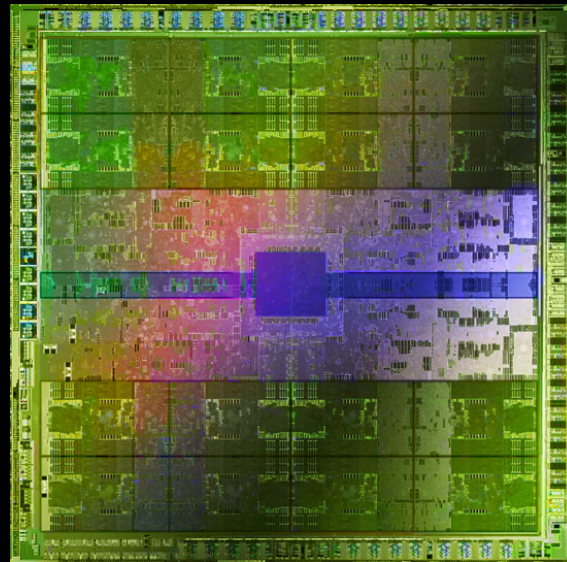
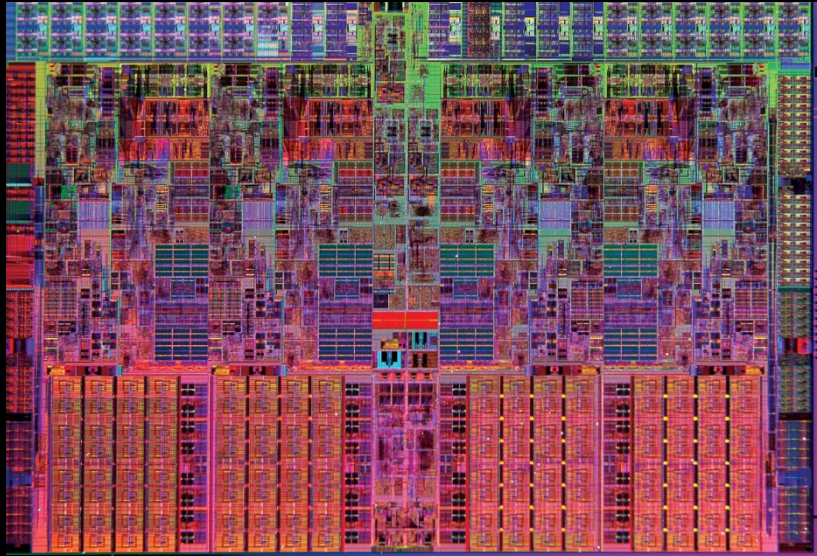


CUDA Memory Model

Anthony Morse

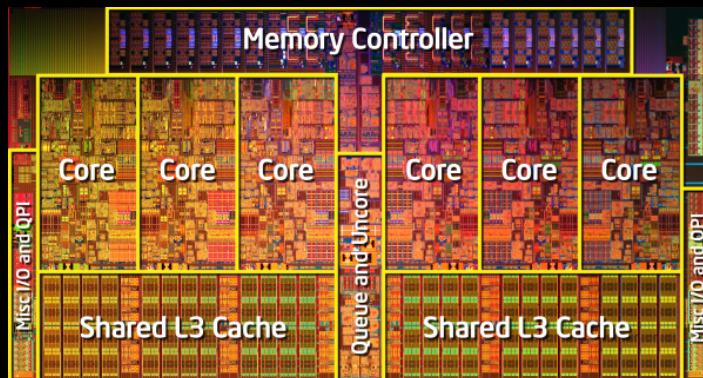


Abstraction vs. Performance

- CPU has a flat memory model
 - All cores can access any memory without restriction
- In practice, CPU hardware has typically three levels of caches
 - The popular trend of abstraction resulted in many programmers being further and further removed from the underlying hardware
 - Compilers implement abstractions into a level understood by the hardware
 - Great in theory but in reality this is more like a marketing dream
- For real performance a programmer needs to understand **cache**

Cache

- Hi-speed memory bank physically close to the processor core
- CPU has typically three cache levels (L1, L2, L3)
 - L1 – fastest but limited size (16K, 32K, 64K)
 - L2 – slower but much larger (256K, 512K)
 - L3 – slowest but largest (8MB, 16MB, 24MB)

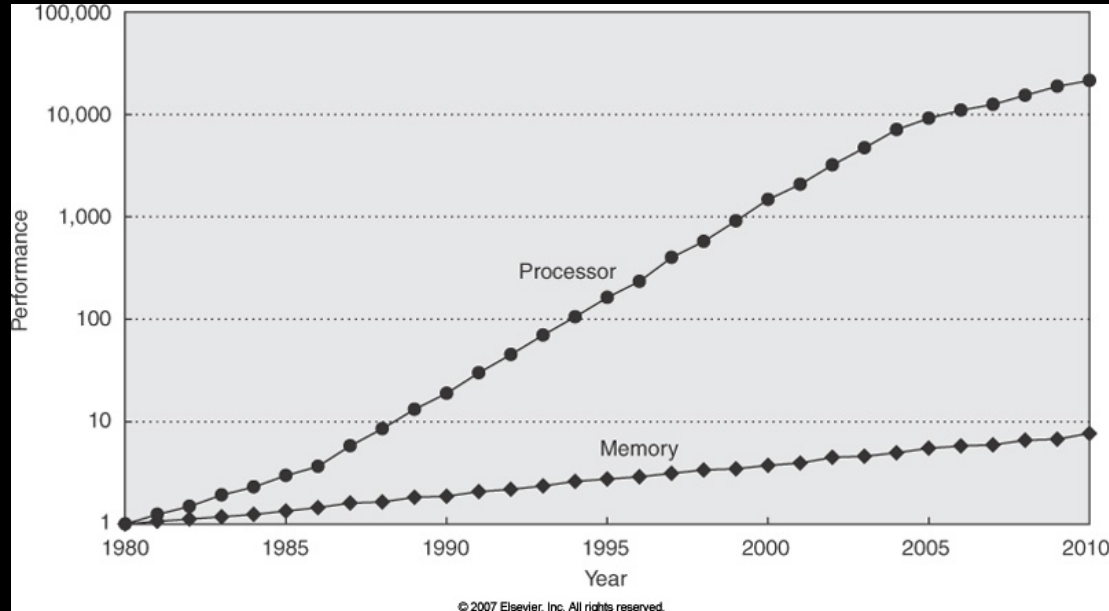


Temporal Locality

- Most programs execute in a serial fashion with various looping constructs in terms of their execution flow
- If a program calls a function it is likely that it will need to call it soon again
- If a program accesses a memory location it is likely it will need to access the same location soon again
- This is the main principle of **temporal locality**
 - reusing data and re-executing the same code

Memory Wall

- The growing disparity of speed between the chip and performing off-chip memory transactions

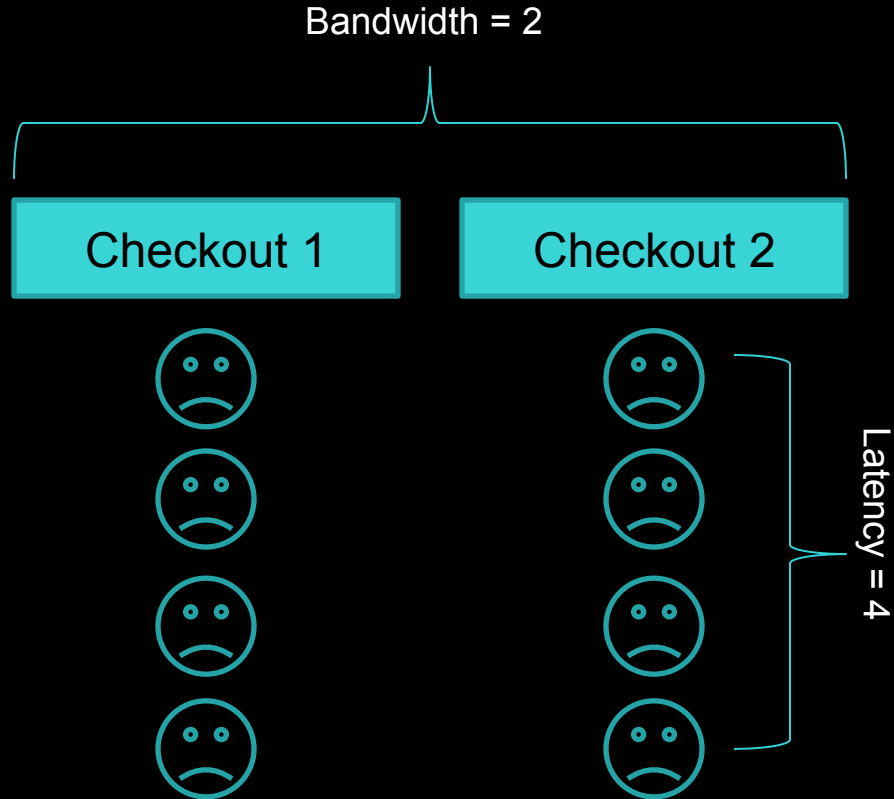


Memory Wall

- CPU is moving so fast that it left the memory far behind
 - The CPU is constantly dragged down by sluggish memory
 - DRAM clock speeds have always been historically slower than processor clock speeds (e.g. 1.6Ghz vs. 3.5Ghz)
 - Without a cache to provide a quick access to areas of memory, DRAM bandwidth would insufficient for the CPU
 - CPU would be very much memory **bandwidth** and **latency** limited
- **Bandwidth** is the amount of data read/stored in a given time
- **Latency** is the time that takes to respond to a fetch request

