



GPU Programming with CUDA

Optimization

Prof. Dr. Matthias S. Müller

Dr. Christian Terboven

Dr. Sandra Wienke

Julian Miller

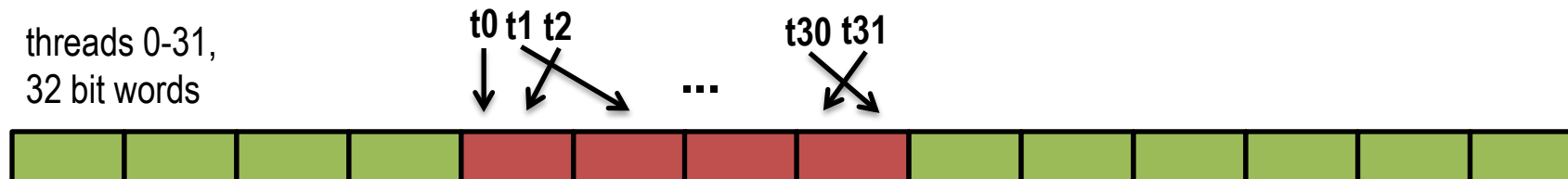
What is This Chapter About?

- How to optimize an application with CUDA
 - Data access patterns
 - Memory coalescing
 - Branching
 - Synchronization
 - Heterogeneous computing

Data Access Patterns

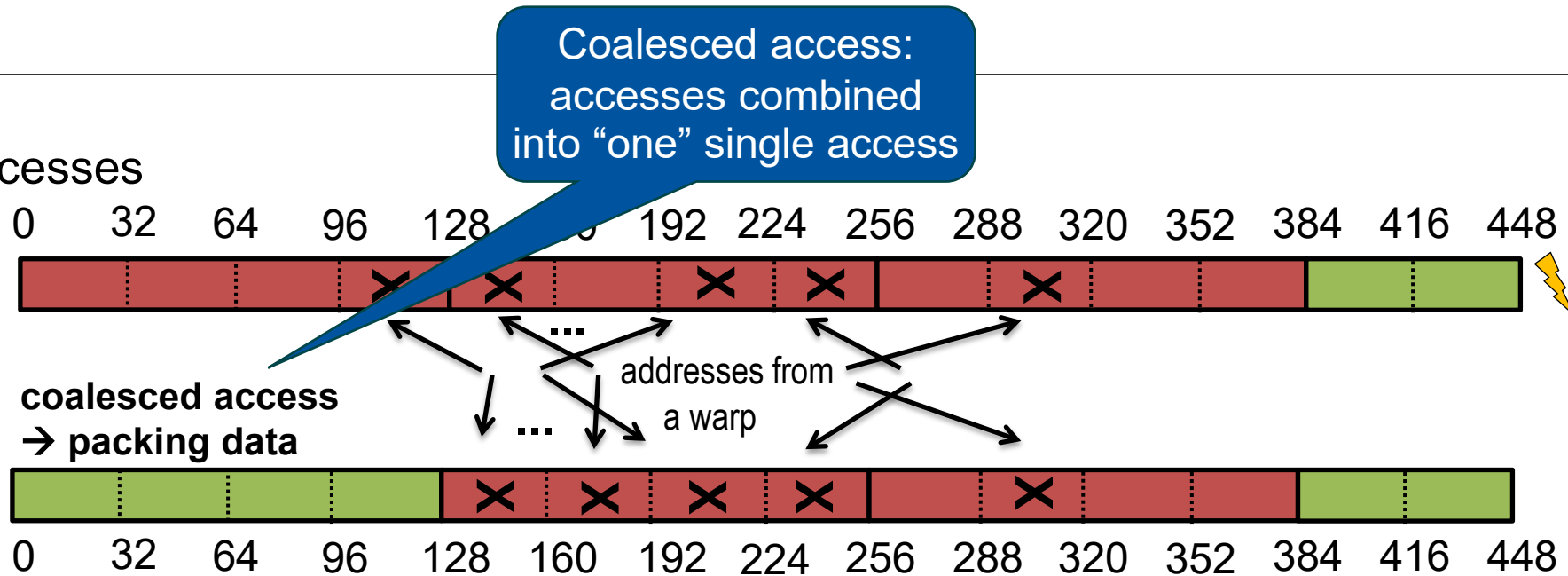
Coalescing

- Coalescing: concept to maximize global memory throughput
- Global memory access per warp
 - Segments of 128-byte (“cache lines”) can be accessed by a warp in a single instruction
 - Determine which segments are needed
 - Request the needed segments
- Degree of coalescing = $\frac{\text{\#bytes requested}}{\text{\#bytes read}}$ describes overhead of reading more data than requested

