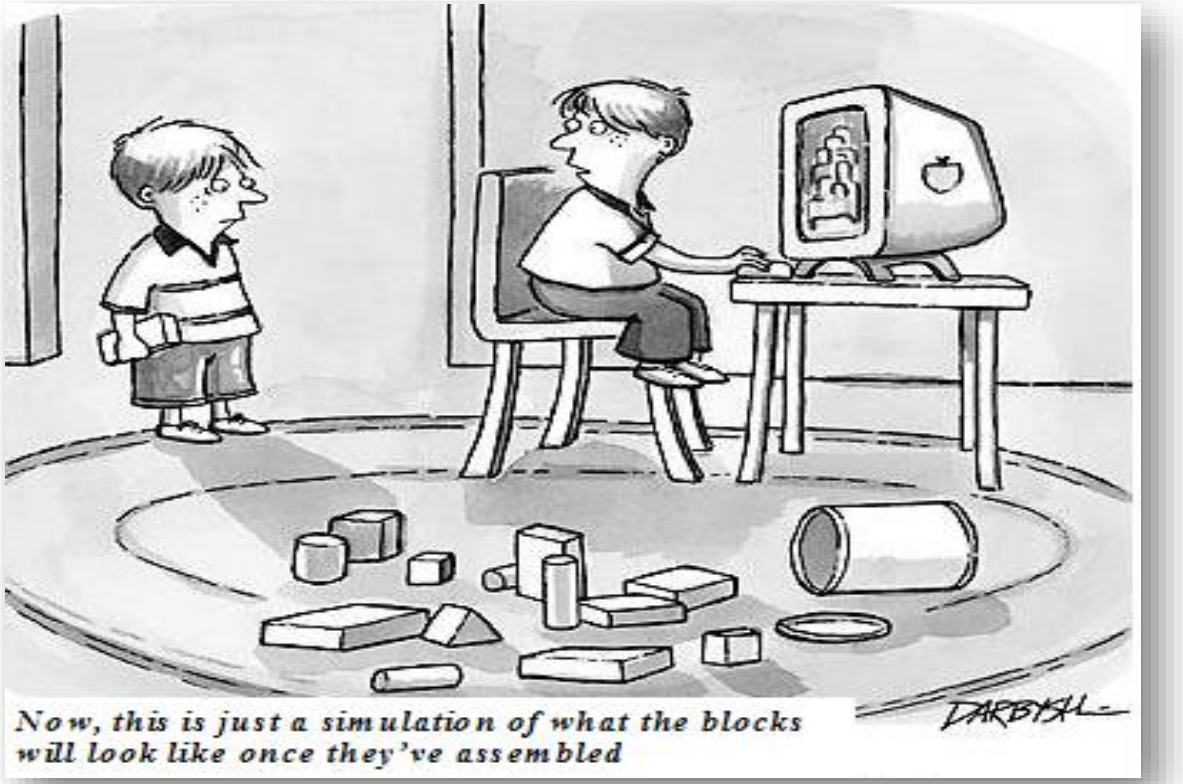
ECE569
Module 19



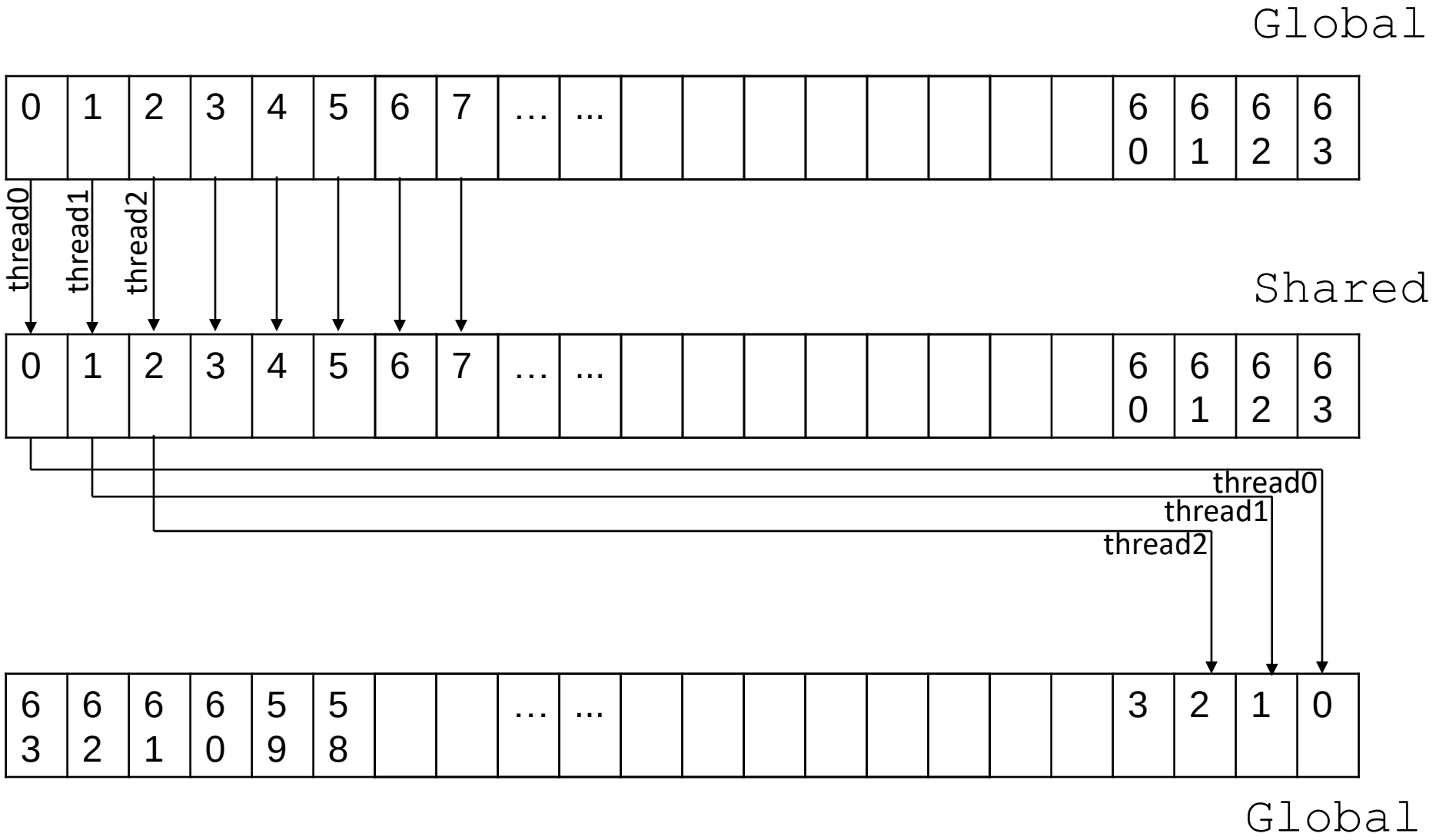
- Global Memory, Shared Memory, Coalesced Memory Access
- Shared Memory: Static vs. Dynamic

Code Review

```
int main(void)
{
  const int n = 64;
  int a[n], d[n];
  for (int i = 0; i < n; i++) {
    a[i] = i;
    d[i] = 0;
  }
  int *d_d;
  cudaMalloc(&d_d, n * sizeof(int));

  // run version with static shared memory
  // code available on D2L->Content->Demo
  cudaMemcpy(d_d,a,n*sizeof(int),cudaMemcpyHostToDevice);
  staticReverse<<<1,n>>>(d_d, n);
  cudaMemcpy(d,d_d,n*sizeof(int),cudaMemcpyDeviceToHost);
  for (int i = 0; i < n; i++)
    printf("%d %d\n", a[i], d[i]);
```

`staticReverse` reverses the data in a 64-element array using shared memory.

```
#include <stdio.h>
// Static shared memory version
__global__ void baselineReverse(int *d, int n)
```