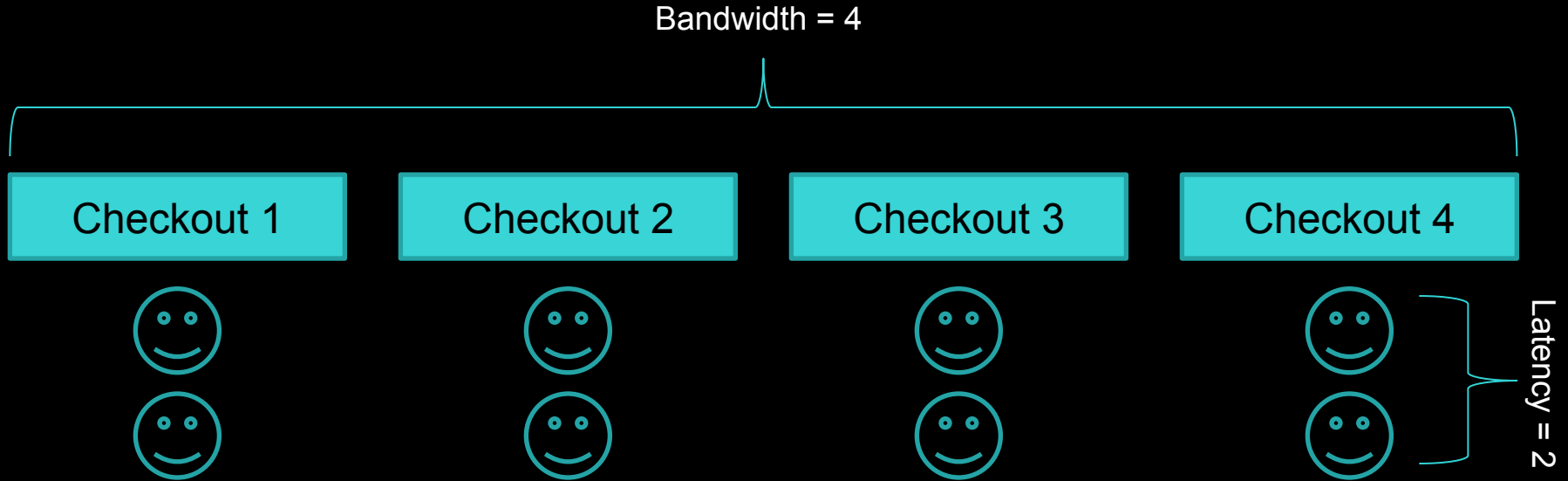
Bandwidth and Latency

Supermarket checkout example



Bandwidth and Latency

Supermarket checkout example



Bandwidth and Latency

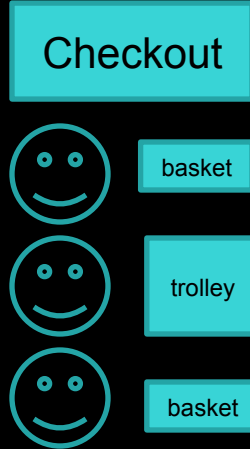
Supermarket checkout example

- **Adding new checkpoints costs more money**
 - New checkpoint area could have been used for products
 - New shop assistants needs to be paid
- **Similar tradeoffs occur in processor design in terms of memory bus width and clock rate of the memory device**
 - There is only so much silicon space on the device
 - The bus width of external memory is limited by the number of physical pins on the processor

Transaction Overhead

Supermarket checkout example

- There is a certain overhead associated with processing the payment for each customer
- The shop owners love cart the trolley shoppers as they can be processed more **efficiently**
- More of the checkout person's time is spent checking out groceries rather than in the overhead of processing the payment
- We see the same on CPUs and GPUs
 - To achieve peak memory efficiency we need lots of large transactions

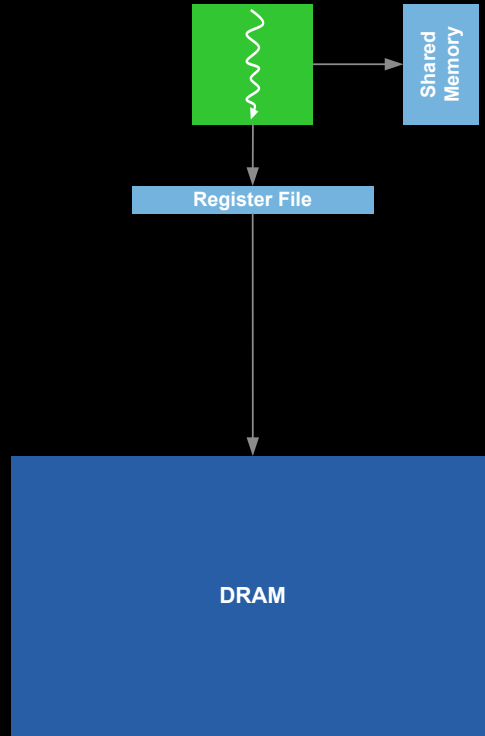


GPU Cache

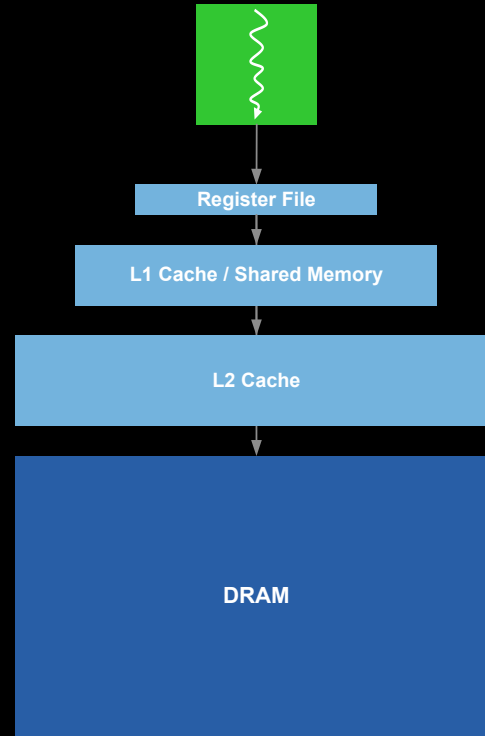
- **Prior to Fermi architecture there was no CPU-like cache**
 - GPU relies primarily on programmer-managed cache
- **Fermi was first to introduce the concept of nonprogrammer-managed cache**
- **L1 cache is both programmer and hardware managed**
- **L2 cache is shared across all Streaming Multiprocessors (SM)**

GPU Cache

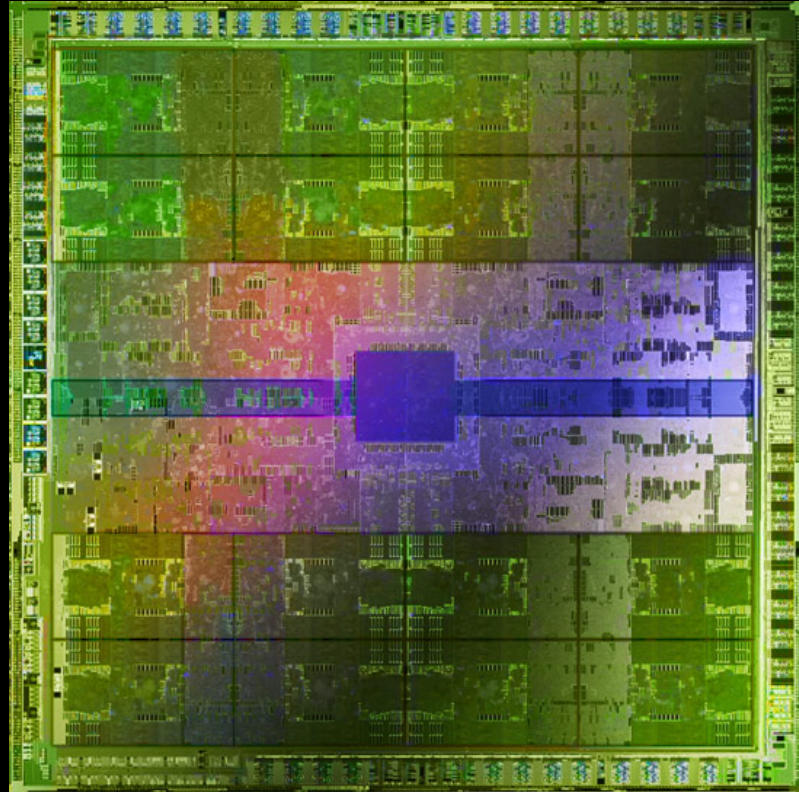
Tesla Memory Hierarchy



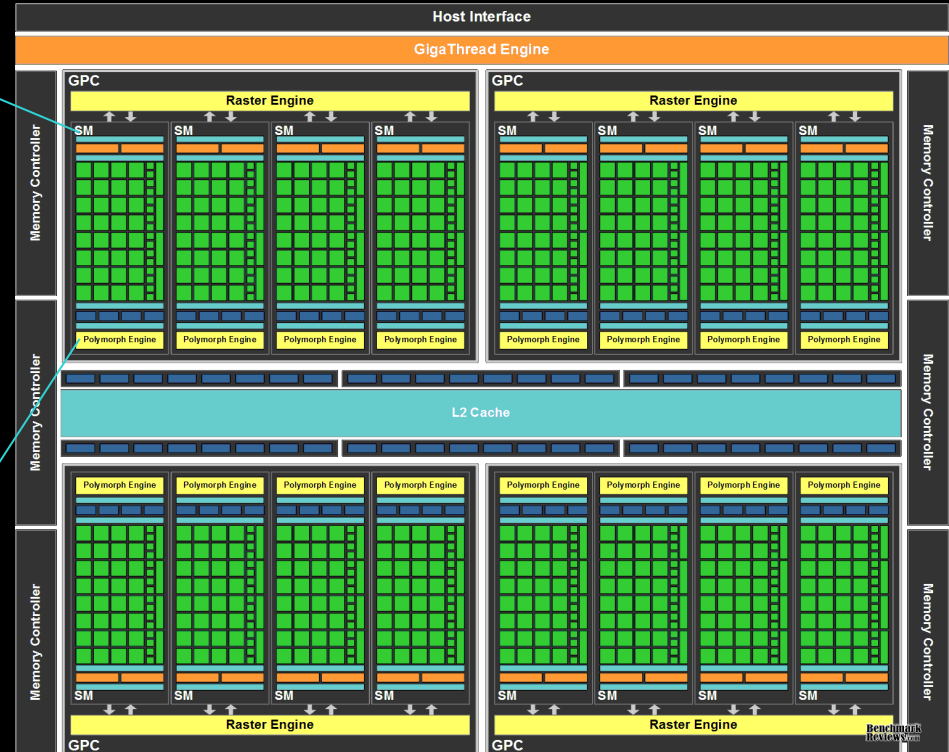
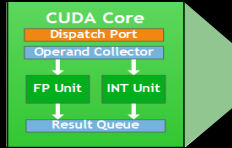
Fermi Memory Hierarchy