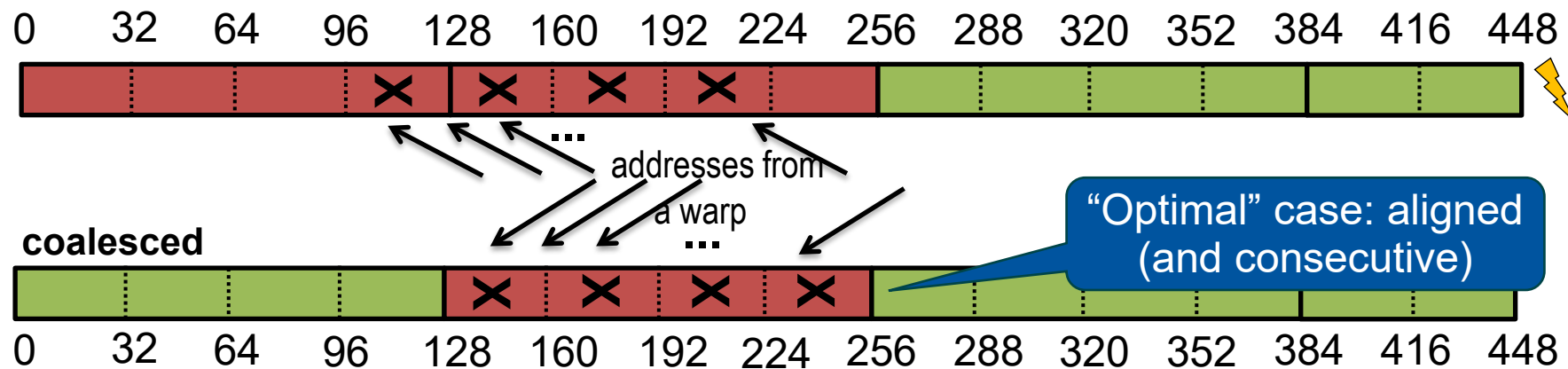


Coalescing

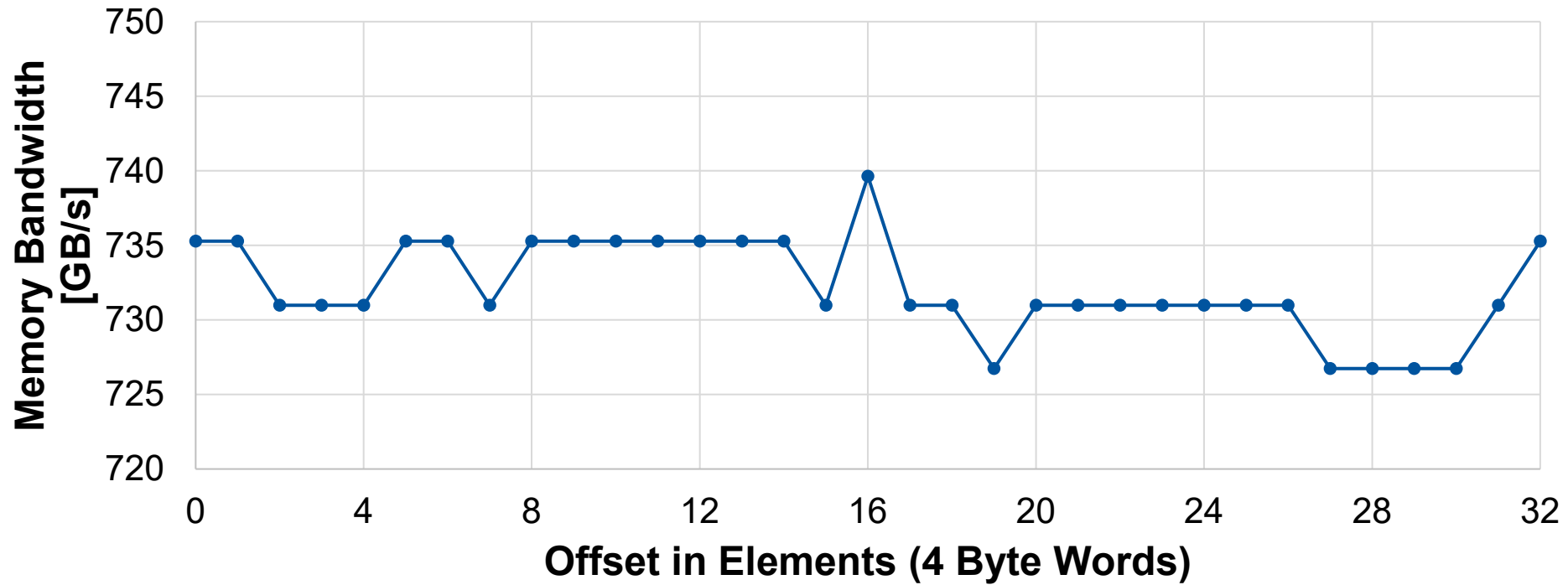
- Range of accesses



- Address alignment



Coalescing - Impact of Address Alignment



- Benchmark to investigate impact of address alignment
 - Copy of 64 MB of integers
 - NVIDIA V100, 256 threads/block
 - Misaligned accesses can drop memory throughput