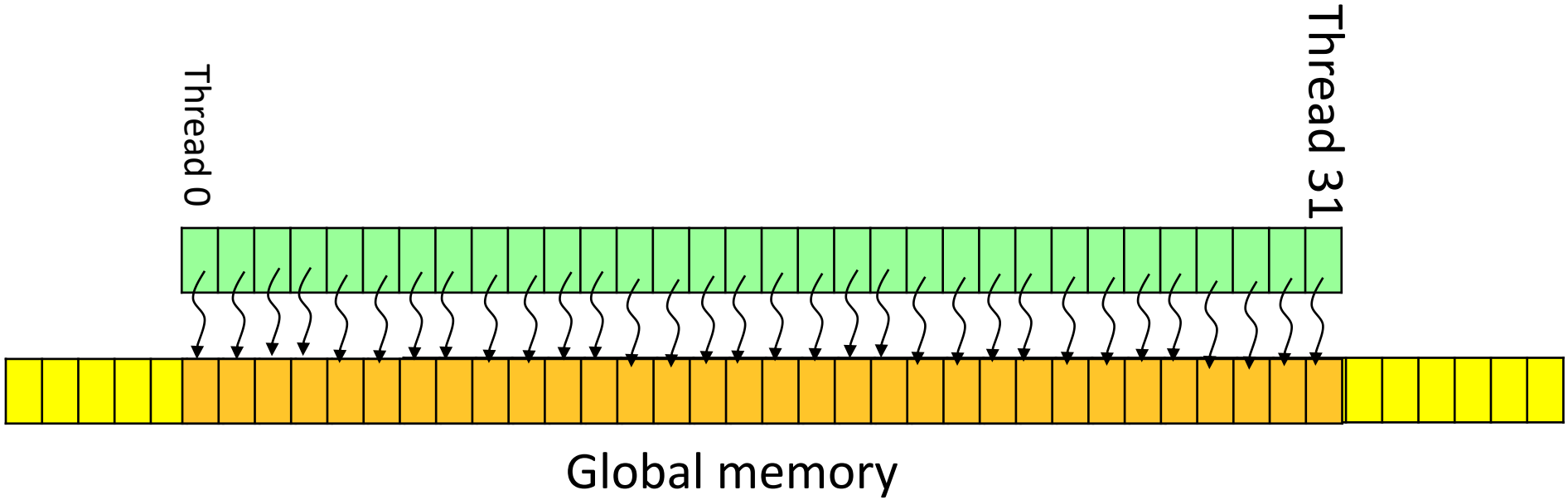


```
#include <stdio.h>
// Static shared memory version
__global__ void baselineReverse(int *d, int n)
{









}
```

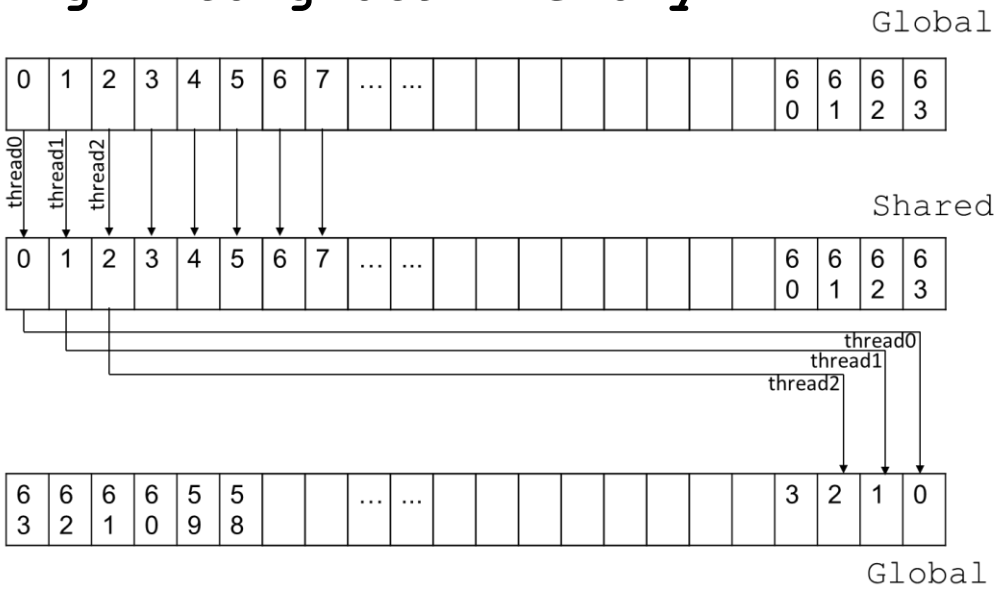
```
#include <stdio.h>
// Static shared memory version
__global__ void baselineReverse(int *d, int n)
{
// shared memory array size is known at compile time
    __shared__  int s[64];

    int id = threadIdx.x;
    s[id] = d[id];
    __syncthreads();
    d[n-id-1] = s[id];
    __syncthreads();


}

// Exercise: Download the code available on
// D2L->Content->Demo and
// run it on the GPU. Collect timing info.
```

Coalesced Memory Access

- **Global memory accesses by threads**
  - GPU is most efficient when threads read/write contiguous memory locations.