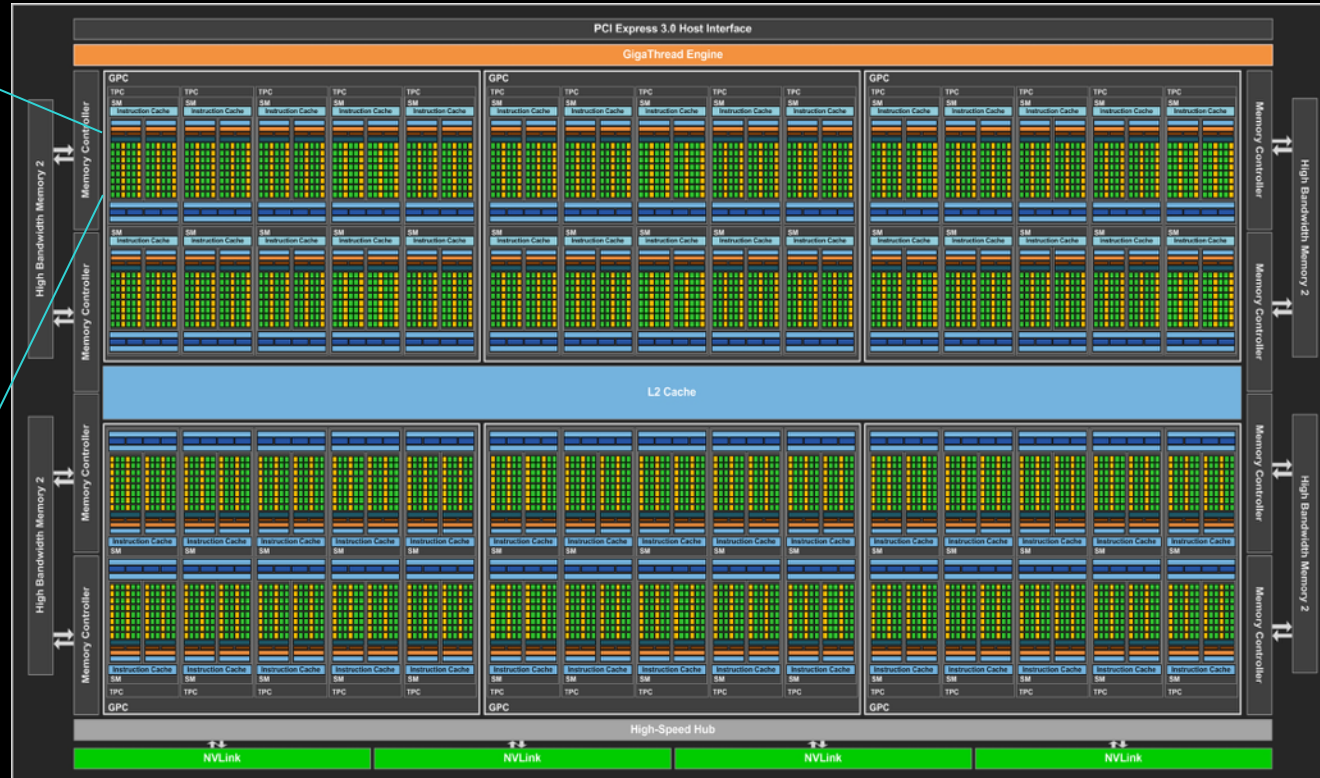
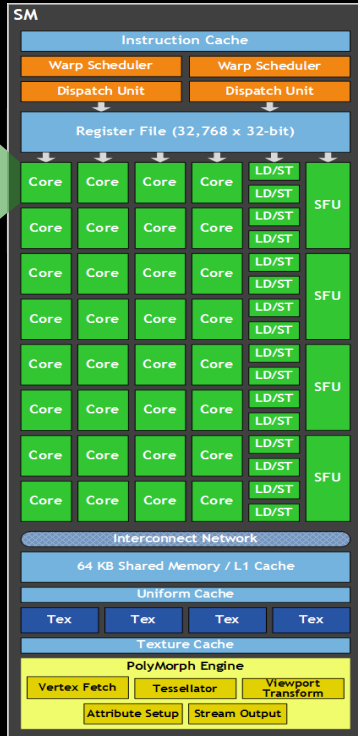
Fermi Architecture



Fermi Architecture



Pascal Architecture



Multiple Memory Scopes

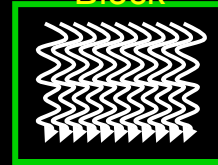
- **Per-thread private memory**
 - Each thread has its own local memory
 - Stacks, other private data
- **Per-thread-block shared memory**
 - Small memory close to the processor, low latency
 - Allocated per thread block
- **Main memory**
 - GPU frame buffer
 - Can be accessed by any thread in any thread block

Thread



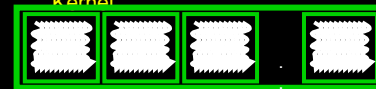
Per-thread
Local Memory

Block

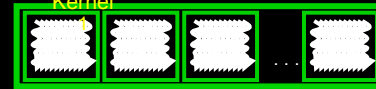


Per-block
Shared
Memory

Kernel



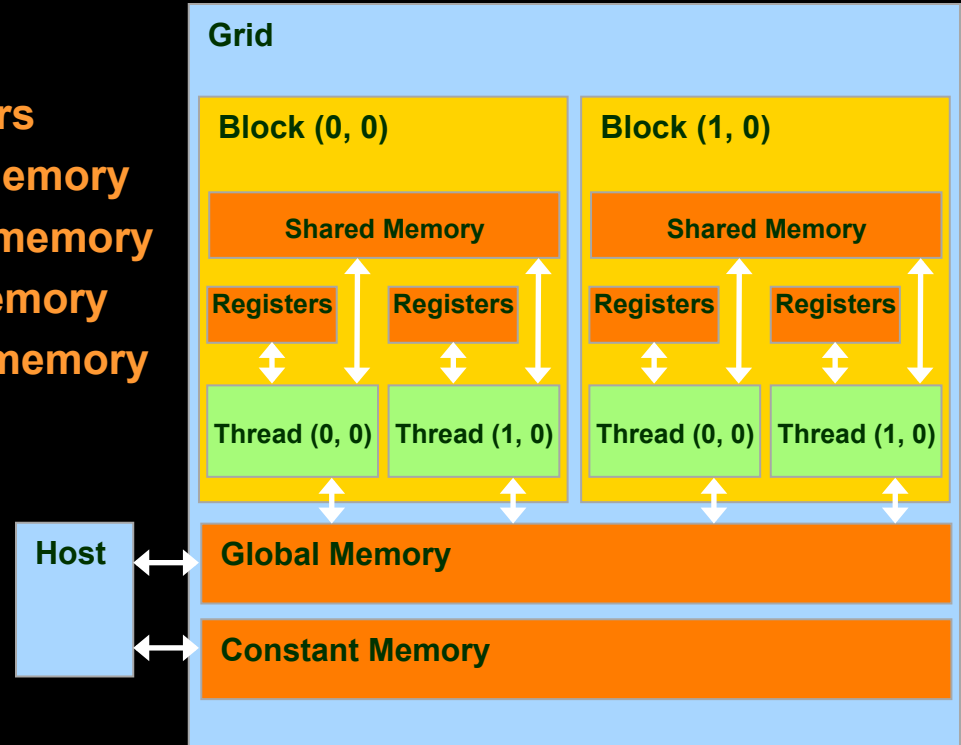
Kernel



Per-device
Global
Memory

CUDA Memories

- Each thread can:
 - Read/write per-thread **registers**
 - Read/write per-thread **local memory**
 - Read/write per-block **shared memory**
 - Read/write per-grid **global memory**
 - Read/only per-grid **constant memory**



CUDA Variable Type Qualifiers

Variable declaration	Memory	Scope	Lifetime
<code>int var;</code>	register	thread	thread
<code>int array_var[10];</code>	local	thread	thread
<code>__shared__ int shared_var;</code>	shared	block	block
<code>__device__ int global_var;</code>	global	grid	application
<code>__constant__ int constant_var;</code>	constant	grid	application

- **“automatic” scalar variables** without qualifier reside in a register
 - compiler will spill to thread local memory
- **“automatic” array variables** without qualifier reside in thread-local memory

CUDA Variable Type Performance

Variable declaration	Memory	Penalty
<code>int var;</code>	register	1x
<code>int array_var[10];</code>	local / global	~100x - ~10,000x
<code>__shared__ int shared_var;</code>	shared	~100x
<code>__device__ int global_var;</code>	global	~10,000x
<code>__constant__ int constant_var;</code>	constant	~10,000x

- **scalar variables** reside in fast, on-chip registers
- **shared variables** reside in fast, on-chip memories
- **thread-local arrays & global variables** reside in uncached off-chip memory
- **constant variables** reside in cached off-chip memory

CUDA Variable Type Scale

Variable declaration	Instances	Visibility
<code>int var;</code>	100,000s	1
<code>int array_var[10];</code>	100,000s	1
<code>__shared__ int shared_var;</code>	100s	100s
<code>__device__ int global_var;</code>	1	100,000s
<code>__constant__ int constant_var;</code>	1	100,000s

- 100Ks per-thread variables, R/W by 1 thread
- 100s shared variables, each R/W by 100s of threads
- 1 global variable is R/W by 100Ks threads
- 1 constant variable is readable by 100Ks threads

Where to declare variables?

