Practical Course: GPU Programming in Computer Vision

Optimization

Björn Häfner, Robert Maier, David Schubert

Technische Universität München
Department of Informatics
Computer Vision Group

Summer Semester 2018
September 17 - October 15

Outline

1. Performance metrics of algorithms running on a GPU
   - occupancy
   - data bandwidth and instruction throughput
2. Maximize instruction throughput
   - branch divergence
3. Maximize memory throughput
   - pitched allocation for images
4. parallel reduction: an example of optimization

Outline

1. Performance metrics of algorithms running on a GPU
   - occupancy
   - data bandwidth and instruction throughput
2. Maximize instruction throughput
   - branch divergence
3. Maximize memory throughput
   - pitched allocation for images
4. parallel reduction: an example of optimization

occupancy

$$\text{occupancy} = \frac{\text{active threads}}{\text{max. threads per SM}}$$

- Multiprocessors (SMs) can have many more active threads than there are CUDA Cores
- High occupancy is important, because if some threads stall, the SM can switch to others
- Pool of limited resources per SM
- Occupancy determined by
  - Register usage per thread
  - Shared memory per block

occupancy

$$\text{occupancy} = \frac{\text{active threads}}{\text{max. threads per SM}}$$

- Multiprocessors (SMs) can have many more active threads than there are CUDA Cores
- High occupancy is important, because if some threads stall, the SM can switch to others
- Pool of limited resources per SM
- Occupancy determined by
  - Register usage per thread
  - Shared memory per block

occupancy

$$\text{occupancy} = \frac{\text{active threads}}{\text{max. threads per SM}}$$

- Multiprocessors (SMs) can have many more active threads than there are CUDA Cores
- High occupancy is important, because if some threads stall, the SM can switch to others
- Pool of limited resources per SM
- Occupancy determined by
  - Register usage per thread
  - Shared memory per block

occupancy

$$\text{occupancy} = \frac{\text{active threads}}{\text{max. threads per SM}}$$

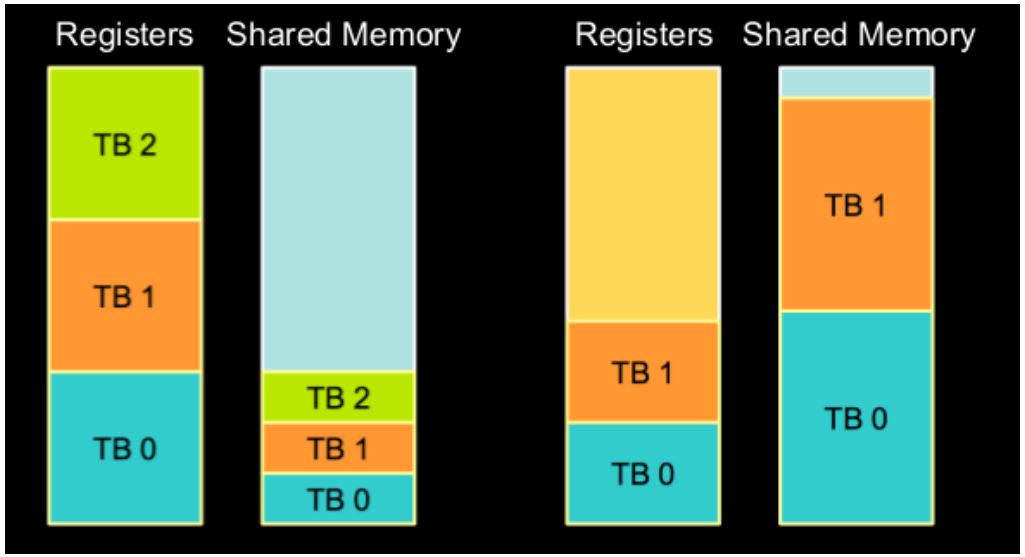
- Multiprocessors (SMs) can have many more active threads than there are CUDA Cores
- High occupancy is important, because if some threads stall, the SM can switch to others
- Pool of limited resources per SM
- Occupancy determined by
  - Register usage per thread
  - Shared memory per block

occupancy

$$\text{occupancy} = \frac{\text{active threads}}{\text{max. threads per SM}}$$

- Multiprocessors (SMs) can have many more active threads than there are CUDA Cores
- High occupancy is important, because if some threads stall, the SM can switch to others
- Pool of limited resources per SM
- Occupancy determined by
  - Register usage per thread
  - Shared memory per block

Resource limits



- Each block grabs registers and shared memory
- If one or the other is fully utilized:
  no more blocks per SM possible

Find Out Resource Usage

- Compile with nvcc option `-ptxas-options=-v`
- Per kernel registers and (static) shared memory:

```
ptxas info: Compiling entry function '_Z10add_kernelPfPKfS1_i' for 'sm_10'

ptxas info: Used 4 registers, 44 bytes smem
```

- Amount of resources per multiprocessor:

```
./deviceQuery
```

Find Out Resource Usage

- Compile with nvcc option `-ptxas-options=-v`
- Per kernel registers and (static) shared memory:

```
ptxas info: Compiling entry function '_Z10add_kernelPfPKfS1_i' for 'sm_10'

ptxas info: Used 4 registers, 44 bytes smem
```

- Amount of resources per multiprocessor:

```
./deviceQuery
```

data bandwidth and instruction throughput

data bandwidth: How much data do we process per second?