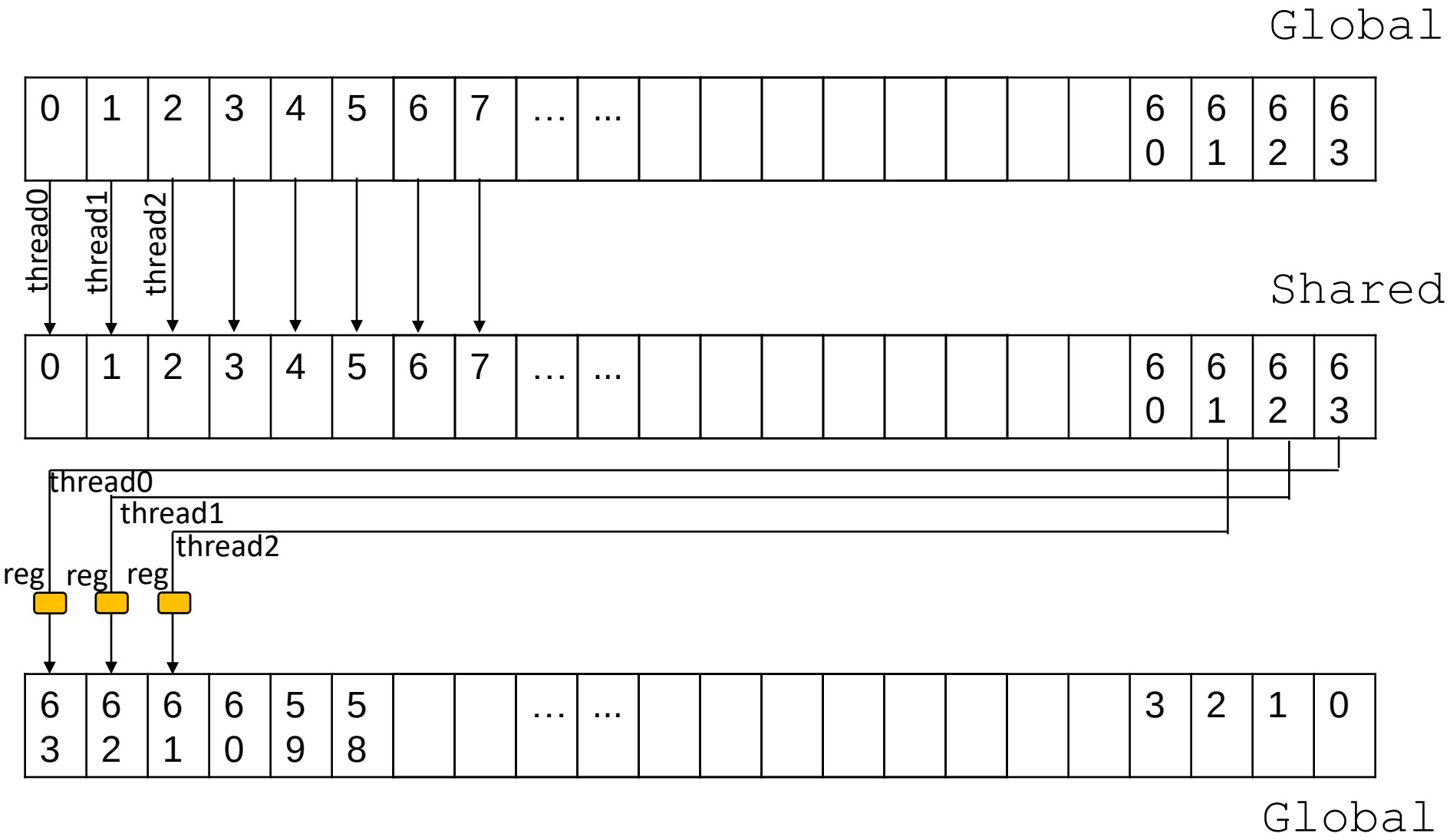


```
#include <stdio.h>
// Static shared memory version
__global__ void baselineReverse(int *d, int n)
{
// shared memory array size is known at compile time
    __shared__  int s[64];
    int id = threadIdx.x;
    s[id] = d[id]; //reading from global memory
    __syncthreads();
    d[n-id-1] = s[id];//writing into global memory
    __syncthreads();
}
```