Can host
access it?

Yes

No

Outside of
any function

In the
kernel

```
__constant__ int constant_var;  
__device__   int global_var;
```

```
int var;  
int array_var[10];  
__shared__ int shared_var;
```

Example – thread-local variables

```
// motivate per-thread variables with
// Ten Nearest Neighbors application
__global__ void ten_nn(float2 *result, float2 *ps, float2 *qs,
                      size_t num_qs)
{
    // p goes in a register
    float2 p = ps[threadIdx.x];

    // per-thread heap goes in off-chip memory
    float2 heap[10];

    // read through num_qs points, maintaining
    // the nearest 10 qs to p in the heap
    ...
    // write out the contents of heap to result
    ...
}
```

Example – shared variables

```
// motivate shared variables with
// Adjacent Difference application:
// compute result[i] = input[i] - input[i-1]
__global__ void adj_diff_naive(int *result, int *input)
{
    // compute this thread's global index
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

    if(i > 0)
    {
        // each thread loads two elements from global memory
        int x_i = input[i];
        int x_i_minus_one = input[i-1];

        result[i] = x_i - x_i_minus_one;
    }
}
```

Example – shared variables

```
// motivate shared variables with
// Adjacent Difference application:
// compute result[i] = input[i] - input[i-1]
__global__ void adj_diff_naive(int *result, int *input)
{
    // compute this thread's global index
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

    if(i > 0)
    {
        // what are the bandwidth requirements of this kernel?
        int x_i = input[i];
        int x_i_minus_one = input[i-1];

        result[i] = x_i - x_i_minus_one;
    }
}
```

Two loads

Example – shared variables

```
// motivate shared variables with
// Adjacent Difference application:
// compute result[i] = input[i] - input[i-1]
__global__ void adj_diff_naive(int *result, int *input)
{
    // compute this thread's global index
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

    if(i > 0)
    {
        // How many times does this kernel load input[i]?
        int x_i = input[i];
        int x_i_minus_one = input[i-1]; // once by thread i
                                         // again by thread i+1

        result[i] = x_i - x_i_minus_one;
    }
}
```

Example – shared variables

```
// motivate shared variables with
// Adjacent Difference application:
// compute result[i] = input[i] - input[i-1]
__global__ void adj_diff_naive(int *result, int *input)
{
    // compute this thread's global index
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

    if(i > 0)
    {
        // Idea: eliminate redundancy by sharing data
        int x_i = input[i];
        int x_i_minus_one = input[i-1];

        result[i] = x_i - x_i_minus_one;
    }
}
```

Example – shared variables

```
// optimized version of adjacent difference
__global__ void adj_diff(int *result, int *input)
{
    // shorthand for threadIdx.x
    int tx = threadIdx.x;
    // allocate a __shared__ array, one element per thread
    __shared__ int s_data[BLOCK_SIZE];
    // each thread reads one element to s_data
    unsigned int i = blockDim.x * blockIdx.x + tx;
    s_data[tx] = input[i];

    // avoid race condition: ensure all loads
    // complete before continuing
    __syncthreads();
    ...
}
```

Example – shared variables

```
// optimized version of adjacent difference
__global__ void adj_diff(int *result, int *input)
{
    ...
    if(tx > 0)
        result[i] = s_data[tx] - s_data[tx-1];
    else if(i > 0)
    {
        // handle thread block boundary
        result[i] = s_data[tx] - input[i-1];
    }
}
```

Example – shared variables

```
// when the size of the array isn't known at compile time...
__global__ void adj_diff(int *result, int *input)
{
    // use extern to indicate a __shared__ array will be
    // allocated dynamically at kernel launch time
    extern __shared__ int s_data[];
    ...
}

// pass the size of the per-block array, in bytes, as the third
// argument to the triple chevrons
adj_diff<<<num_blocks, block_size, block_size * sizeof(int)>>>(r,i);
```

Optimisation Analysis

Implementation	Original	Improved
Global Loads	$2N$	$N + N/BLOCK_SIZE$
Global Stores	N	N
Throughput	36.8 GB/s	57.5 GB/s
SLOCs	18	35
Relative Improvement	1x	1.57x
Improvement/SLOC	1x	0.81x

- **Experiment performed on a GT200 chip**
 - Improvement likely better on an older architecture
 - Improvement likely worse on a newer architecture
- **Optimisations tend to come with a development cost**

About Pointers

- Yes, you can use them!
- You can point at any memory space per se:

```
__device__ int my_global_variable;  
__constant__ int my_constant_variable = 13;  
  
__global__ void foo(void)  
{  
    __shared__ int my_shared_variable;  
  
    int *ptr_to_global = &my_global_variable;  
    const int *ptr_to_constant = &my_constant_variable;  
    int *ptr_to_shared = &my_shared_variable;  
    ...  
    *ptr_to_global = *ptr_to_shared;  
}
```

About Pointers

- Pointers aren't typed on memory space
 - `__shared__ int *ptr;`
 - `ptr` is a `__shared__` pointer variable, not a pointer to a `__shared__` variable!

Don't confuse the compiler!

```
__device__ int my_global_variable;
__global__ void foo(int *input)
{
    __shared__ int my_shared_variable;

    int *ptr = 0;
    if(input[threadIdx.x] % 2)
        ptr = &my_global_variable;
    else
        ptr = &my_shared_variable;
    // where does ptr point?
}
```

Advice

- Prefer dereferencing pointers in simple, regular access patterns
- Avoid propagating pointers
- Avoid pointers to pointers
 - The GPU would rather not pointer chase
 - Linked lists will not perform well
- Pay attention to compiler warning messages
 - Warning: Cannot tell what pointer points to, assuming global memory space
 - Crash waiting to happen

A Common Programming Strategy

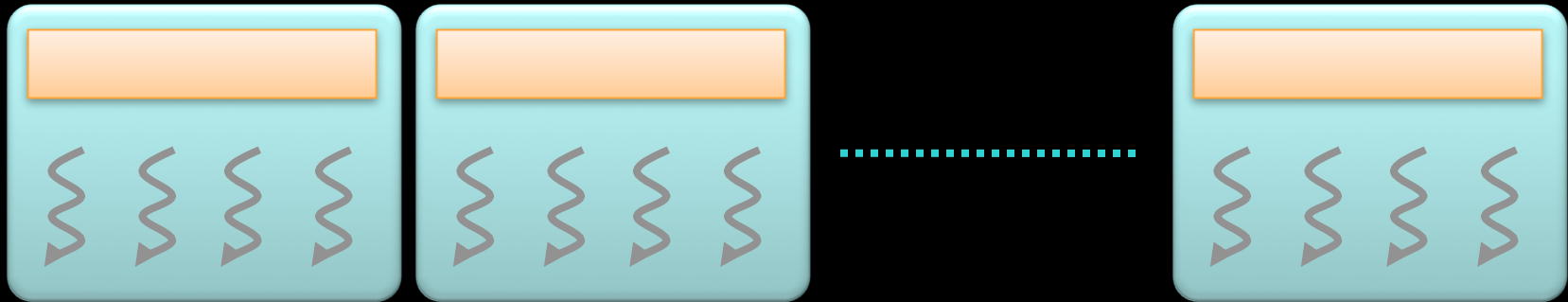
- Global memory resides in device memory (DRAM)
 - Much slower access than shared memory
- **Tile data** to take advantage of fast shared memory:
 - Generalise from adjacent_difference example
 - Divide and conquer