Sources: <https://github.com/NVIDIA-developer-blog/code-samples/blob/master/series/cuda-cpp/coalescing-global>

Example 3D Points

- Array of Structures (AoS)

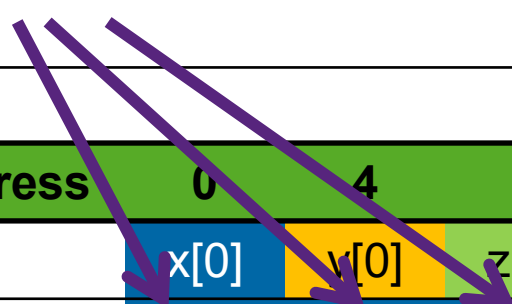
```
struct Pt {  
    float x;  
    float y;  
    float z;  
}  
Struct Pt myPts[N];  
  
// Offload to GPU  
for (int i = 0; i < N; i++) {  
    myPts[i].x = i; // Do something with x  
}
```



Example 3D Points

- Structure of Arrays (SoA)

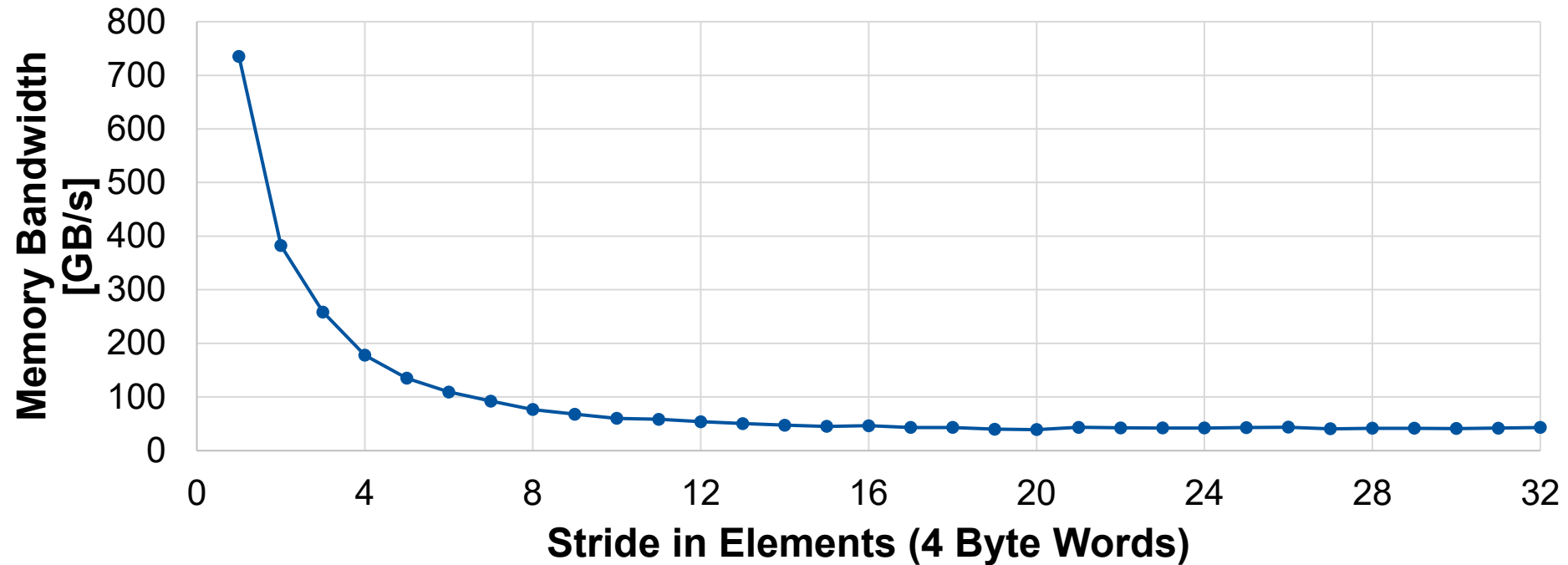
```
struct Pt {  
    float x[N];  
    float y[N];  
    float z[N];  
}  
Struct Pt myPts;  
  
// Offload to GPU  
for (int i = 0; i < N; i++) {  
    myPts.x[i] = i; // Do something with x  
}
```



Address	0	4	8	12	16	20	24	28	32
AoS	x[0]	y[0]	z[0]	x[1]	y[1]	z[1]	x[2]	y[2]	z[2]
SoA	x[0]	x[1]	x[2]	y[0]	y[1]	y[2]	z[0]	z[1]	z[2]

Recommended
Paper: Mei, G. and
Tian, H., 2016.
Impact of data
layouts on the
efficiency of GPU-
accelerated IDW
interpolation. *Sprin
gerPlus*, 5(1),
p.104.