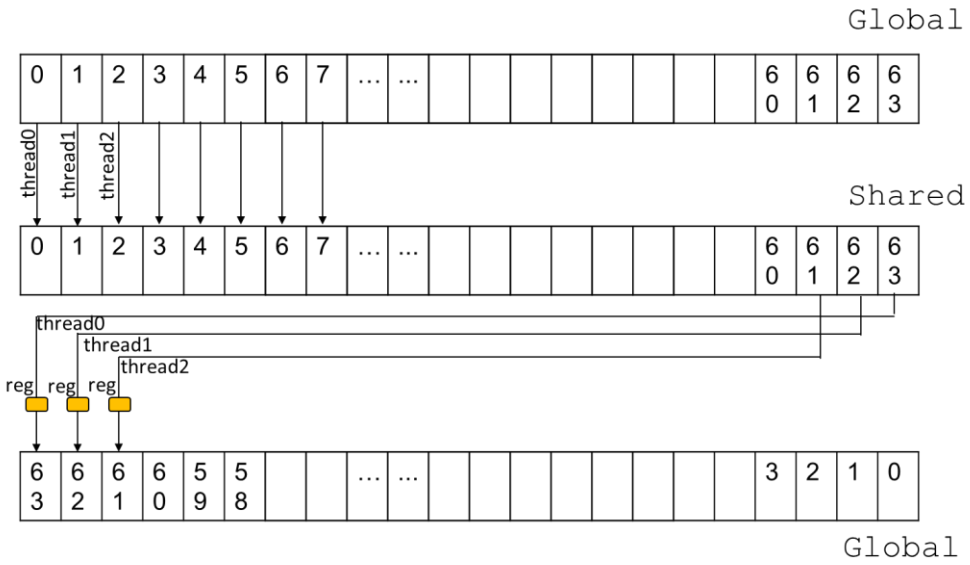


Do we have coalesced memory access to global memory in this code?

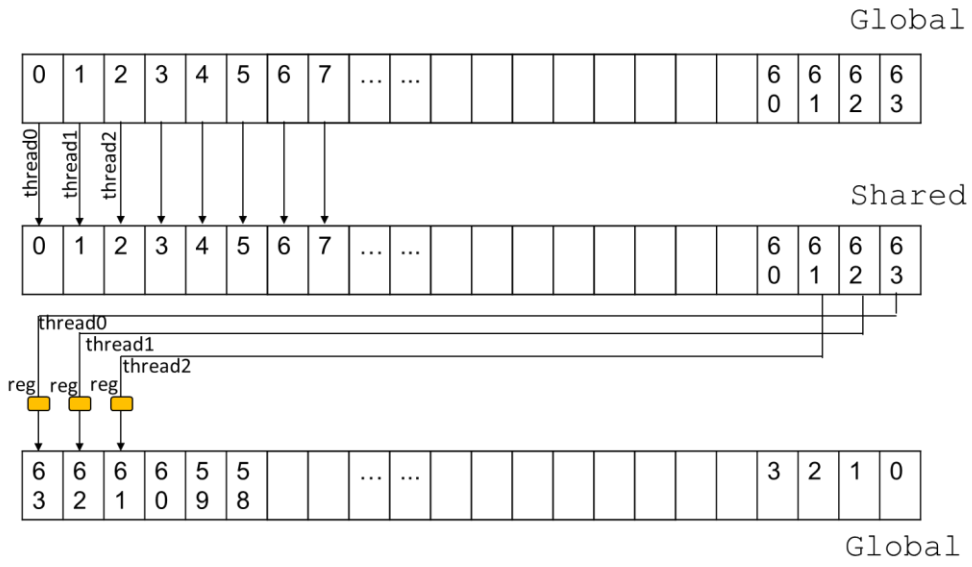
```
#include <stdio.h>
// Static shared memory version
__global__ void baselineReverse(int *d, int n)
```



```
#include <stdio.h>
// Static shared memory version
__global__ void baselineReverse(int *d, int n)
{
// shared memory array size is known at compile time
    __shared__  int s[64];
    int id = threadIdx.x;


    s[id] = d[id];              //reading from global memory
    __syncthreads();
    d[n-id-1] = s[id];          //writing into global memory
    __syncthreads();


}
```



```
#include <stdio.h>

__global__ void coalescedReverse(int *d, int n)
{
  __shared__ int s[64];
  int id = threadIdx.x;
  int tr = n-id-1;
  int temp;

  s[id] = d[id];
  __syncthreads();
  temp = s[tr];
  __syncthreads();
  d[id] = temp;
  __syncthreads();

}
```



This code could be improved, meaning we can eliminate some of the synchthreads. Do you see how?

```
#include <stdio.h>
__global__ void coalescedReverse(int *d, int n)
{
  __shared__ int s[64];
  int id = threadIdx.x;
  int tr = n-id-1;
  s[id] = d[id];
  __syncthreads();
  d[id] = s[tr];
}
```

Exercise: Copy this kernel code into reverse.cu line 47. File is available on D2L->Content->Demo and run it on the GPU. Compare the execution time performance with the baseline implementation.