A Common Programming Strategy



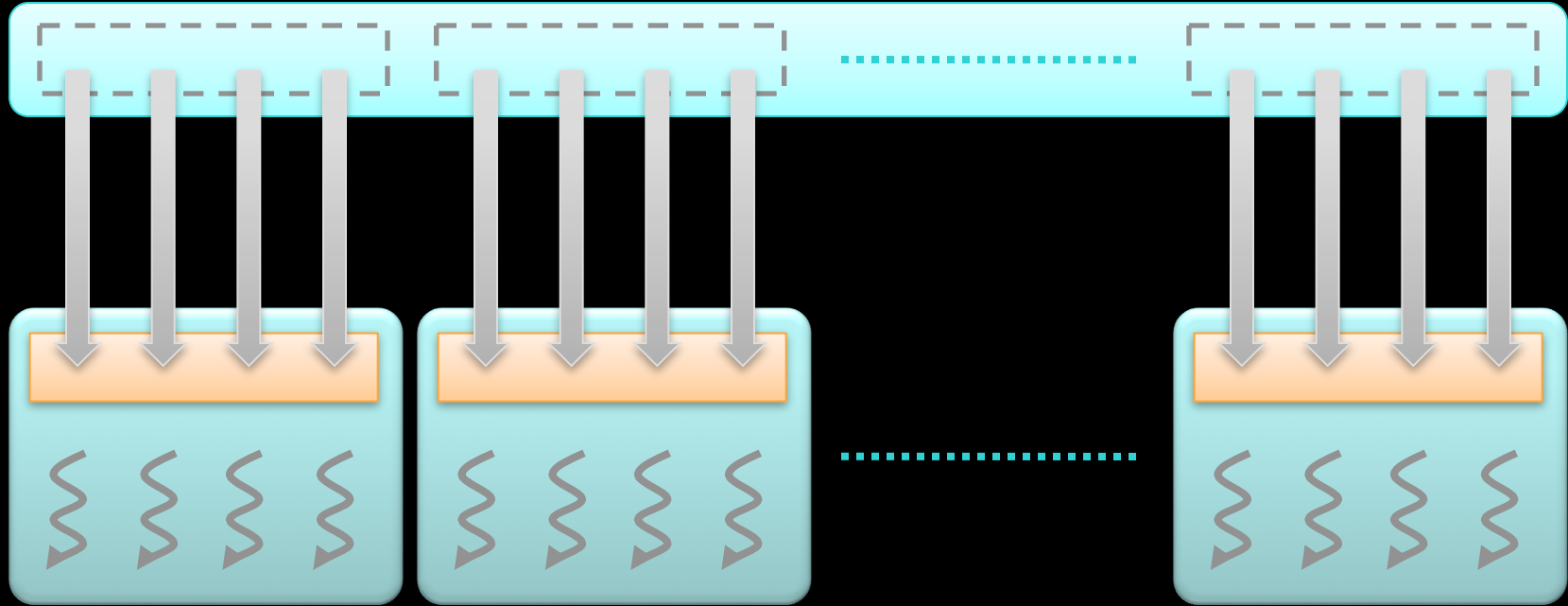
- Partition data into subsets that fit into shared memory

A Common Programming Strategy



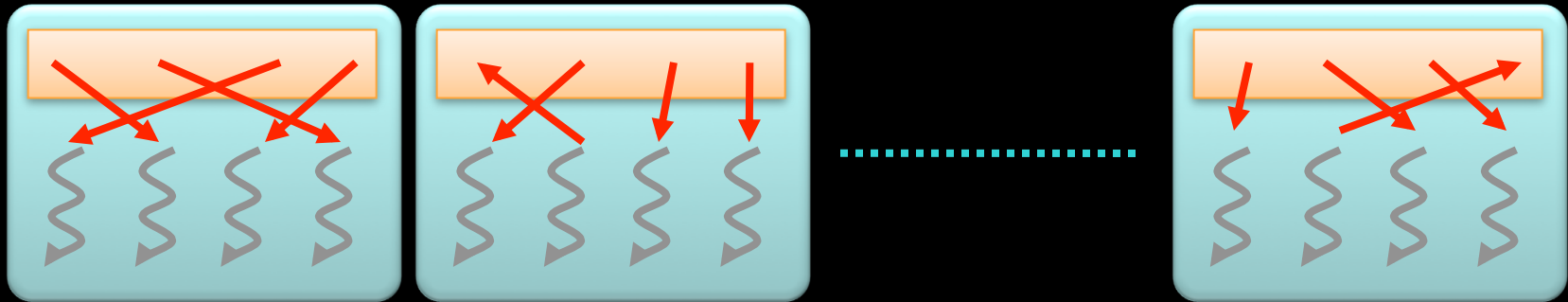
- Handle each data subset with one **thread block**

A Common Programming Strategy



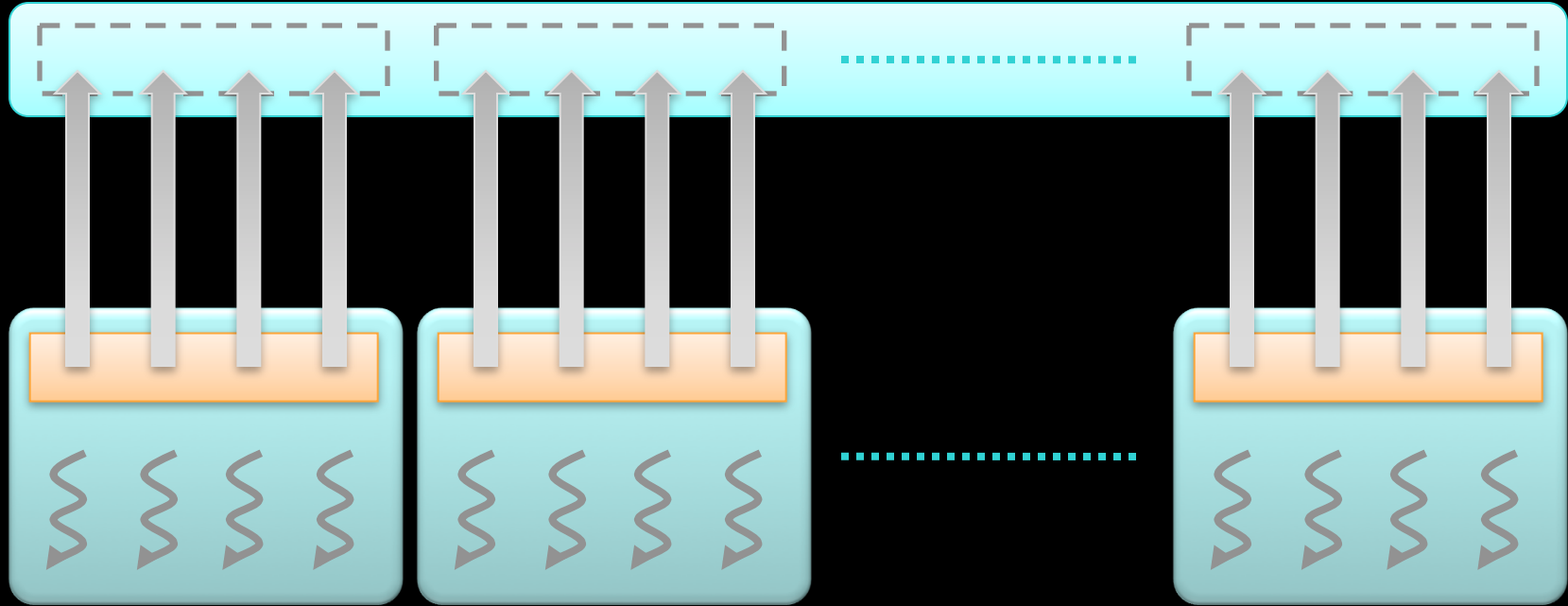
- Load the subset from global memory to shared memory, **using multiple threads to exploit memory-level parallelism**

A Common Programming Strategy



- Perform the computation on the subset from **shared memory**

A Common Programming Strategy



- Copy the result from **shared memory** back to global memory

A Common Programming Strategy

- Carefully partition data according to access patterns
- Read-only → constant memory (fast)
- R/W & shared within block → shared memory (fast)
- R/W within each thread → registers (fast)
- Indexed R/W within each thread → local memory (slow)
- R/W inputs/results → `cudaMalloc`'ed global memory (slow)

Communication Through Memory

● Question:

```
__global__ void race(void)
{
    __shared__ int my_shared_variable;
    my_shared_variable = threadIdx.x;

    // what is the value of
    // my_shared_variable?
}
```

Communication Through Memory

- This is a **race condition**
- The result is **undefined**
- The order in which threads access the variable is undefined without explicit coordination
- Use barriers (e.g., **__syncthreads**) or atomic operations (e.g., **atomicAdd**) to enforce **well-defined** semantics

Communication Through Memory

- Use `__syncthreads` to ensure data is ready for access

```
__global__ void share_data(int *input)
{
    __shared__ int data[BLOCK_SIZE];
    data[threadIdx.x] = input[threadIdx.x];
    __syncthreads();
    // the state of the entire data array
    // is now well-defined for all threads
    // in this block
}
```

Communication Through Memory

- Use **atomic operations** to ensure exclusive access to a variable

```
// assume *result is initialized to 0
__global__ void sum(int *input, int *result)
{
    atomicAdd(result, input[threadIdx.x]);

    // after this kernel exits, the value of
    // *result will be the sum of the input
}
```