Coalescing - Impact of Strided Memory Access



- Benchmark to investigate impact of address alignment
 - Copy of 64 MB of integers
 - NVIDIA V100, 256 threads/block
- Strided accesses can drop memory throughput

Sources: <https://github.com/NVIDIA-developer-blog/code-samples/blob/master/series/cuda-cpp/coalescing-global>

Branching

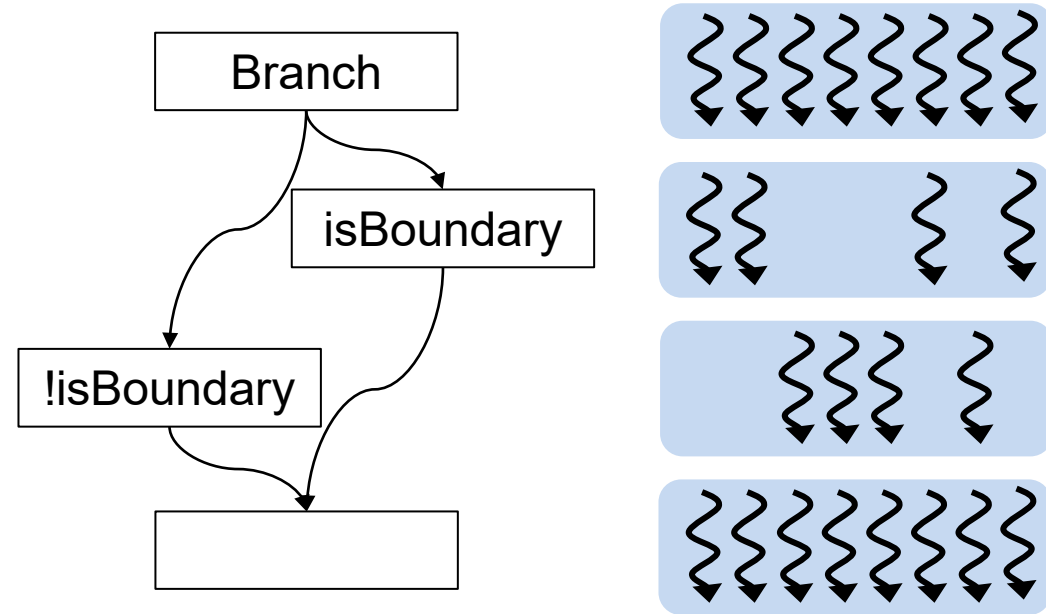
Example Finite Difference Method

- What happens if threads within a warp execute different instructions?

```
__global__ void finiteDifference(int n, double *u, double *u_new) {  
    int tid = blockDim.x * blockIdx.x + threadIdx.x;  
    // Calculate the central difference at u[i] if u[i] is  
    // not at the boundary of the domain  
    if (isBoundary(tid))  
        u_new[tid] = 0;  
    else  
        u_new[tid] = (u[tid + 1] - u[tid - 1]) * 0.5;  
}
```

Branch Divergence

- Remember threads are organized in warps which share a program counter
 - Hardware serializes the different execution paths
 - Up to 32x performance loss
- Branch Efficiency: The ratio of executed uniform flow control decisions over all executed conditionals



Dealing with Branch Divergence

- Avoid different execution paths within the same warp
 - Typically, avoid flow control instructions wherever possible
 - Compiler tries to mitigate the effect of branch divergence by e.g. predication, but performance implications (instruction throughput) can be significant
- Common case: function depends on thread ID
 - Align controlling condition on warp granularity
 - e.g. `if (threadIdx.x / WARP_SIZE) { }`
- Example Finite Difference Method
 - Two separate kernels for the boundaries and the inner points

Synchronization

Synchronization

- By default: all calls are placed in the default stream
 - Stream: queue of kernels that are executed sequentially
 - However: calls return on the CPU once placed in the stream
- GPUs only allow for synchronization within a streaming multiprocessor
 - Synchronization or memory fences across SMs not supported due to limited control logic
 - Barriers, critical regions, locks, atomics only apply to the threads within a team
 - No cache coherence between L1 caches

Example DAXPY: Kernel Timing

```
int main(int argc, const char* argv[]) {  
    ...  
    double runtime = GetRealTime();  
    daxpyGPU<<<(n+255)/256, 256>>>(n, a, d_x, d_y); // kernel  
    runtime = GetRealTime() - runtime;  
    ...  
    printf("Time Elapsed: %f s\n", runtime);  
    ...  
}
```

Output:
\$ nvcc daxpy.cu
\$ a.out
Max error: 0.00000
Time elapsed: 70us

- Kernel launches are asynchronous
 - Control returns to CPU immediately

Example DAXPY: Kernel Timing

```
int main(int argc, const char* argv[]) {
    ...
    double runtime = GetRealTime();
    daxpyGPU<<<(n+255)/256, 256>>>(n, a, d_x, d_y); // kernel
    cudaDeviceSynchronize();
    runtime = GetRealTime() - runtime;
    ...
    printf("Time Elapsed: %f s\n", runtime);
    ...
}
```