- **global memory coalescing** for reads and writes on d[id]
  - global memory always accessed through the linear, aligned index id.
- The reversed index tr only used to access shared memory,
  - does not have the sequential access restrictions of global memory for optimal performance.
  - The only performance issue with shared memory is **bank conflicts**, which we will discuss later.

```
int main(void)
{
  const int n = 64;
  int a[n], r[n], d[n];
  for (int i = 0; i < n; i++) {
    a[i] = i;
    r[i] = n-i-1;
    d[i] = 0;
  }
  int *d_d;
  cudaMalloc(&d_d, n * sizeof(int));

  // run version with static shared memory
  cudaMemcpy(d_d,a,n*sizeof(int),cudaMemcpyHostToDevice);
  baselineReverse<<<1,n>>>(d_d, n);
  cudaMemcpy(d,d_d,n*sizeof(int),cudaMemcpyDeviceToHost);
  for (int i = 0; i < n; i++)
    if (d[i] != r[i])
      printf("Error: d[%d]!=r[%d] (%d, %d)\n",
             i, i, d[i], r[i]);
```

What if we don't know the amount of shared memory at compile time?

```
int main(void)
{
  const int n = 64;
  int a[n], r[n], d[n];
  for (int i = 0; i < n; i++) {
    a[i] = i;
    r[i] = n-i-1;
    d[i] = 0;
  }
  int *d_d;
  cudaMalloc(&d_d, n * sizeof(int));
  // run dynamic shared memory version
  // shared memory allocation size per thread block
  // must be specified (in bytes) using an optional
  // third execution configuration parameter
  cudaMemcpy(d_d,a,n*sizeof(int),cudaMemcpyHostToDevice);
  dynamicReverse<<<1,n,n*sizeof(int)>>>(d_d, n);
  cudaMemcpy(d,d_d,n*sizeof(int),cudaMemcpyDeviceToHost);
  for (int i = 0; i < n; i++)
    if (d[i] != r[i])
      printf("Error: d[%d]!=r[%d] (%d, %d)\n",
             i, i, d[i], r[i]);
```

```
__global__ void coalescedReverse(int *d, int n){
  __shared__ int s[64];
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}

__global__ void dynamicReverse(int *d, int n){
  extern __shared__ int s[];
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}
```

Exercise: Copy this kernel code into reverse.cu line 53. Compare the execution time performance with the coalesced version.

Performance Comparison

- **You will observe output similar to this:**
  - Even for a 64 element array we observe 42x speedup with the coalesced memory access!!

Kernel Version	Time (ms)	Speedup
Baseline static	0.036832	
Coalesced static	0.008704	42X
Coalesced dynamic	0.006944	53X

What if you need multiple dynamically sized arrays in a single kernel?

For example:

```
// nI ints
// nF floats
// nC chars
```

What if you need multiple dynamically sized arrays in a single kernel?

```
// In the kernel launch, specify the total shared memory
needed in bytes
```

For example:

```
// nI ints
// nF floats
// nC chars


myKernel<<<gridSize, blockSize,
nI*sizeof(int)+nF*sizeof(float)+nC*sizeof(char)>>>(...);
```

What if you need multiple dynamically sized arrays in a single kernel?

You must declare a single extern unsized array as before, and use pointers into it to divide it into multiple arrays

```
extern __shared__ int s[];
int *integerData = ???;                          // nI ints
```

What if you need multiple dynamically sized arrays in a single kernel?

You must declare a single extern unsized array as before, and use pointers into it to divide it into multiple arrays

```
extern __shared__ int s[];
int *integerData = s;                          // nI ints

float *floatData =????;                        // nF floats
```

What if you need multiple dynamically sized arrays in a single kernel?

You must declare a single extern unsized array as before, and use pointers into it to divide it into multiple arrays

```
extern __shared__ int s[];
int *integerData = s;                          // nI ints

float *floatData = (float*)&integerData[nI]; // nF floats

char *charData = ???;                          // nC chars
```

What if you need multiple dynamically sized arrays in a single kernel?

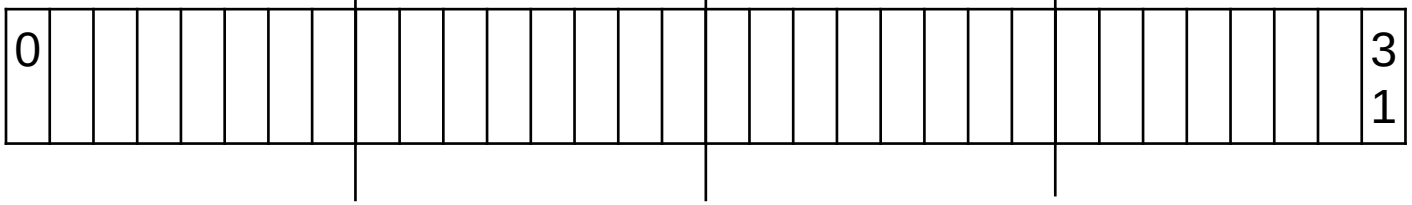
You must declare a single extern unsized array as before, and use pointers into it to divide it into multiple arrays

```
extern __shared__ int s[];
int *integerData = s;                          // nI ints

float *floatData = (float*)&integerData[nI]; // nF floats

char *charData = (char*)&floatData[nF];       // nC chars
```