- Minimize data transfers with low bandwidth (host - device, global memory - device)
- Make use of the different types of memory
- Align your 2D array to make use of coalescing

instruction throughput: How many instructions do we execute per second?

- Trade precision for speed
- Minimize branch divergence

data bandwidth and instruction throughput

data bandwidth: How much data do we process per second?

- Minimize data transfers with low bandwidth (host - device, global memory - device)
- Make use of the different types of memory
- Align your 2D array to make use of coalescing

instruction throughput: How many instructions do we execute per second?

- Trade precision for speed
- Minimize branch divergence

data bandwidth and instruction throughput

data bandwidth: How much data do we process per second?

- Minimize data transfers with low bandwidth (host - device, global memory - device)
- Make use of the different types of memory
- Align your 2D array to make use of coalescing

instruction throughput: How many instructions do we execute per second?

- Trade precision for speed
- Minimize branch divergence

data bandwidth and instruction throughput

data bandwidth: How much data do we process per second?

- Minimize data transfers with low bandwidth (host - device, global memory - device)
- Make use of the different types of memory
- Align your 2D array to make use of coalescing

instruction throughput: How many instructions do we execute per second?

- Trade precision for speed
- Minimize branch divergence

Outline

1. Performance metrics of algorithms running on a GPU
   - occupancy
   - data bandwidth and instruction throughput
2. Maximize instruction throughput
   - branch divergence
3. Maximize memory throughput
   - pitched allocation for images
4. parallel reduction: an example of optimization

branch divergence

Reminder: All 32 threads of a warp execute the *same* instruction. *Always!*

```
1  __global__ void kernel (float *result, float *input)
2  {
3    int i = threadIdx.x + blockDim.x*blockIdx.x;
4    if (input[i]>0)
5      result[i] = 1.f;
6    else
7      result[i] = 0.f;
8  }
```

What if different paths are taken within a warp?

branch divergence

Reminder: All 32 threads of a warp execute the *same* instruction. *Always!*

```
1  __global__ void kernel (float *result, float *input)
2  {
3    int i = threadIdx.x + blockDim.x*blockIdx.x;
4    if (input[i]>0)
5      result[i] = 1.f;
6    else
7      result[i] = 0.f;
8  }
```

What if different paths are taken within a warp?

branch divergence

Reminder: All 32 threads of a warp execute the *same* instruction. *Always!*

```
1  __global__ void kernel (float *result, float *input)
2  {
3    int i = threadIdx.x + blockDim.x*blockIdx.x;
4    if (input[i]>0)
5      result[i] = 1.f;
6    else
7      result[i] = 0.f;
8  }
```

What if different paths are taken within a warp?