Output:
\$ nvcc daxpy.cu
\$ a.out
Max error: 0.00000
Time elapsed: 2.91ms

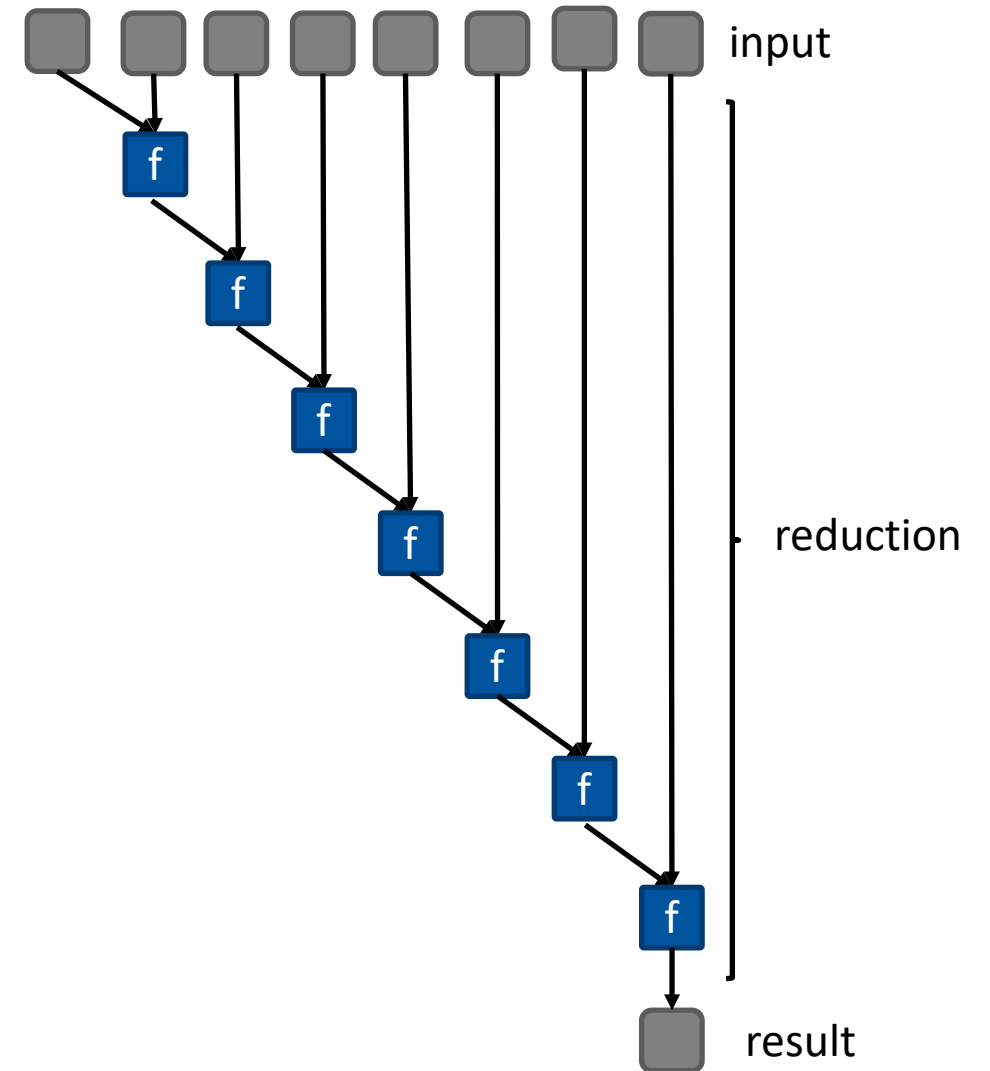
- Kernel launches are asynchronous
 - Control returns to CPU immediately
- `cudaDeviceSynchronize()`
 - Blocks until all outstanding CUDA calls are complete

Reduction

```
void sum(float * input,
        float output,
        int n) {
    for (int i = 0; i < n; ++i) {
        output += input[i];
    }
}
```