Minimize memory access time: Coalesced Memory Read and Write

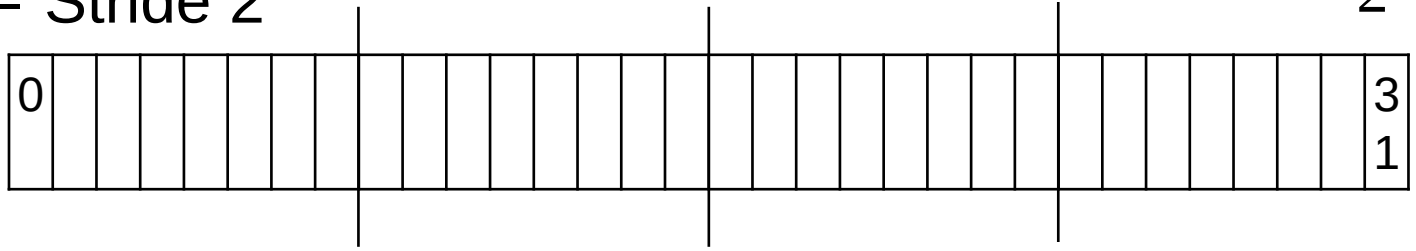
- Assume memory serves 8 elements. How many memory transactions for 8 threads accessing data for each stride?
  - Stride 0