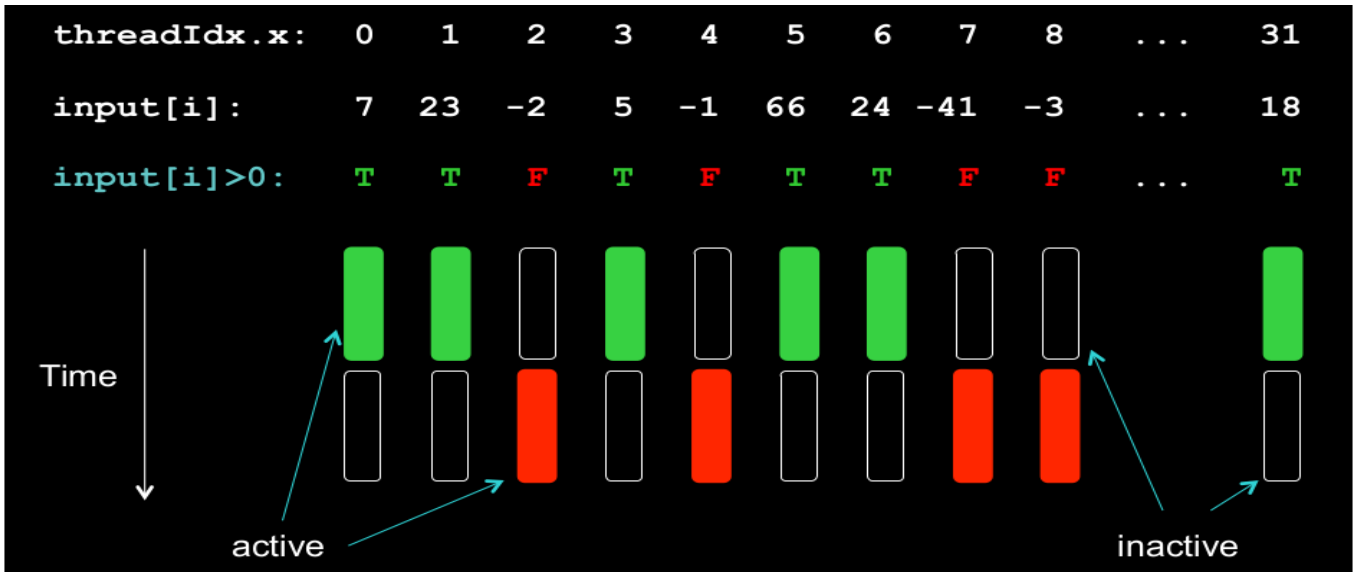
branch divergence

Reminder: All 32 threads of a warp execute the *same* instruction. *Always!*

```
1  __global__ void kernel (float *result, float *input)
2  {
3    int i = threadIdx.x + blockDim.x*blockIdx.x;
4    if (input[i]>0)
5      result[i] = 1.f;
6    else
7      result[i] = 0.f;
8  }
```

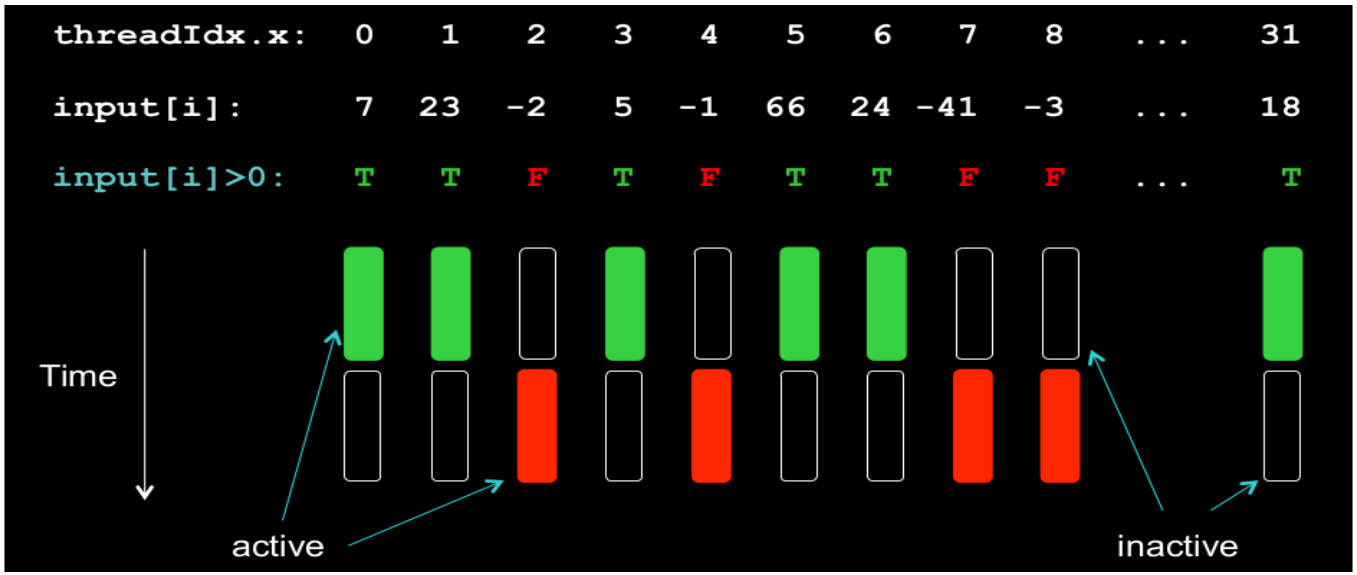
What if different paths are taken within a warp?