CPU: 2.72s for 2^{30} elements
GPU: 53.48s for 2^{30} elements

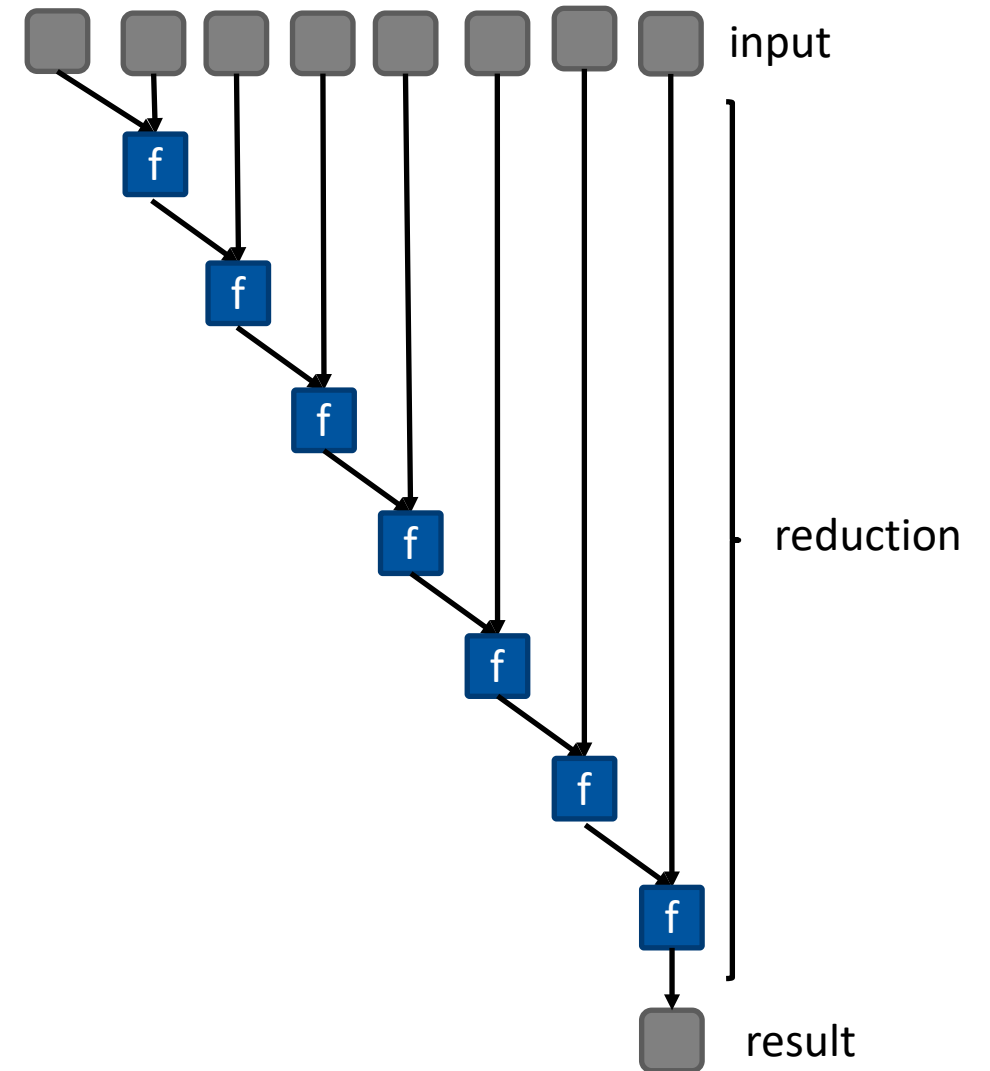
- Can this be parallelized?



Naïve Reduction on GPU

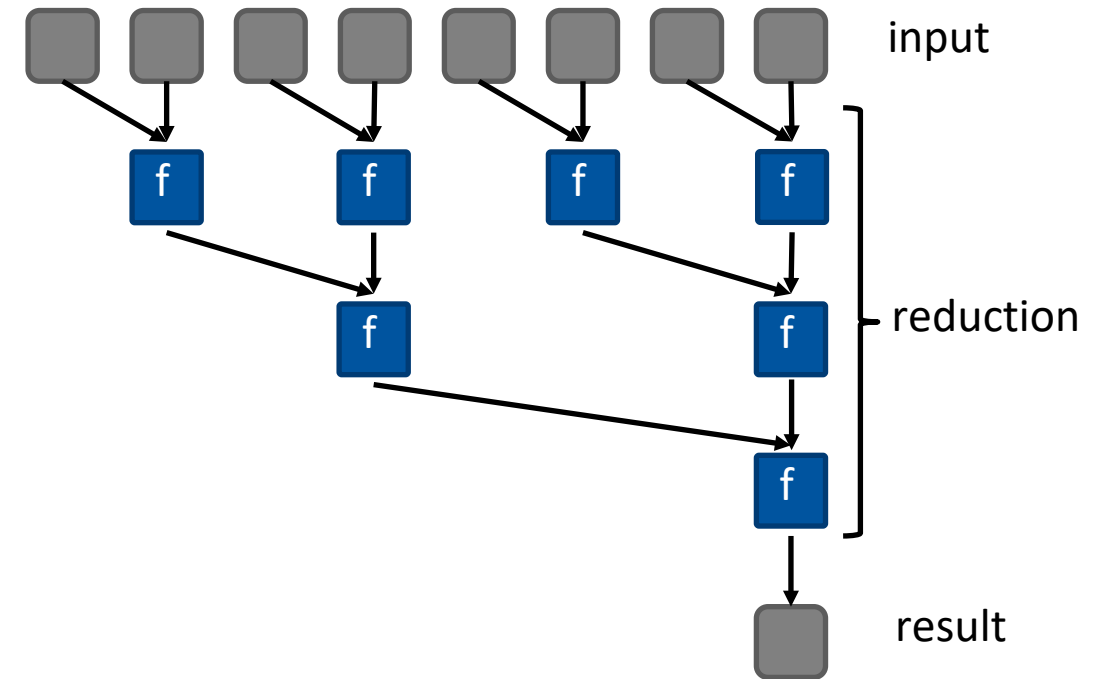
```
__global__  
void sum(float * input,  
         float output,  
         int n) {  
    int tid = blockDim.x *  
             blockIdx.x +  
             threadIdx.x;  
    if (tid < n)  
        atomicAdd(output, input[tid]);  
}
```

