle and warp voting instructions allow for very efficient coordination between threads in the same warp
- Warp-level processing can be used alongside or in place of block-level processing to increase efficiency of CUDA code
- There are “special” registers and instructions hiding in the PTX documentation!

Further Reading

“Optimized Filtering with Warp-Aggregated Atomics”

https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/

“Faster Parallel Reductions on Kepler”

https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/

“Shuffle: Tips and Tricks”

http://on-demand.gputechconf.com/gtc/2013/presentations/S3174-Kepler-Shuffle-Tips-Tricks.pdf

Sources

Cheng, John, Max Grossman, and Ty McKercher. Professional Cuda C Programming. John Wiley & Sons, 2014.