Serialization



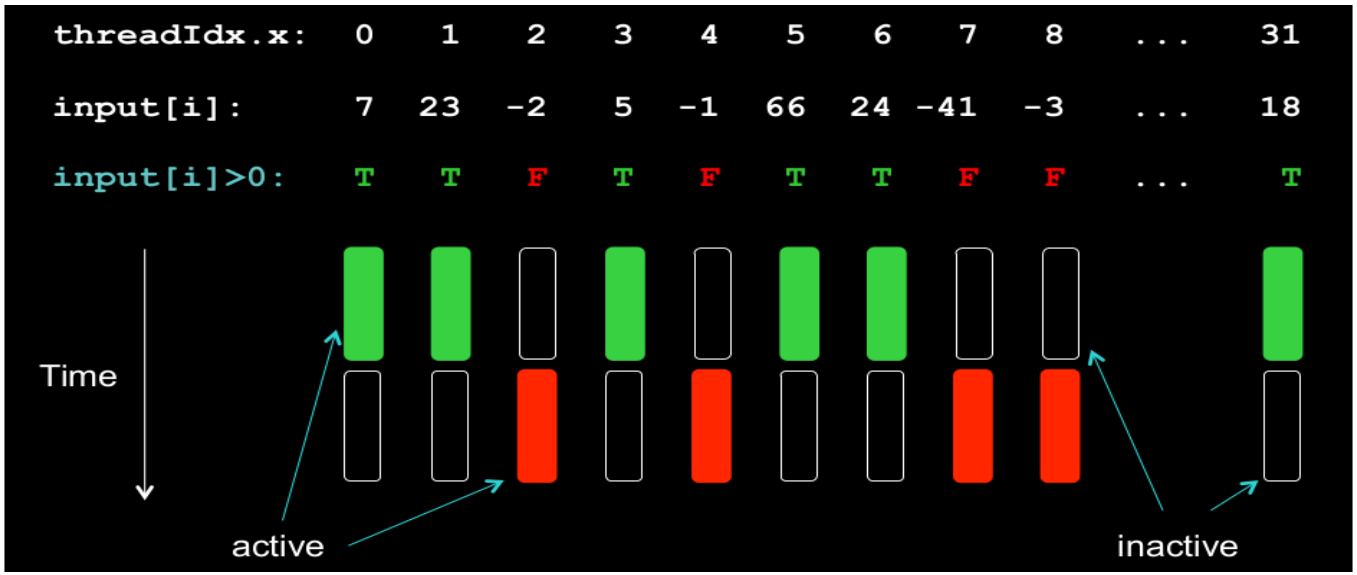
- Each path is taken by each thread.
- Threads that should take an other path are marked inactive.
- The execution of the warp is serialized.

Serialization



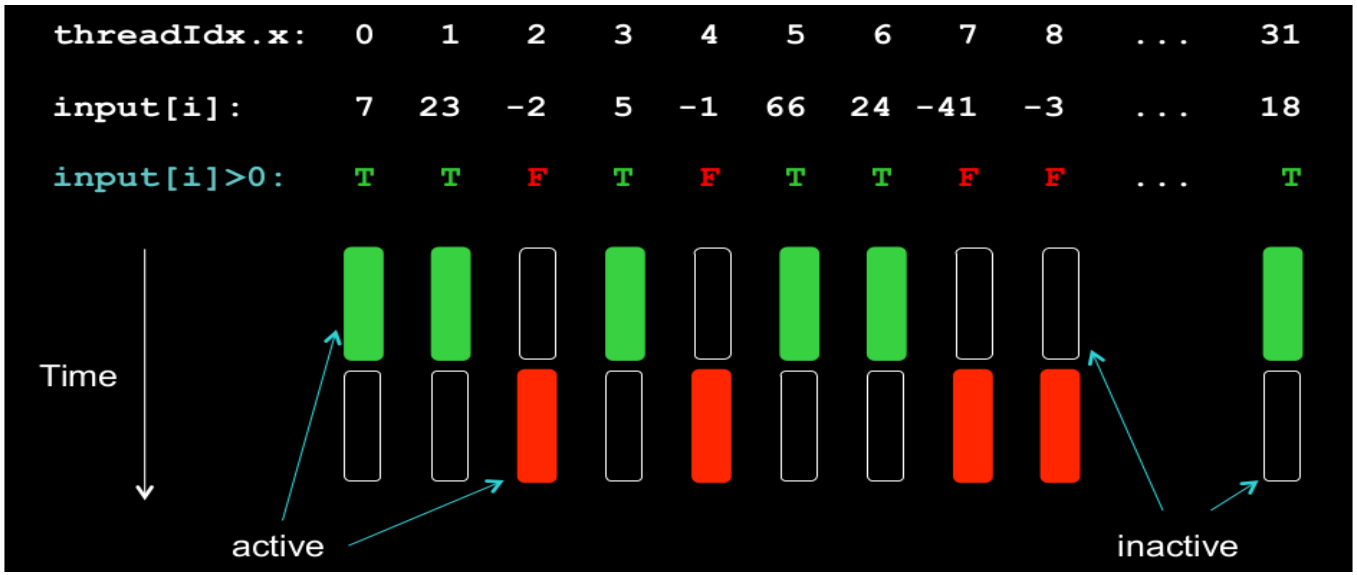
- Each path is taken by each thread.
- Threads that should take an other path are marked inactive.
- The execution of the warp is serialized.