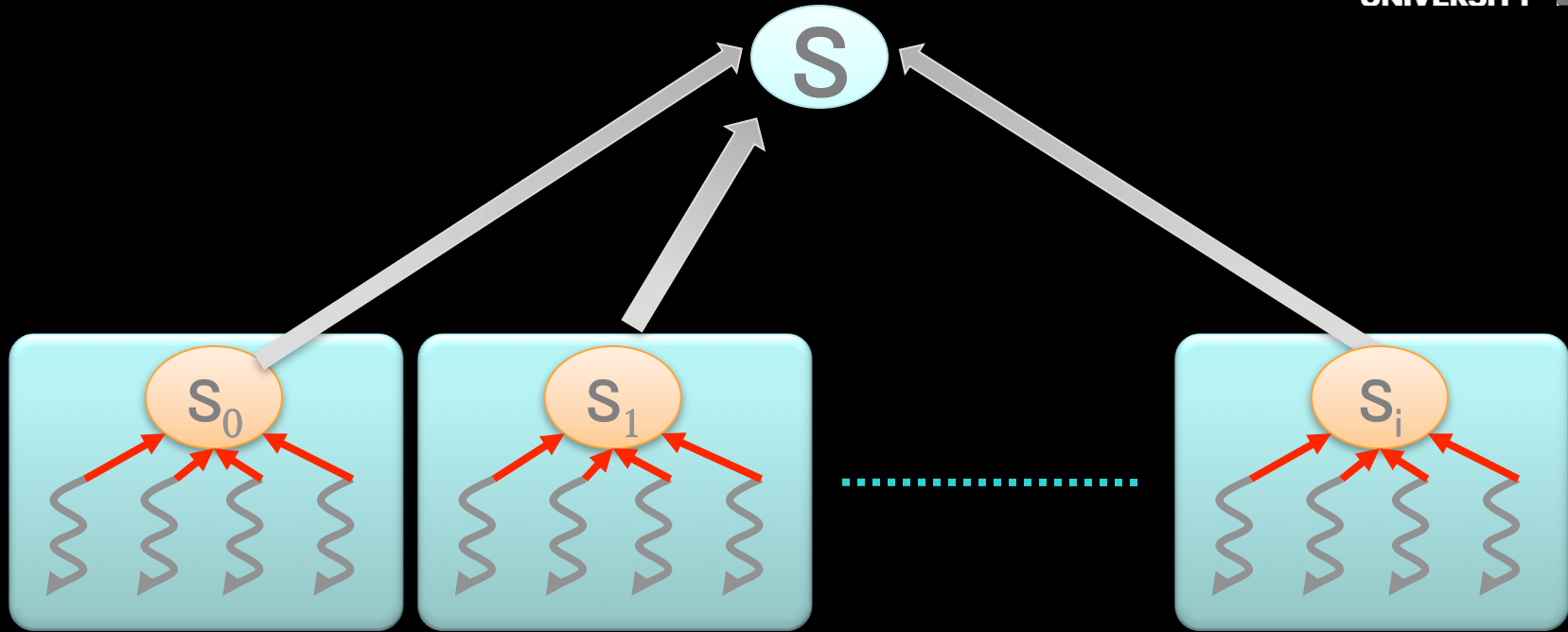
Resource Contention

- Atomic operations aren't cheap!
- They imply **serialised access** to a variable

```
__global__ void sum(int *input, int *result)
{
    atomicAdd(result, input[threadIdx.x]);
}

...
// how many threads will contend
// for exclusive access to result?
sum<<<B,N/B>>>(input,result);
```

Hierarchical Atomics



- **Divide & Conquer**
 - Per-thread **atomicAdd** to a **__shared__** partial sum
 - Per-block **atomicAdd** to the total sum

Hierarchical Atomics

```
__global__ void sum(int *input, int *result)
{
    __shared__ int partial_sum;

    // thread 0 is responsible for
    // initializing partial_sum
    if(threadIdx.x == 0)
        partial_sum = 0;
    __syncthreads();

    ...
}
```

Hierarchical Atomics

```
__global__ void sum(int *input, int *result)
{
    ...
    // each thread updates the partial sum
    atomicAdd(&partial_sum, input[threadIdx.x]);
    __syncthreads();

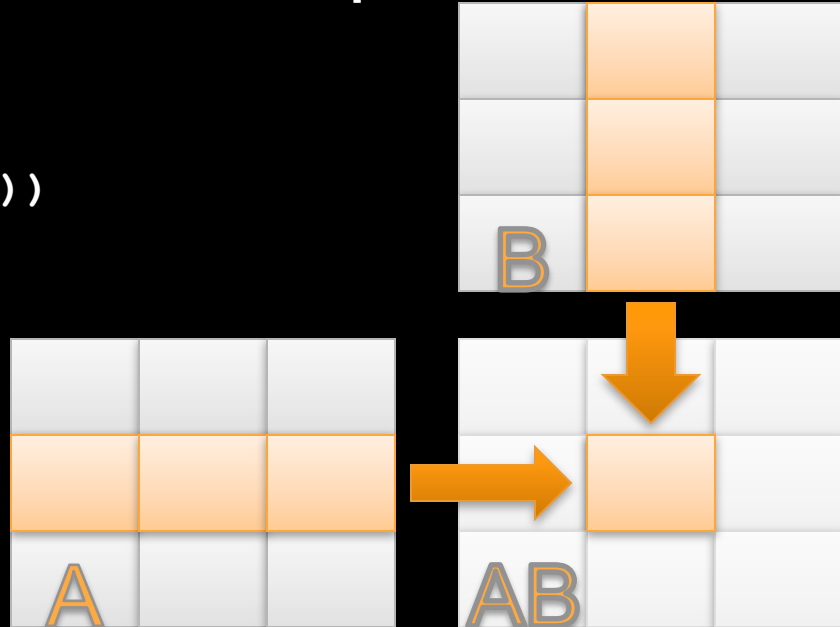
    // thread 0 updates the total sum
    if(threadIdx.x == 0)
        atomicAdd(result, partial_sum);
}
```

Advice

- Use barriers such as `__syncthreads` to wait until `__shared__` data is ready
- Prefer barriers to atomics when data access patterns are **regular** or **predictable**
- Prefer atomics to barriers when data access patterns are **sparse** or **unpredictable**
- Atomics to `__shared__` variables are **much faster** than atomics to global variables
- Don't synchronise or serialise unnecessarily

Matrix Multiplication Example

- Generalise adjacent_difference example
- $AB = A * B$
 - Each element AB_{ij}
 - $= \text{dot}(\text{row}(A, i), \text{col}(B, j))$
- Parallelisation strategy
 - Thread $\rightarrow AB_{ij}$
 - 2D kernel



First Implementation

```
__global__ void mat_mul(float *a, float *b,  
                        float *ab, int width)  
{  
    // calculate the row & col index of the element  
    int row = blockIdx.y*blockDim.y + threadIdx.y;  
    int col = blockIdx.x*blockDim.x + threadIdx.x;  
  
    float result = 0;  
  
    // do dot product between row of a and col of b  
    for(int k = 0; k < width; ++k)  
        result += a[row*width+k] * b[k*width+col];  
  
    ab[row*width+col] = result;  
}
```

NVVP – NVIDIA Visual Profiler

L2 Cache

Reads	134446749	496.144 GB/s	
Writes	131078	483.712 MB/s	
Total	134577827	496.627 GB/s	

Unified Cache

Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	402653184	495.299 GB/s	
Global Stores	131072	483.69 MB/s	
Texture Reads	268435456	990.597 GB/s	
Unified Total	671219712	1,486.379 GB/s	

Device Memory

Reads	8545850	31.536 GB/s	
Writes	157035	579.5 MB/s	
Total	8702885	32.116 GB/s	

System Memory [PCIe configuration: Gen2 x16, 5 Gbit/s]

Reads	0	0 B/s	
Writes	5	18.451 kB/s	

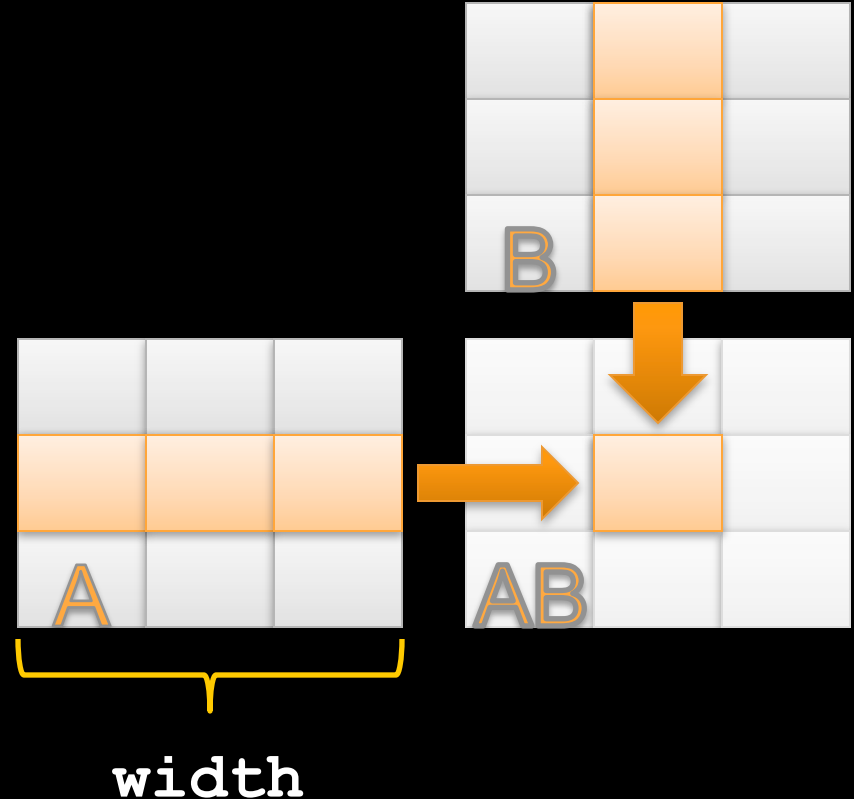
Start	10.854
End	10.864
Duration	9.592 r
Grid Size	[64,64
Block Size	[16,16
Registers/Thread	27
Shared Memory/Block	0 B
▼ Efficiency	
Global Load Efficiency	⚠ 56.2%
Global Store Efficiency	100%
Shared Efficiency	n/a
Warp Execution Efficiency	100%
Non-Predicated Warp Exe	100%
▼ Occupancy	
Achieved	94.1%
Theoretical	100%
▼ Shared Memory Configurati	
Shared Memory Requeste	96 KiB

How will this perform?