22.25s for 2^{30} elements



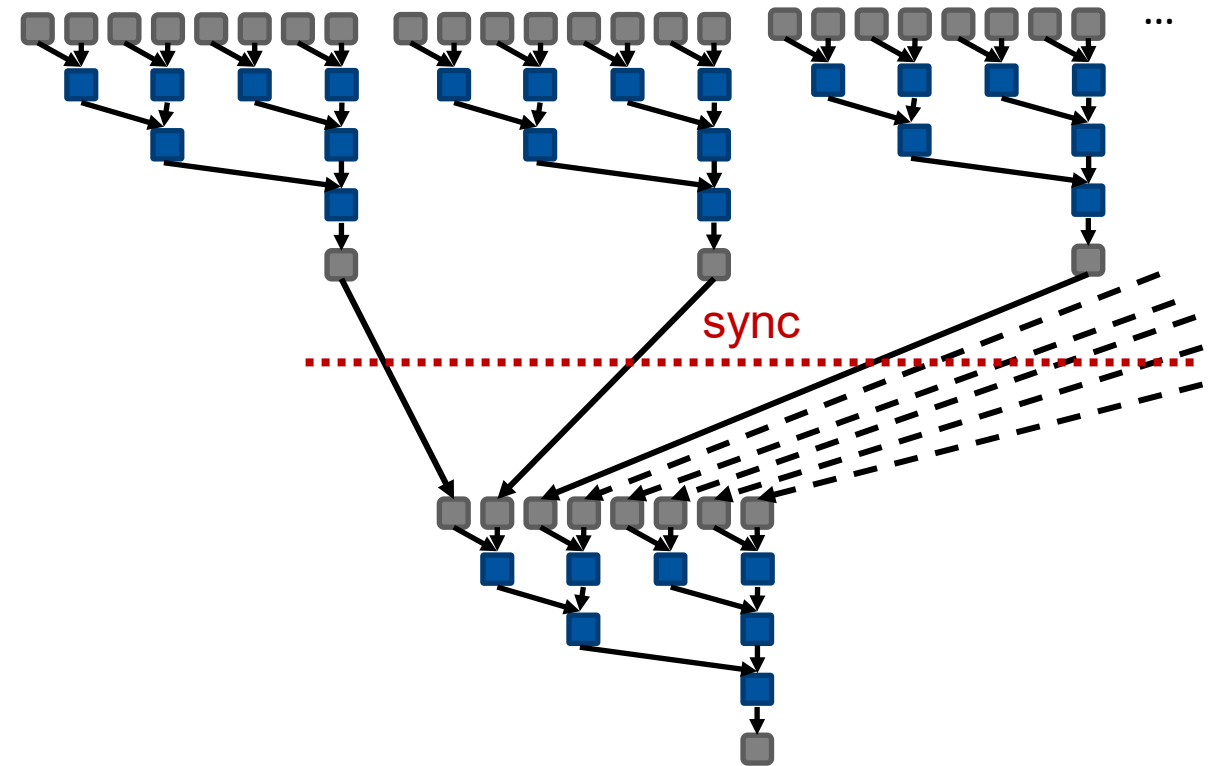
Parallel Reduction

- Natural parallelization approach is to perform a tree reduction, delivering a speedup of $n / \lg n$
 - employs an associative combiner function
 - consequently, different ordering are possible
- Reduce result on each block
- But how do we communicate partial results between thread blocks?



Reduction Pattern

- No global synchronization on the GPU
 - Inefficient and expensive to build in hardware
- Solution: Synchronize via kernel launch
 - Decomposition into multiple kernels