Serialization



- Each path is taken by each thread.
- Threads that should take an other path are marked inactive.
- The execution of the warp is serialized.

Serialization



- Each path is taken by each thread.
- Threads that should take an other path are marked inactive.
- The execution of the warp is serialized.

Serialization cont.

- Also happens with the following statements: `for`, `while`, `switch`
- Worst case: 1 active thread, 31 inactive $\Rightarrow$ performance is reduced to $1/32 \approx 3\%$
- No divergence if all threads take the same path.

```
if (tid/32 == 0) {...}
```

Serialization cont.

- Also happens with the following statements: `for`, `while`, `switch`
- Worst case: 1 active thread, 31 inactive $\Rightarrow$ performance is reduced to $1/32 \approx 3\%$
- No divergence if all threads take the same path.

```
if (tid/32 == 0) {...}
```

Serialization cont.

- Also happens with the following statements: `for`, `while`, `switch`
- Worst case: 1 active thread, 31 inactive $\Rightarrow$ performance is reduced to $1/32 \approx 3\%$
- No divergence if all threads take the same path.

```
if (tid/32 == 0) {...}
```

Outline

1. Performance metrics of algorithms running on a GPU
   - occupancy
   - data bandwidth and instruction throughput
2. Maximize instruction throughput
   - branch divergence
3. Maximize memory throughput
   - pitched allocation for images
4. parallel reduction: an example of optimization

Linear allocation

- one can allocate 2d images as 1d arrays and access in a linearized way: `img[x+w*y]`
- this works, but is in general suboptimal for CUDA
- for a 6*3 float image, the addresses `&img[x+6*y]` are

48	52	56	60	64	68
24	28	32	36	40	44
0	4	8	12	16	20

- read/write accesses are fastest when the starting address of each row is a multiple of a big power of 2.
(most common: 128)

Linear allocation

- one can allocate 2d images as 1d arrays and access in a linearized way: `img[x+w*y]`
- this works, but is in general suboptimal for CUDA
- for a 6*3 float image, the addresses `&img[x+6*y]` are

48	52	56	60	64	68
24	28	32	36	40	44
0	4	8	12	16	20

- read/write accesses are fastest when the starting address of each row is a multiple of a big power of 2.
(most common: 128)

Linear allocation

- one can allocate 2d images as 1d arrays and access in a linearized way: `img[x+w*y]`
- this works, but is in general suboptimal for CUDA
- for a 6*3 float image, the addresses `&img[x+6*y]` are

48	52	56	60	64	68
24	28	32	36	40	44
0	4	8	12	16	20

- read/write accesses are fastest when the starting address of each row is a multiple of a big power of 2.
(most common: 128)

Linear allocation

- one can allocate 2d images as 1d arrays and access in a linearized way: `img[x+w*y]`
- this works, but is in general suboptimal for CUDA
- for a 6*3 float image, the addresses `&img[x+6*y]` are

48	52	56	60	64	68
24	28	32	36	40	44
0	4	8	12	16	20

- read/write accesses are fastest when the starting address of each row is a multiple of a big power of 2.
(most common: 128)

pitched allocation for images cont.

- the total new width in bytes is called **pitch**

64	68	72	76	80	84	88	92
32	36	40	44	48	52	56	60
0	4	8	12	16	20	24	28

- here: pitch = 32 bytes (=8*sizeof(float))
- in general pitch != multiple of element size
  - float3 array

adding padding bytes at the end of each row resolves this

pitched allocation for images cont.

- on host:

```
1  float *d_a;
2  size_t pitch;
3  cudaMallocPitch(&d_a, &pitch, w*sizeof(float), h);
```

- in kernel:

```
1  float value =
2  *((float*)( (char*)a + x*sizeof(float) + pitch*y) );
```

- Copying: `cudaMemcpy2D(...)`
- For 3D-Data: `cudaMalloc3D(...)`

pitched allocation for images cont.

- on host:

```
1  float *d_a;
2  size_t pitch;
3  cudaMallocPitch(&d_a, &pitch, w*sizeof(float), h);
```

- in kernel:

```
1  float value =
2  *((float*)( (char*)a + x*sizeof(float) + pitch*y) );
```

- Copying: `cudaMemcpy2D(...)`
- For 3D-Data: `cudaMalloc3D(...)`

pitched allocation for images cont.