How many loads per term of dot product?	$2(a \& b) = 8 \text{ Bytes}$
How many floating point operations?	2 (multiply & addition)
Global memory access to flop ratio (GMAC)	$8 \text{ Bytes} / 2 \text{ ops} = 4 \text{ B/op}$
What is the peak fp performance of GeForce GTX 260?	805 GFLOPS
Lower bound on bandwidth required to reach peak fp performance	$\text{GMAC} * \text{Peak FLOPS} = 4 * 805 = 3.2 \text{ TB/s}$
What is the actual memory bandwidth of GeForce GTX 260?	112 GB/s
Then what is an upper bound on performance of our implementation?	$\text{Actual BW} / \text{GMAC} = 112 / 4 = 28 \text{ GFLOPS}$

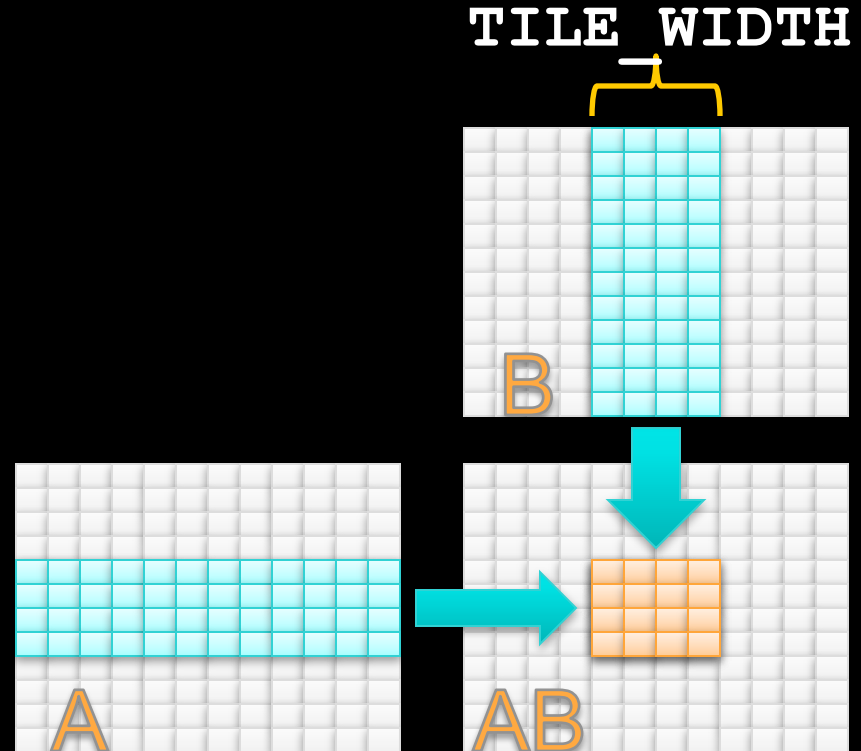
Idea: Use shared memory to reuse global data

- Each input element is read by width threads
- Load each element into shared memory and have several threads use the local version to reduce the memory bandwidth



Tiled Multiply

- Partition kernel loop into **phases**
- Load a tile of both matrices into **__shared__** each phase
- Each phase, each thread computes a **partial** result



Better Implementation

```
__global__ void mat_mul(float *a, float *b, float *ab, int width)
{
    // shorthand
    int tx = threadIdx.x, ty = threadIdx.y;
    int bx = blockIdx.x, by = blockIdx.y;

    // allocate tiles in __shared__ memory
    __shared__ float s_a[TILE_WIDTH][TILE_WIDTH];
    __shared__ float s_b[TILE_WIDTH][TILE_WIDTH];

    // calculate the row & col index
    int row = by*blockDim.y + ty;
    int col = bx*blockDim.x + tx;

    float result = 0;
```

Better Implementation

```
// loop over the tiles of the input in phases
for(int p = 0; p < width/TILE_WIDTH; ++p)
{
    // collaboratively load tiles into __shared__
    s_a[ty][tx] = a[row*width + (p*TILE_WIDTH + tx)];
    s_b[ty][tx] = b[(m*TILE_WIDTH + ty)*width + col];
    __syncthreads();

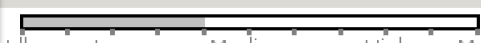
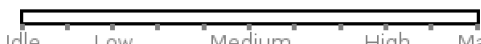
    // dot product between row of s_a and col of s_b
    for(int k = 0; k < TILE_WIDTH; ++k)
        result += s_a[ty][k] * s_b[k][tx];
    __syncthreads();
}

ab[row*width+col] = result;
}
```

Use of Barriers in `mat_mul`

- Two barriers per phase:
 - `__syncthreads` after all data is loaded into `__shared__` memory
 - `__syncthreads` after all data is read from `__shared__` memory
 - Note that second `__syncthreads` in phase `p` guards the load in phase `p+1`
- Use barriers to **guard** data
 - Guard against using uninitialised data
 - Guard against bashing live data

NVVP – NVIDIA Visual Profiler

L2 Cache			
Reads	16893147	166.554 GB/s	
Writes	131078	1.292 GB/s	
Total	17024225	167.846 GB/s	
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	33554432	165.411 GB/s	
Global Stores	131072	1.292 GB/s	
Texture Reads	16777216	165.411 GB/s	
Unified Total	50462720	332.114 GB/s	
Device Memory			
Reads	7345276	72.419 GB/s	
Writes	156986	1.548 GB/s	
Total	7502262	73.967 GB/s	
System Memory [PCIe configuration: Gen2 x16, 5 Gbit/s]			
Reads	0	0 B/s	
Writes	5	49.296 kB/s	

Start	10.864
End	10.867
Duration	3.446 r
Grid Size	[64,64
Block Size	[16,16
Registers/Thread	23
Shared Memory/Block	2 KiB
▼ Efficiency	
Global Load Efficiency	100%
Global Store Efficiency	100%
Shared Efficiency	130.8%
Warp Execution Efficiency	100%
Non-Predicated Warp Exec	99.9%
▼ Occupancy	
Achieved	99%
Theoretical	100%
▼ Shared Memory Configuratio	
Shared Memory Requeste	96 KiB

First Order Size Considerations

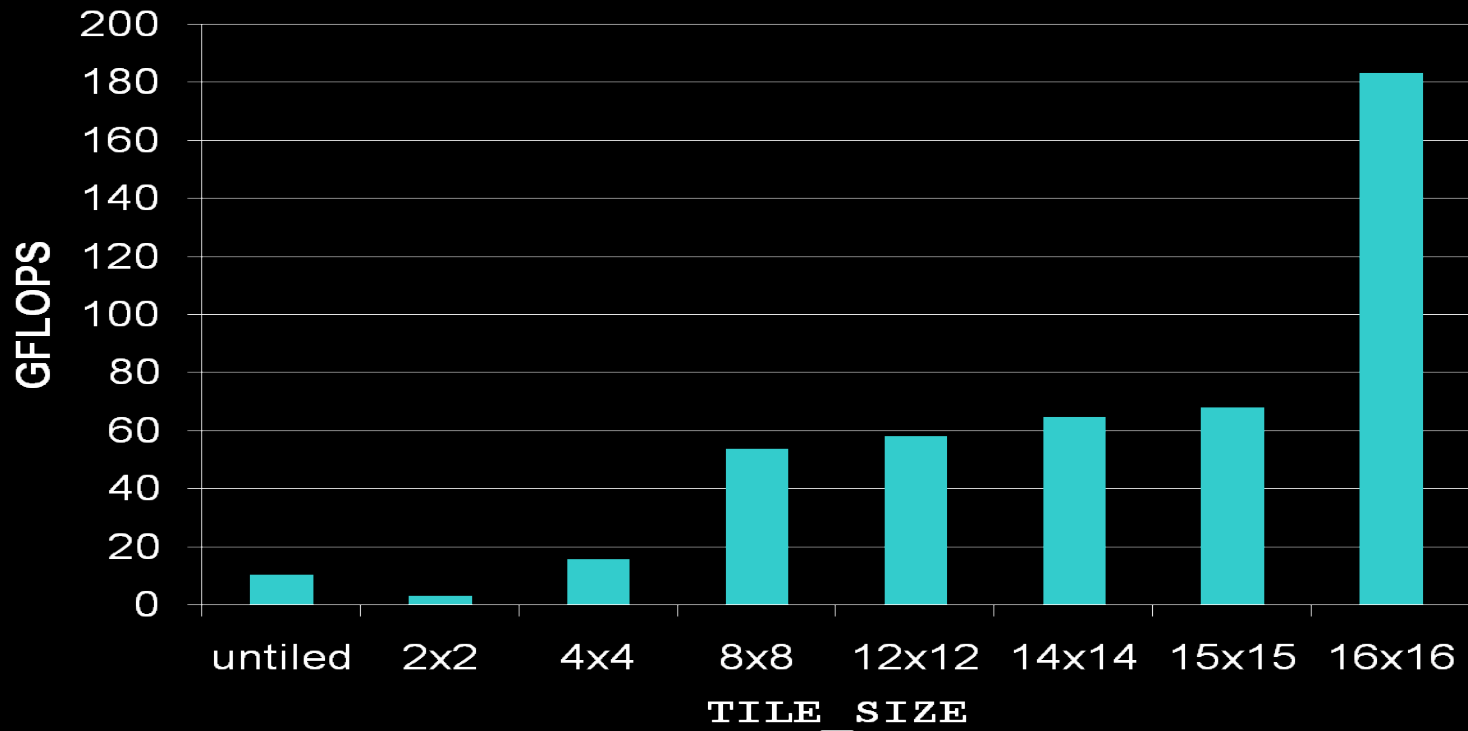
- Each **thread block** should have many threads
 - $\text{TILE_WIDTH} = 16 \rightarrow 16 * 16 = 256$ threads
- There should be many thread blocks
 - $1024 * 1024$ matrices $\rightarrow 64 * 64 = 4096$ thread blocks
 - $\text{TILE_WIDTH} = 16 \rightarrow$ gives each SM 3 blocks, 768 threads
 - Full **occupancy**
- Each thread block performs $2 * 256 = 512$ 32b loads for $256 * (2 * 16) = 8,192$ fp ops
 - Memory bandwidth no longer limiting factor

Optimization Analysis

Implementation	Original	Improved
Global Loads	$2N^3$	$2N^2 * (N/TILE_WIDTH)$
Throughput	10.7 GFLOPS	183.9 GFLOPS
SLOCs	20	44
Relative Improvement	1x	17.2x
Improvement/SLOC	1x	7.8x

- Experiment performed on a GT200
- This optimisation was clearly worth the effort
- Better performance still possible in theory

TILE_SIZE Effects



Memory Resources as Limit to Parallelism

Resource	Per GT200 SM	Full Occupancy on GT200
Registers	16384	$\leq 16384 / 768$ threads = 21 per thread
__shared__ Memory	16KB	$\leq 16\text{KB} / 8$ blocks = 2KB per block

- Effective use of different memory resources reduces the number of accesses to global memory
- These resources are **finite!**
- The more memory locations each thread requires → the fewer threads an SM can accommodate

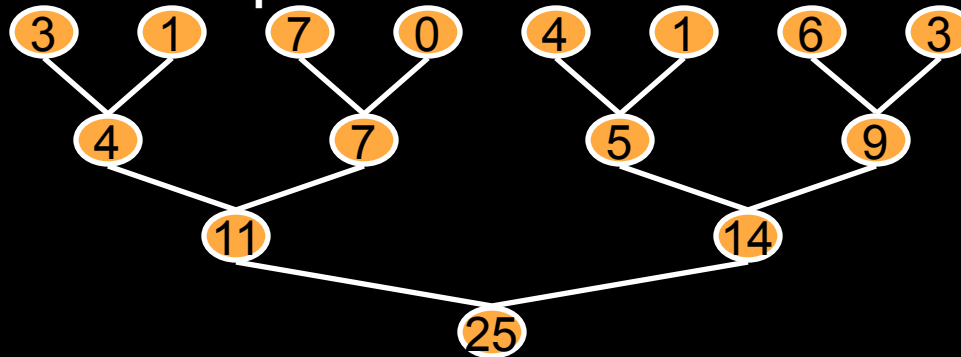
Final Thoughts

- Effective use of CUDA memory hierarchy decreases bandwidth consumption to increase **throughput**
- Use **__shared__** memory to eliminate redundant loads from global memory
 - Use **__syncthreads** barriers to protect **__shared__** data
 - Use atomics if access patterns are sparse or unpredictable
- Optimisation comes with a development cost
- Memory resources ultimately limit parallelism

Reduction

- **Reduce** vector to a single value
 - Via an associative operator (+, *, min/max, AND/OR, ...)
 - CPU: sequential implementation

```
for(int i = 0, i < n, ++i) ...
```
 - GPU: “tree”-based implementation



Serial Reduction

```
// reduction via serial iteration
float sum(float *data, int n)
{
    float result = 0;
    for(int i = 0; i < n; ++i)
    {
        result += data[i];
    }

    return result;
}
```

Parallel Reduction – Interleaved

Values (in shared memory)

Step 1
Stride 1

Thread
IDs

Values

Step 2
Stride 2

Thread
IDs

Values

Step 3
Stride 4

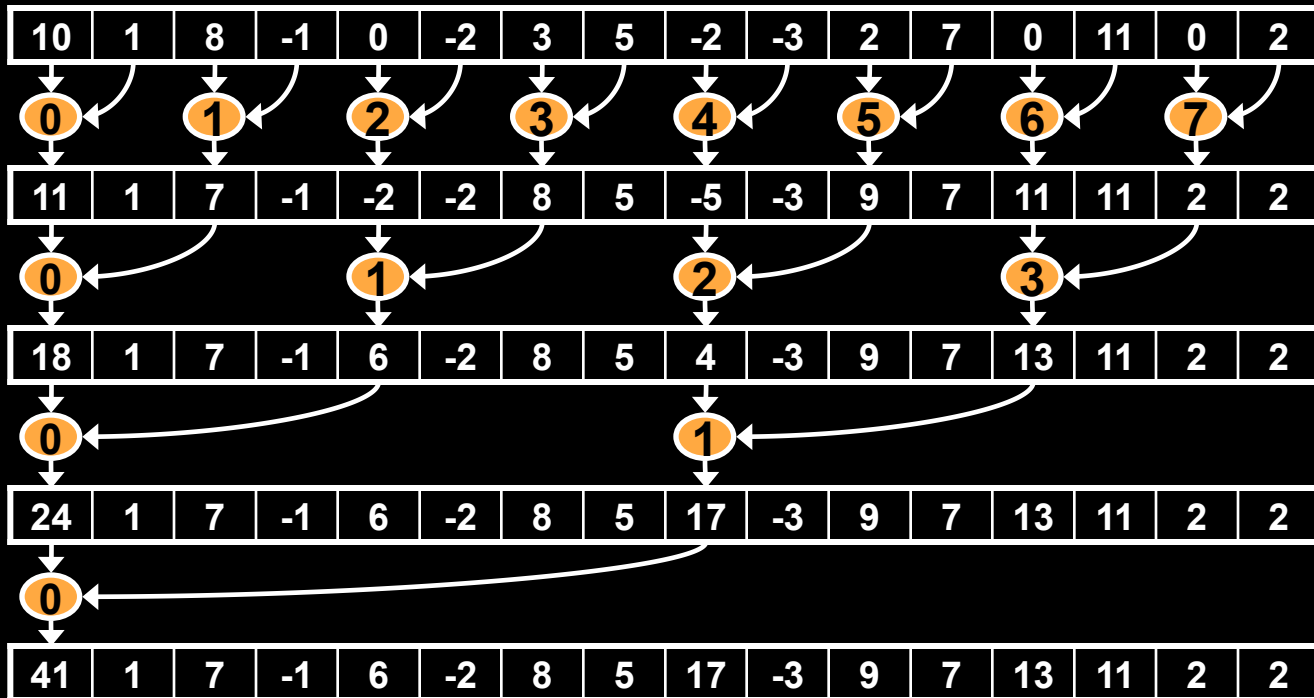
Thread
IDs

Values

Step 4
Stride 8

Thread
IDs

Values

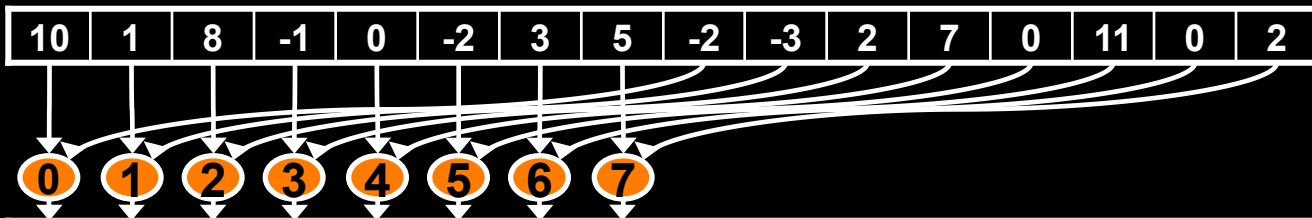


Parallel Reduction – Contiguous

Values (in shared memory)

Step 1
Stride 8

Thread
IDs



Values

8	-2	10	6	0	9	3	7	-2	-3	2	7	0	11	0	2
---	----	----	---	---	---	---	---	----	----	---	---	---	----	---	---

Step 2
Stride 4

Thread
IDs



Values

8	7	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2
---	---	----	----	---	---	---	---	----	----	---	---	---	----	---	---

Step 3
Stride 2

Thread
IDs



Values

21	20	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2
----	----	----	----	---	---	---	---	----	----	---	---	---	----	---	---

Step 4
Stride 1

Thread
IDs



Values

41	20	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2
----	----	----	----	---	---	---	---	----	----	---	---	---	----	---	---

CUDA Reduction

```
__global__ void block_sum(float *input,
                          float *results,
                          size_t n)
{
    extern __shared__ float sdata[];
    int i = ..., int tx = threadIdx.x;

    // load input into __shared__ memory
    float x = 0;
    if(i < n)
        x = input[i];
    sdata[tx] = x;
    __syncthreads();
}
```

CUDA Reduction

```
// block-wide reduction in __shared__ mem
for(int offset = blockDim.x / 2;
    offset > 0;
    offset >>= 1)
{
    if(tx < offset)
    {
        // add a partial sum upstream to our own
        sdata[tx] += sdata[tx + offset];
    }
    __syncthreads();
}
```

CUDA Reduction

```
// finally, thread 0 writes the result
if(threadIdx.x == 0)
{
    // note that the result is per-block
    // not per-thread
    results[blockIdx.x] = sdata[0];
}
```

CUDA Reduction

```
// global sum via per-block reductions
float sum(float *d_input, size_t n)
{
    size_t block_size = ..., num_blocks = ...;

    // allocate per-block partial sums
    // plus a final total sum
    float *d_sums = 0;
    cudaMalloc((void**)&d_sums,
        sizeof(float) * (num_blocks + 1));
    ...
}
```

CUDA Reduction

```
// reduce per-block partial sums
int smem_sz = block_size*sizeof(float);
block_sum<<<num_blocks,block_size,smem_sz>>>
    (d_input, d_sums, n);

// reduce partial sums to a total sum
block_sum<<<1,block_size,smem_sz>>>
    d_sums, d_sums + num_blocks, num_blocks);

// copy result to host
float result = 0;
cudaMemcpy(&result, d_sums+num_blocks, ...);
return result;

}
```

Thank you