threadIdx.x



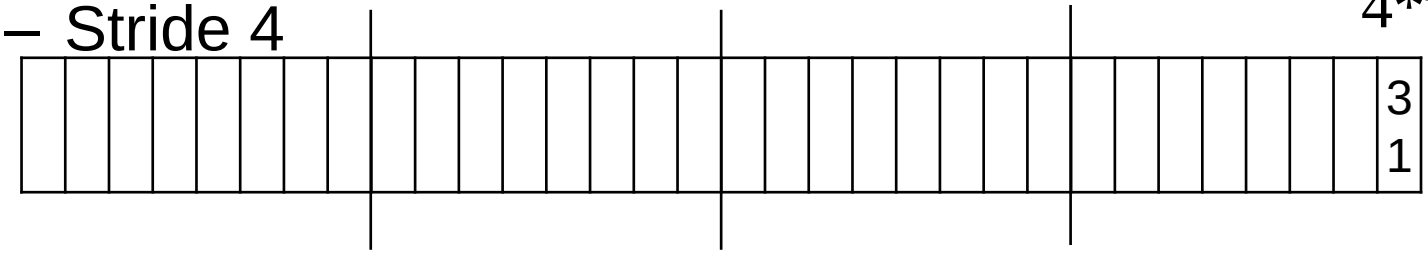
  - Stride 2

2*threadIdx.x



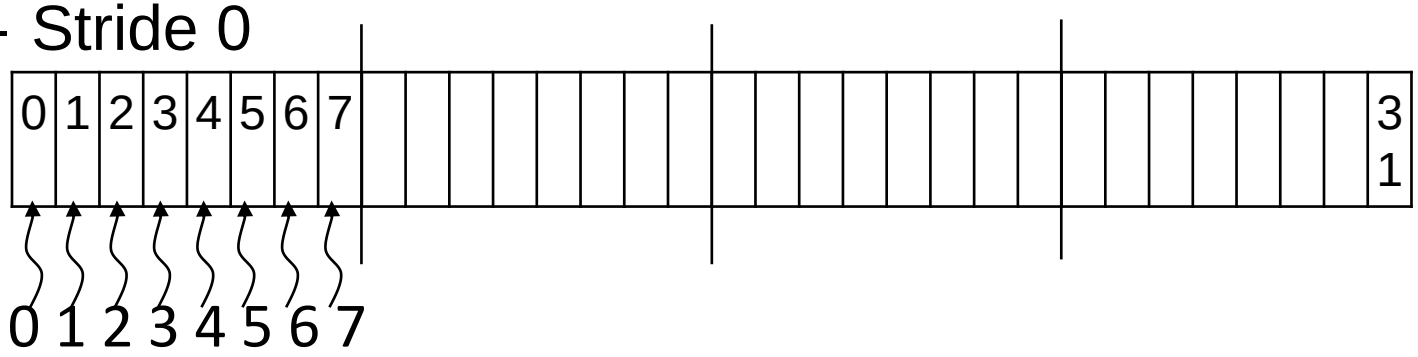
  - Stride 4

4*threadIdx.x

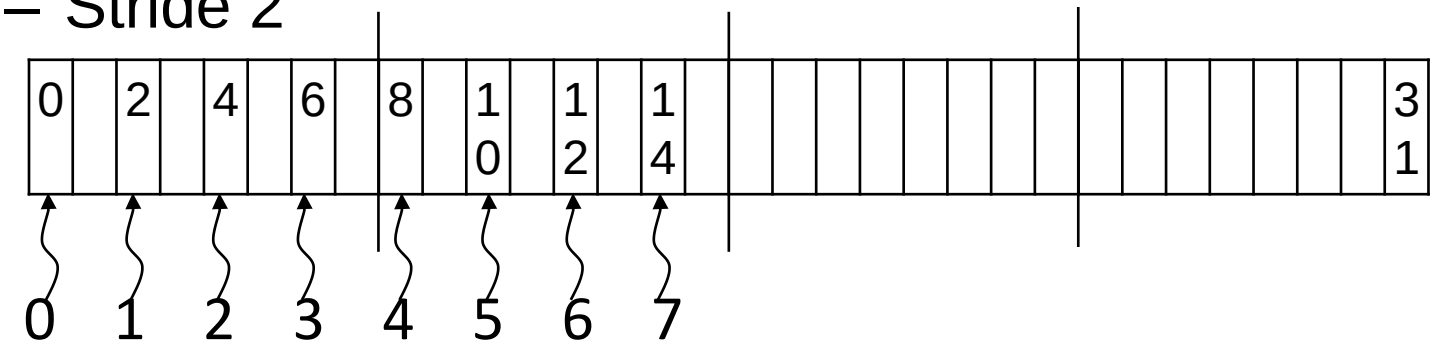


Minimize memory access time: Coalesced Memory Read and Write

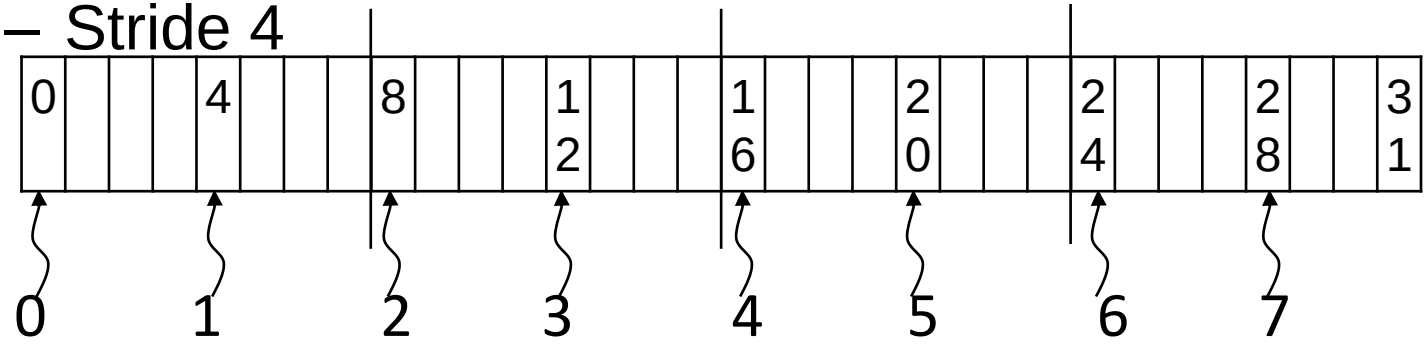
- Assume memory serves 8 elements. How many memory transactions for 8 threads accessing data for each stride?
  - Stride 0



  - Stride 2



  - Stride 4



Exercise: Which statements have coalesced access pattern?

```
__ global void foo(int * g) {
  float a = 3.14;
  int i = threadIdx.x;
  g[i] = a;                          ❑1
  g[i*2] = a;                        ❑2
  a = g[i];                          ❑3
  a = a*g[BLOCK_WIDTH/2+i];          ❑4
  g[i]= a*g[BLOCK_WIDTH/2+i];        ❑5
  g[Block_WIDTH-1-i] = a;            ❑6
}
```

Exercise: Which statements have coalesced access pattern?

```
__ global void foo(int * g) {
  float a = 3.14;
  int i = threadIdx.x;
  g[i] = a;                          ✓ 1
  g[i*2] = a;                        ❑ 2
  a = g[i];                          ✓ 3
  a = a*g[BLOCK_WIDTH/2+i];          ✓ 4
  g[i]= a*g[BLOCK_WIDTH/2+i];        ✓ 5
  g[Block_WIDTH-1-i] = a;            ❑ 6
}
```

Next

- **Thread divergence**