- on host:

```
1  float *d_a;
2  size_t pitch;
3  cudaMallocPitch(&d_a, &pitch, w*sizeof(float), h);
```

- in kernel:

```
1  float value =
2  *((float*)( (char*)a + x*sizeof(float) + pitch*y) );
```

- Copying: `cudaMemcpy2D(...)`
- For 3D-Data: `cudaMalloc3D(...)`

Outline

1. Performance metrics of algorithms running on a GPU
   - occupancy
   - data bandwidth and instruction throughput
2. Maximize instruction throughput
   - branch divergence
3. Maximize memory throughput
   - pitched allocation for images
4. parallel reduction: an example of optimization

- Want to process very large arrays
- Keep all the SM busy
- each block reduces a part of the array but how do we communicate the partial results in an efficient way?

- Want to process very large arrays
- Keep all the SM busy
- each block reduces a part of the array but how do we communicate the partial results in an efficient way?

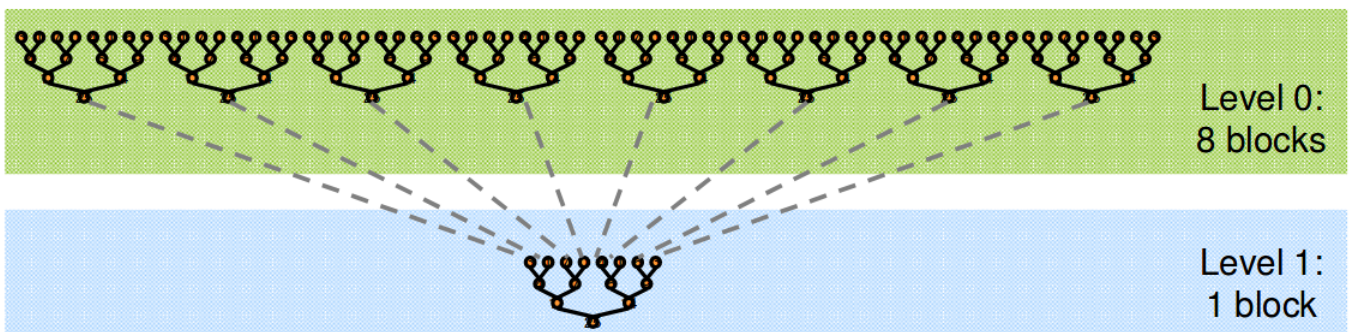
- Want to process very large arrays
- Keep all the SM busy
- each block reduces a part of the array but how do we communicate the partial results in an efficient way?

- CUDA does not have global synchronization
- solution: decompose into multiple kernels and use launch as synchronization point
- two different metrics of performance: bandwidth and GFLOP/s
- Reductions have low arithmetric intensity $\Rightarrow$ bandwidth is the proper metric

- CUDA does not have global synchronization
- solution: decompose into multiple kernels and use launch as synchronization point
- two different metrics of performance: bandwidth and GFLOP/s
- Reductions have low arithmetric intensity $\Rightarrow$ bandwidth is the proper metric

- CUDA does not have global synchronization
- solution: decompose into multiple kernels and use launch as synchronization point
- two different metrics of performance: bandwidth and GFLOP/s
- Reductions have low arithmetic intensity $\Rightarrow$ bandwidth is the proper metric