Reduction

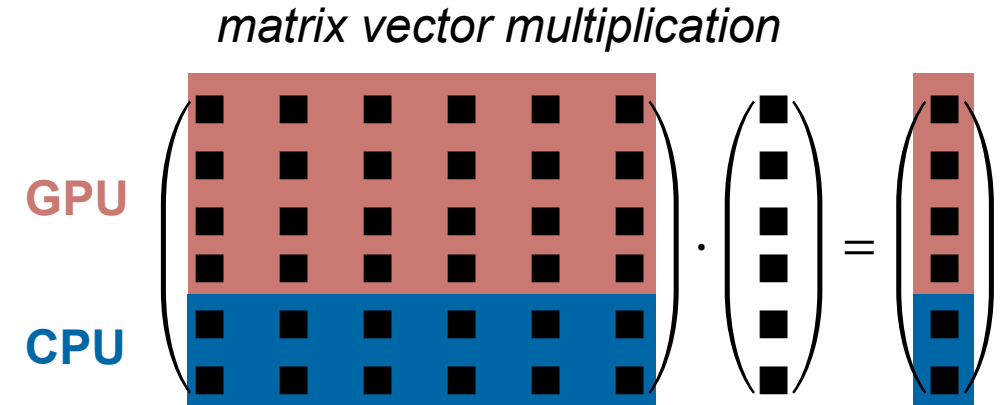
```
template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n)
{
    extern __shared__ int sdata[];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockSize*2) + tid;
    unsigned int gridSize = blockSize*2*gridDim.x;
    sdata[tid] = 0;
    while (i < n) { sdata[tid] += g_idata[i] + g_idata[i+blockSize]; i += gridSize; }
    __syncthreads();
    if (blockSize >= 512) { if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads(); }
    if (blockSize >= 256) { if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads(); }
    if (blockSize >= 128) { if (tid < 64) { sdata[tid] += sdata[tid + 64]; } __syncthreads(); }
    if (tid < 32) {
        if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
        if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
        if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
        if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
        if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
        if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
    }
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

0.38s for 2^{30} elements for
an optimized version

Efficient reduction implementations with
CUDA are hard! Use a library like Thrust
or CUB or see the NVIDIA SDK for hints.

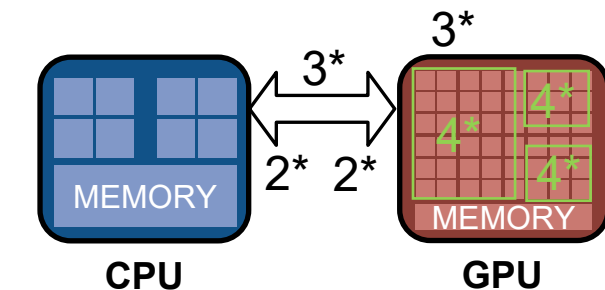
Heterogeneous Computing

- Heterogeneous Computing
 - CPU & GPU are (fully) utilized
- Challenge: load balancing
- Domain decomposition
 - If load is known beforehand, static decomposition
 - Exchange data if needed (e.g. halos)



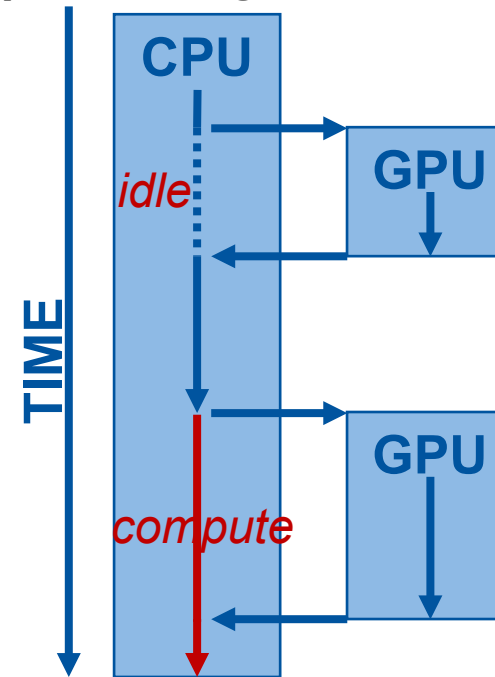
Asynchronous Operations

- Definition
 - Synchronous: Control does not return until accelerator action is complete
 - Asynchronous: Control returns immediately
- Asynchronicity allows, e.g.,
 1. Heterogeneous computing (CPU + GPU) →
 2. Overlap of PCIe transfers in both directions
 3. Overlap of data transfers and computation
 4. Simultaneous execution of several kernels (if resources are available)



<num>* Can be executed simultaneously

processing flow (simplified)



Asynchronous Operations

- Asynchronous operations with streams
 - Declare a stream handle: `cudaStream_t stream;`
 - Allocate a stream: `cudaStreamCreate(&stream);`
 - Deallocate and synchronize host until queue is empty: `cudaStreamDestroy(stream);`
- Place work in streams: `kernel<<<blocks , threads, smem, stream>>>();`
- Asynchronous data transfers: `cudaMemcpyAsync(dst, src, size, dir, stream);`

```
cudaStream_t stream1, stream2;  
cudaStreamCreate(&stream1);  
cudaStreamCreate(&stream2);  
foo<<<blocks, threads, 0, stream1>>>();  
bar<<<blocks, threads, 0, stream2>>>(); // concurrent execution of foo and bar  
cudaStreamDestroy(stream1);  
cudaStreamDestroy(stream2); // CPU waits until both functions are executed
```

GPU Performance Summary

- Data parallelism
 - Massive parallelism required
 - Use data parallel friendly data structures
- GPU is best for simple instructions
 - Leave tough operations to CPU
- Latency hiding
 - Keep the GPU busy: needs enough parallelism
- Avoid data dependencies and branch divergence
 - Can lead to wrong results or may require serialization
- Load balancing
 - Divide work into equal chunks