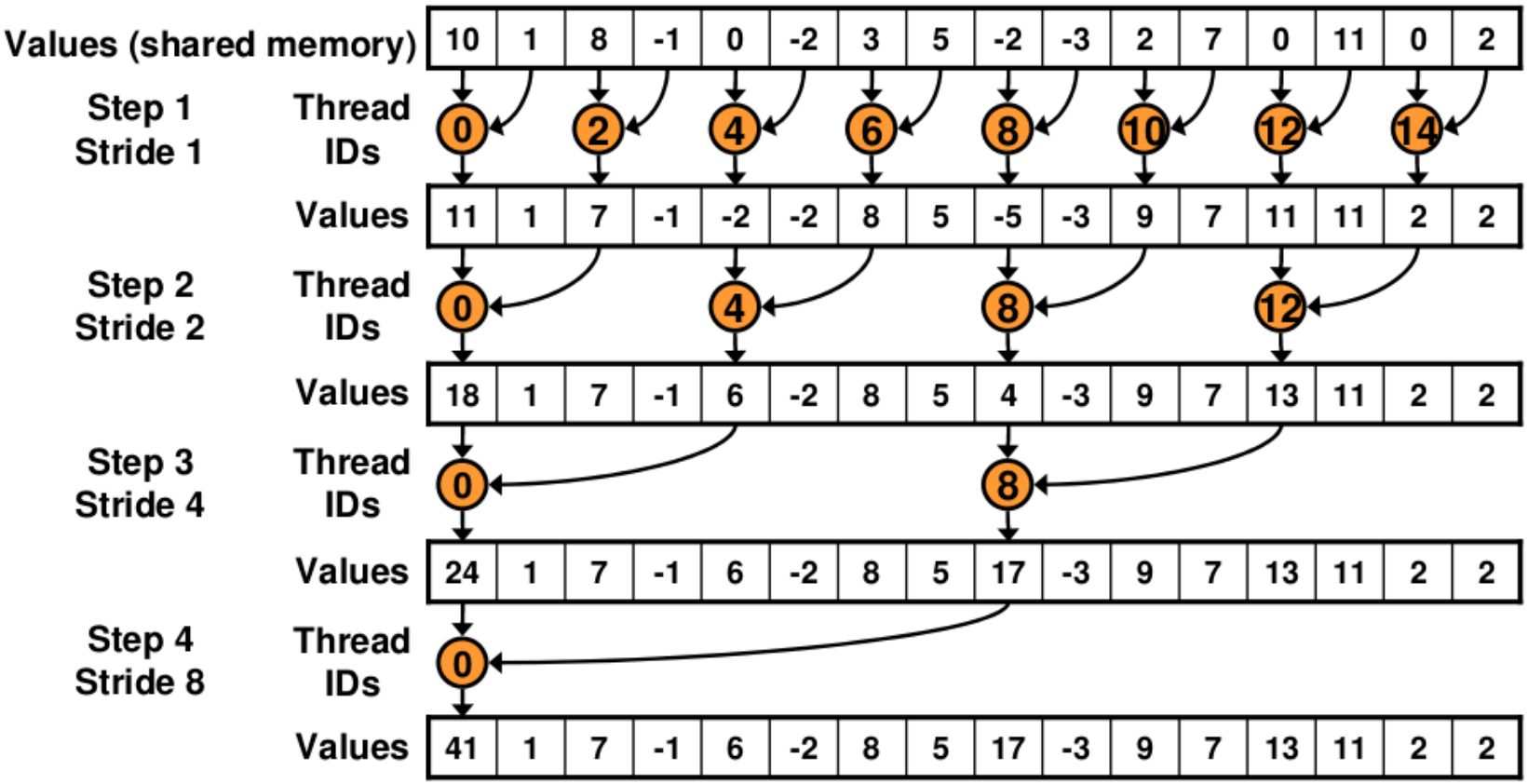
- CUDA does not have global synchronization
- solution: decompose into multiple kernels and use launch as synchronization point
- two different metrics of performance: bandwidth and GFLOP/s
- Reductions have low arithmetic intensity $\Rightarrow$ bandwidth is the proper metric



A first implementation

```
1  __global__ void reduce0(int *g_idata, int *g_odata) {
2  extern __shared__ int sdata[];
3
4  // each thread loads one element from global to shared mem
5  unsigned int tid = threadIdx.x;
6  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
7  sdata[tid] = g_idata[i];
8  __syncthreads();
9
10 // do reduction in shared mem
11 for(unsigned int s=1; s < blockDim.x; s *= 2) {
12   if (tid % (2*s) == 0) {
13     sdata[tid] += sdata[tid + s];
14   }
15   __syncthreads();
16 }
17
18 // write result for this block to global mem
19 if (tid == 0) g_odata[blockIdx.x] = sdata[0];
20 }
```

A first implementation



how can we accelerate the code?

hint: branch divergence

how can we accelerate the code?

hint: branch divergence

Just replace divergent branch in inner loop:

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

With strided index and non-divergent branch:

```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    int index = 2 * s * tid;

    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

- This is already better, but still we can improve a lot.

- Let's take a closer look at the shared memory:
  - On modern GPUs the shared memory is divided into 32 banks.
  - Adresses in different banks can be read at the same time.
  - If different threads within a warp want to read different adresses from a single bank, the accesses are executed in serial.
  - Successive 32-bit words are assigned to successive banks
  - This is commonly refered to as a bank conflict

- This is already better, but still we can improve a lot.
- Let's take a closer look at the shared memory:
  - On modern GPUs the shared memory is divided into 32 banks.
  - Adresses in different banks can be read at the same time.
  - If different threads within a warp want to read different adresses from a single bank, the accesses are executed in serial.
  - Successive 32-bit words are assigned to successive banks
  - This is commonly refered to as a bank conflict

- This is already better, but still we can improve a lot.
- Let's take a closer look at the shared memory:
  - On modern GPUs the shared memory is divided into 32 banks.
  - Adresses in different banks can be read at the same time.
  - If different threads within a warp want to read different adresses from a single bank, the accesses are executed in serial.
  - Successive 32-bit words are assigned to successive banks
  - This is commonly refered to as a bank conflict

- This is already better, but still we can improve a lot.

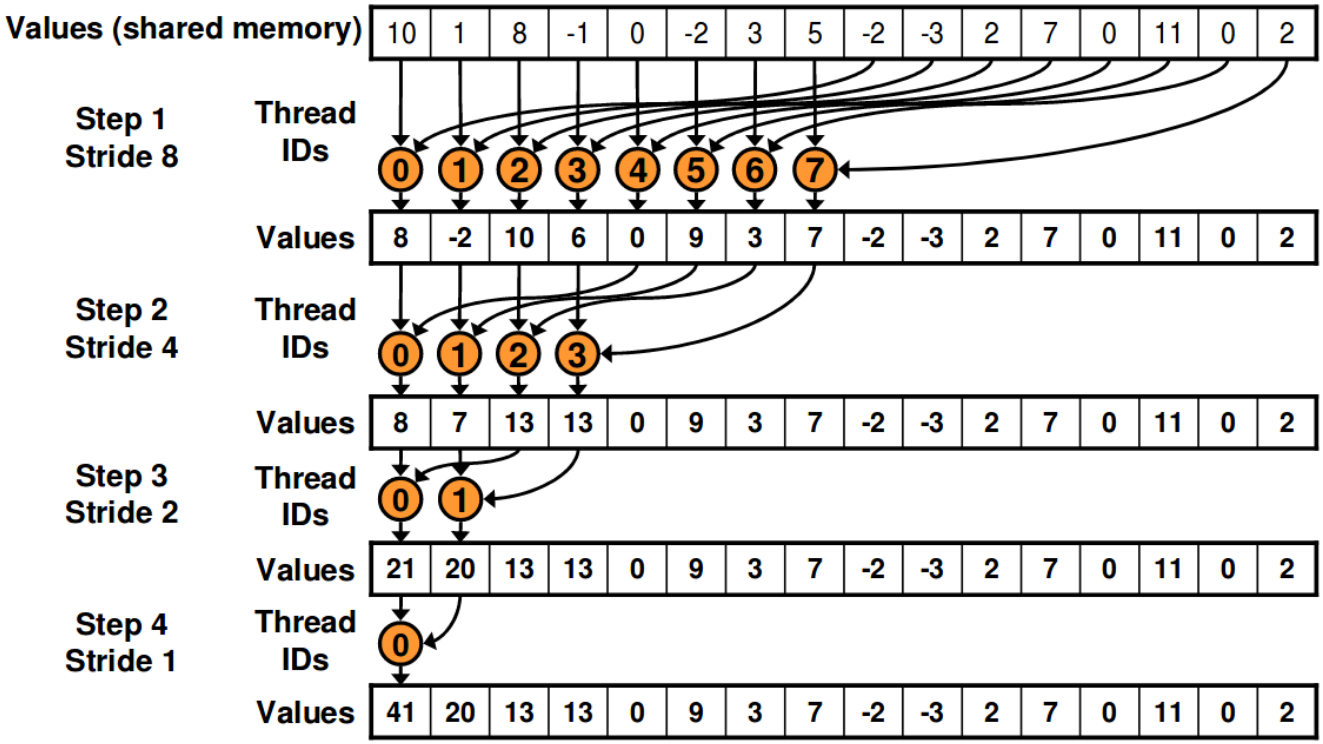
- Let's take a closer look at the shared memory:
  - On modern GPUs the shared memory is divided into 32 banks.
  - Adresses in different banks can be read at the same time.
  - If different threads within a warp want to read different adresses from a single bank, the accesses are executed in serial.
  - Successive 32-bit words are assigned to successive banks
  - This is commonly refered to as a bank conflict

- This is already better, but still we can improve a lot.

- Let's take a closer look at the shared memory:
  - On modern GPUs the shared memory is divided into 32 banks.
  - Adresses in different banks can be read at the same time.
  - If different threads within a warp want to read different adresses from a single bank, the accesses are executed in serial.
  - Successive 32-bit words are assigned to successive banks
  - This is commonly refered to as a bank conflict

- This is already better, but still we can improve a lot.

- Let's take a closer look at the shared memory:
  - On modern GPUs the shared memory is divided into 32 banks.
  - Adresses in different banks can be read at the same time.
  - If different threads within a warp want to read different adresses from a single bank, the accesses are executed in serial.
  - Successive 32-bit words are assigned to successive banks
  - This is commonly refered to as a bank conflict


Values (shared memory)
10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2
Step 1
Stride 8
Thread IDs
0 1 2 3 4 5 6 7
Values
8 -2 10 6 0 9 3 7 -2 -3 2 7 0 11 0 2
Step 2
Stride 4
Thread IDs
0 1 2 3
Values
8 7 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2
Step 3
Stride 2
Thread IDs
0 1
Values
21 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2
Step 4
Stride 1
Thread IDs
0
Values
41 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2

After a few additional optimizations, this is the final speed up:

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x
Kernel 6: completely unrolled	0.381 ms	43.996 GB/s	1.41x	21.16x
Kernel 7: multiple elements per thread	0.268 ms	62.671 GB/s	1.42x	30.04x

for the full details see:
http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf