Advanced Topics in Numerical Analysis: High Performance Computing

Cross-listed as MATH-GA.2012-001 and CSCI-GA 2945.001

Benjamin Peherstorfer
Courant Institute, NYU
pehersto@cims.nyu.edu

Spring 2020, Monday, 1:25–3:15PM, WWH #512

April 6, 2020

Slightly adapted from Georg Stadler's lectures.

Today

Last lecture

- First steps with GPUs and CUDA programming

Today

- Global memory
- Shared memory
- Reduction

Announcements and upcoming

- Homework 4 due April 20
- If you haven't submitted your final project proposal, then do so today

Review of Last Class

CPU

- fewer cores $\sim O(10)$
- smaller vector lengths (2-SSE, 4-AVX)
- large amounts of cache (L1, L2, L3)

GPU

- many cores $\sim O(10^5)$
- wider vector lengths (warp size $=$ 32)
- smaller cache (L1/programmer managed buffer, L2 on more recent GPUs)

Review of Last Class

Computing on GPU:

- Device (GPU) is slave to the host (CPU).
- Workflow for computing on GPU:
  1. copy data from host to device,
  2. launch GPU kernel to compute result on device,
  3. copy result from device to host.
- Host - device interconnect (PCI Express or NVLink[*]) is slow (O(10-50) GB/s).
- Computing on GPUs useful only for compute-intensive tasks (when overhead of data transfer is small compared to compute time).

[*] Proprietary Nvidia system

Review of Last Class

GPU Architecture: consists of,

- main memory (typically DRAM).
- PCI Express or other interconnection to host (CPU) DRAM and other GPUs.
- several Streaming Multiprocessors (SM)
    - with shared L2 cache.

Each Streaming Multiprocessor,

- 32 scalar double-precision cores (each executing the same instruction, Single Instruction Multiple Data).
- can execute up to 1024 scalar threads (or 32-warps) simultaneously (pipelined, hyper-threading to hide instruction latency).
- L1 cache / shared-memory: shared by all threads in the thread-block.

Review of Last Class

GPU Threads

- parallelism $\sim O(10^5)$,   orders of magnitude more than CPUs
- design philosophy on CPU vs GPU,
  - On CPU: #-of-threads $\sim$ #-of-cores
  - On GPU: #-of-threads $\sim O(N)$ (problem size)
- thread hierarchy
  - **threads** partitioned into **blocks**
  - **blocks** arranged into a **grid**

`blockIdx.x`	0				1				2			
`threadIdx.x`	0	1	2	3	0	1	2	3	0	1	2	3
A_i	A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}	A_{11}

```
idx = threadIdx.x + blockDim.x * blockIdx.x
```

```
dim3 GridDim(3), BlockDim(4);
mykernel<<<GridDim, BlockDim>>>();
```

Review of Last Class

2D Grid:

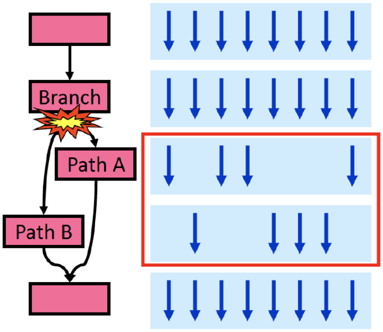
		0			1		
		0	1	2	0	1	2
0	0	$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{0,4}$	$A_{0,5}$
	1	$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{1,4}$	$A_{1,5}$
	2	$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{2,4}$	$A_{2,5}$
1	0	$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$	$A_{3,4}$	$A_{3,5}$
	1	$A_{4,0}$	$A_{4,1}$	$A_{4,2}$	$A_{4,3}$	$A_{4,4}$	$A_{4,5}$
	2	$A_{5,0}$	$A_{5,1}$	$A_{5,2}$	$A_{5,3}$	$A_{5,4}$	$A_{5,5}$

$A_{i,j}$

```
 i = threadIdx.x + blockDim.x * blockIdx.x
 j = threadIdx.y + blockDim.y * blockIdx.y
```

```
dim3 GridDim(2,2), BlockDim(3,3);
mykernel<<<GridDim, BlockDim>>>();
```

- **Performance will be degraded because of warp divergence.**



50% Performance Loss

Memory Hierarchy

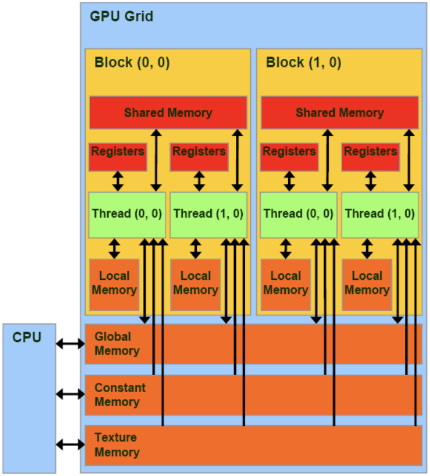
Local memory: Local to each thread (large arrays or register spilling), reside in GPU main-memory.

Shared memory: accessible by all threads in the same thread-block. Eg: `__shared__ A[100];`

Global memory: accessible by all threads, resides in DRAM. Eg: `cudaMalloc(&A,10*sizeof(double))`

Constant memory: read-only, resides in DRAM and constant cache.

Texture memory: read-only, resides in DRAM and texture cached.



Global memory

Global memory

- Resides in device DRAM
- For transfers between the host and device as well as for the data input to and output from kernels.
- "Global" because can be accessed and modified from both the host and the device.
- Global memory are cached on the chip (except for very old hardware)

```
__device__ int globalArray[256];

void foo()
{
    ...
    int *myDeviceMemory = 0;
    cudaError_t result = cudaMalloc(&myDeviceMemory, 256 *
        sizeof(int));
    ...
}
```

Global memory can be declared

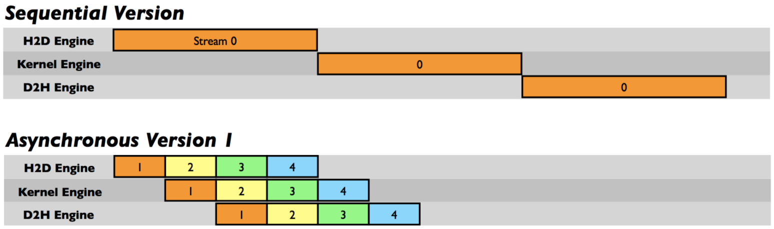
- Static: Global (variable) scope using the __device__ declaration specifier
- Dynamic: Using cudaMalloc() and assigned to a regular C pointer variable

Asynchronous computations

The asynchronous behavior of kernel launches from the host's perspective makes overlapping device and host computation very simple. We can modify the code to add some independent CPU computation as follows.

```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
increment<<<1,N>>>(d_a)
myCpuFunction(b)
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

Overlap transfer and computations?



- Two copy engines, one for host-to-device transfers and another for device-to-host transfers
- The device-to-host transfer of data in stream[i] does not block the host-to-device transfer of data in stream[i+1]

Details:
https://devblogs.nvidia.com/how-overlap-data-transfers-cuda-cc/

Overlapping Computation and Data-transfer

Streams

- A sequence of operations that execute on the device in the order in which they are issued by the host code
- Operations in different streams can be interleaved and, when possible, they can even run concurrently.
- A stream can be sequence of kernel launches and host-device memory copies
- Can have several open streams to the same device at once
- Need GPUs with concurrent transfer/execution capability
- Potential performance improvement: can overlap transfer and computation

Non-default streams in CUDA C/C++ are declared, created, and destroyed in host code as follows.

```
cudaStream_t stream1;
cudaError_t result;
result = cudaStreamCreate(&stream1)
result = cudaStreamDestroy(stream1)
```

To issue a data transfer to a non-default stream we use the cudaMemcpyAsync() function, which is similar to the cudaMemcpy() function, but takes a stream identifier as a fifth argument.

```
result = cudaMemcpyAsync(d_a, a, N, cudaMemcpyHostToDevice, stream1)
```

cudaMemcpyAsync() is non-blocking on the host, so control returns to the host thread immediately after the transfer is issued.

To issue a kernel to a non-default stream we specify the stream identifier as a fourth execution configuration parameter (the third execution configuration parameter allocates shared device memory, which we'll talk about later; use 0 for now).

```
increment<<<1,N,0,stream1>>>(d_a)
```

Since all operations in non-default streams are non-blocking with respect to the host code, we will run across situations where we need to synchronize the host code with operations in a stream.

- cudaDeviceSynchronize(): blocks the host code until all previously issued operations on the device have completed. In most cases this is overkill, and can really hurt performance due to stalling the entire device and host thread.
- cudaStreamSynchronize(stream): blocks the host thread until all previously issued operations in the specified stream have completed.
- cudaStreamQuery(stream) tests whether all operations issued to the specified stream have completed, without blocking host execution.
- ... (many more)

Overlapping Kernel Execution and Data Transfers

Requirements for overlap kernel execution with data transfers

- The device must be capable of "concurrent copy and execution". This can be queried from the deviceOverlap field of a cudaDeviceProp struct, or from the output of the deviceQuery sample included with the CUDA SDK/Toolkit. Nearly all devices with compute capability 1.1 and higher have this capability.
- The kernel execution and the data transfer to be overlapped must both occur in different, non-default streams.
- The host memory involved in the data transfer must be pinned (page-locked) memory.

Page-locked memory: memory pages on host have fixed mapping (pages cannot be moved (swap)). Allows direct memory access (DMA) of host memory from GPU (without CPU involvement).

Allocate and free memory on host (replace malloc, free)

```
cudaError_t cudaMallocHost(void** A, size_t size);
cudaError_t cudaFreeHost(void* A);
```

Global Memory Management (Asynchronous)

Page-locked memory: memory pages on host have fixed mapping (pages cannot be moved). Allows direct memory access (DMA) of host memory from GPU (without CPU involvement).

Allocate and free memory on host (replace malloc, free)

```
cudaError_t cudaMallocHost(void** A, size_t size);
cudaError_t cudaFreeHost(void* A);
```

Asynchronous memory transfers

```
cudaError_t cudaMemcpyAsync(void* A_d, const void* A,
                size_t N, cudaMemcpyKind kind);
```

Wait / Synchronize

```
cudaError_t cudaDeviceSynchronize();
```

Overlap tasks on CPU and GPU and synchronize at the end!

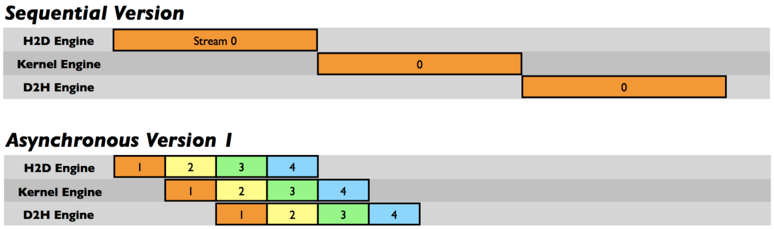
Introduce offset for current memory block to work on

```
__global__
void vec_add_kernel(double* c, const double* a, const double* b,
     long N, long offset){
  int idx = offset + blockIdx.x * blockDim.x + threadIdx.x;
  if (idx < N) c[idx] = a[idx] + b[idx];
}
```

Overlap computations and data transfer: loop over all the operations for each chunk of the array

```
for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes,
       cudaMemcpyHostToDevice, stream[i]);
  kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a,
       offset);
  cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes,
       cudaMemcpyDeviceToHost, stream[i]);
}
```

DEMO: gpu05.cu



- Two copy engines, one for host-to-device transfers and another for device-to-host transfers
- The device-to-host transfer of data in stream[i] does not block the host-to-device transfer of data in stream[i+1]

Details:
https://devblogs.nvidia.com/how-overlap-data-transfers-cuda-cc/

Global Memory Management (Managed)

Allocate memory on both host and device (accessed using the same pointer).
Let CUDA driver manage synchronizing memory between host and device.

Allocate memory on host and device

```
cudaError_t cudaMallocManaged(void** A, size_t size);
```

Free memory on host and device

```
cudaError_t cudaFree(void* A);
```

Wait / Synchronize before accessing memory on host

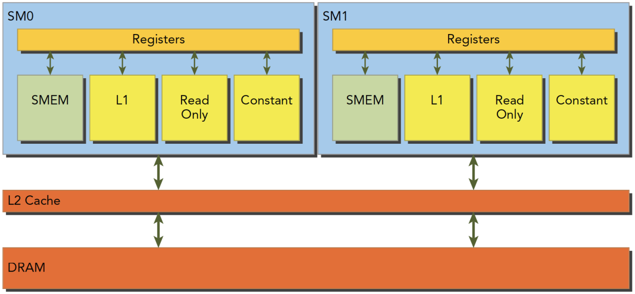
```
cudaError_t cudaDeviceSynchronize();
```

DEMO: gpu06.cu

Memory

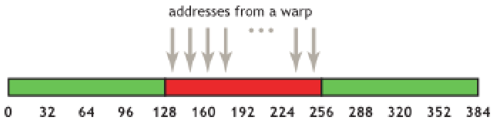
One of the distinguishing features of the CUDA execution model is that memory operations are issued per warp.

- Each thread in a warp provides a memory address it is loading or storing.
- Cooperatively, the 32 threads in a warp present a single memory access request comprised of the requested addresses, which is serviced by one or more device memory transactions.
- Global memory loads/stores are staged through caches

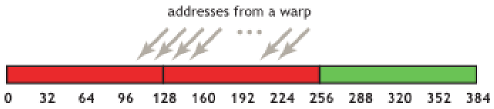


Optimizing Global Memory Accesses

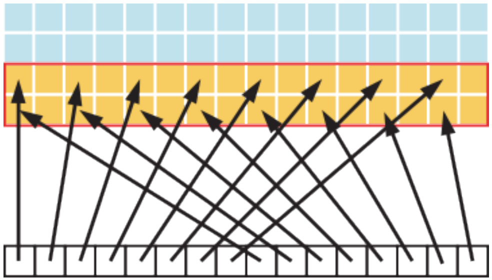
Use aligned, regular memory accesses from each warp



Avoid unaligned access

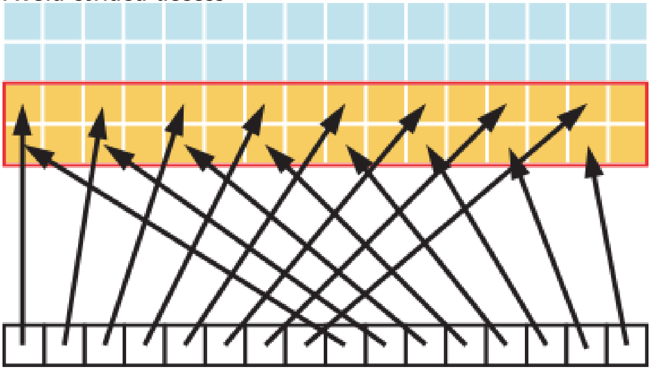


Avoid strided access



Global memory: Strided memory access

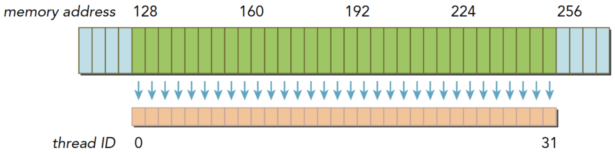
Avoid strided access

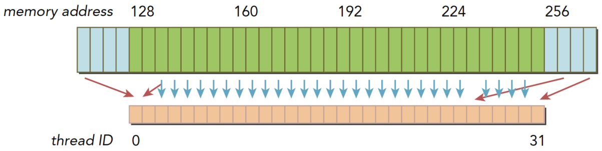


DEMO: gpu07.cu

https://devblogs.nvidia.com/how-access-global-memory-efficiently-cuda-c-kernels/

- **Aligned memory** accesses occur when the first address of a device memory transaction is an even multiple of the cache granularity being used to service the transaction (either 32 bytes for L2 cache or 128 bytes for L1 cache). Performing a misaligned load will cause wasted bandwidth.
- **Coalesced memory accesses** occur when all 32 threads in a warp access a contiguous chunk of memory.
- **Aligned coalesced memory accesses are ideal**: A wrap accessing a contiguous chunk of memory starting at an aligned memory address.

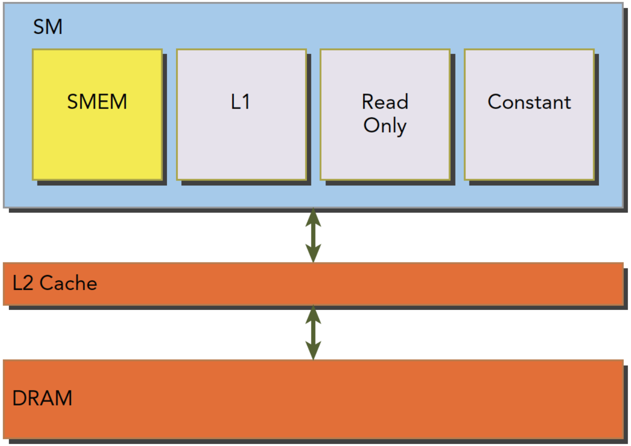




Shared memory

GPUs are equipped with two types of memory:

- On-board memory (global memory (DRAM); high latencies)
- On-chip memory (shared memory; small, low-latency, high bandwidth)
- Shared memory latency is roughly 20 to 30 times lower than global memory, and bandwidth is nearly 10 times higher



Shared memory

- Physically, each SM contains a small low-latency memory pool shared by all threads in the thread block currently executing on that SM.
- Shared memory enables threads within the same thread block to cooperate, facilitates reuse of on-chip data, and can greatly reduce the global memory bandwidth needed by kernels.
- Because the contents of shared memory are explicitly managed by the application, it is often described as a program-managed cache.

Access shared memory

- A fixed amount of shared memory is allocated to each thread block when it starts executing.
- Shared memory address space is shared by all threads in a thread block.
- Shared memory accesses are issued per warp (best: access by a warp is serviced by one transaction; worst case requires 32 unique transactions)

Warning: Shared memory is partitioned among all resident thread blocks on an SM; therefore, shared memory is a critical resource that limits device parallelism. The more shared memory used by a kernel, the fewer possible concurrently active thread blocks.

The following code segment statically declares a shared memory 2D float array. If declared inside a kernel function, the scope of this variable is local to the kernel. If declared outside of any kernels in a file, the scope of this variable is global to all kernels.

```
__shared__ float tile[size_y][size_x];
```

If the size of shared memory is unknown at compile time, can declare an un-sized array with the extern keyword.

```
extern __shared__ int tile[];
```

Then, to dynamically allocate shared memory at each kernel invocation one specifies the desired size in bytes as a third argument inside the triple angled brackets, as follows:

```
kernel<<<grid, block, isize * sizeof(int)>>>(...)
```

Note that you can only declare 1D arrays dynamically.

Thread synchronization

Careful with race conditions and shared memory: Threads in a block run logically in parallel, however not all threads can execute physically at the same time.

- Two threads A and B each load a data element from global memory and store it to shared memory.
- Then, thread A wants to read B's element from shared memory, and vice versa.
- If B has not finished writing its element before A tries to read it, then there is a race condition

Synchronization

- Simple barrier synchronization primitive, __syncthreads(). A thread's execution can only proceed past a __syncthreads() after all threads in its block have executed the __syncthreads()
- Avoid race condition by calling __syncthreads() after the store to shared memory and before any threads load from shared memory
- Calling __syncthreads() in divergent code is undefined and can lead to deadlock - all threads within a thread block must call __syncthreads() at the same point
- **__syncwarp()** synchronize all threads within a warp (32-threads).

DEMO: gpu08.cu

Code reverses the data in a 64-element array using shared memory

```
__global__ void staticReverse(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];
}
```

- In this kernel, t and tr are the two indices representing the original and reverse order, respectively.
- Threads copy the data from global memory to shared memory with the statement s[t] = d[t]
- The reversal is done two lines later with the statement d[t] = s[tr].
- But before executing this final line in which each thread accesses data in shared memory that was written by another thread, need to call __syncthreads() to make sure all threads have completed the loads to shared memory.

https://devblogs.nvidia.com/using-shared-memory-cuda-cc/

Using dynamic shared memory

```
__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];
}

int main(void)
{
  ...
  dynamicReverse<<<1,n,n*sizeof(int)>>>(d_d, n);
 ...
```

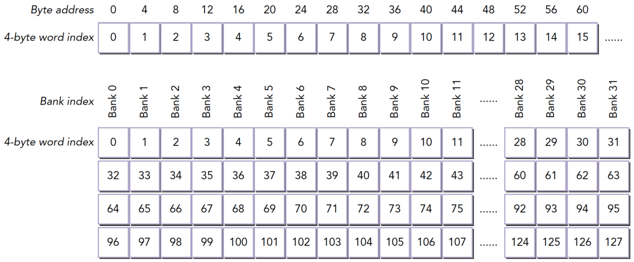
- Amount of shared memory is not known at compile time
- Shared memory allocation size per thread block must be specified (in bytes) using an optional third execution configuration parameter

https://devblogs.nvidia.com/using-shared-memory-cuda-cc/

Efficiently accessing shared memory

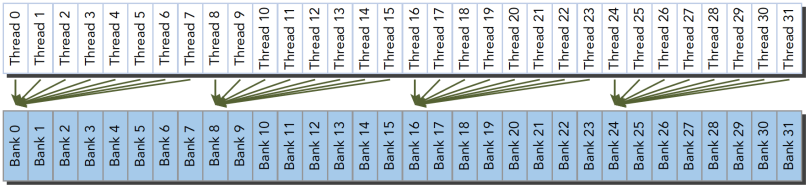
Memory banks

- To achieve high memory bandwidth, shared memory is divided into 32 equally-sized memory modules, called banks, which can be accessed simultaneously.
- If a shared memory load or store operation issued by a warp does not access more than one memory location per bank, the operation can be serviced by one memory transaction.
- Otherwise, the operation is serviced by multiple memory transactions, thereby decreasing memory bandwidth utilization.



Bank conflict

- When multiple addresses in a shared memory request fall into the same memory bank, a bank conflict occurs, causing the request to be replayed.
- The hardware splits a request with a bank conflict into as many separate conflict-free transactions as necessary $\Rightarrow$ decreasing the effective bandwidth by a factor equal to the number of separate memory transactions required.



Shared memory access patterns

Parallel access:

- Multiple addresses accessed by a warp that fall into multiple banks.
- Some, if not all, of the addresses can be serviced in a single memory transaction.
- Optimally, a conflict-free shared memory access is performed when every address is in a separate bank.

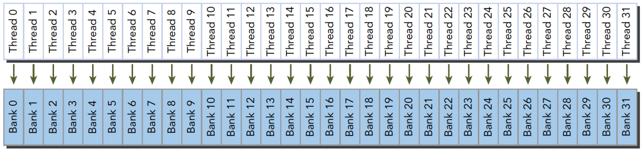
Serial access (worst):

- Multiple addresses fall into the same bank
- The request must be serialized.
- If all 32 threads in a warp access different memory locations in a single bank, 32 memory transactions will be required and satisfying those accesses will take 32 times as long as a single request.

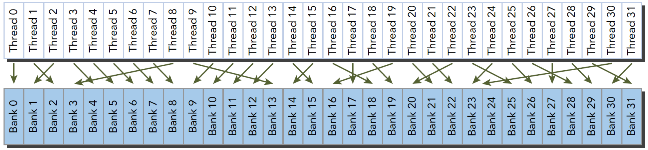
Broadcast access:

- All threads in a warp read the same address within a single bank.
- One memory transaction is executed, and the accessed word is broadcast to all requesting threads.
- While only a single memory transaction is required for a broadcast access, bandwidth utilization is poor because only a small number of bytes are read.

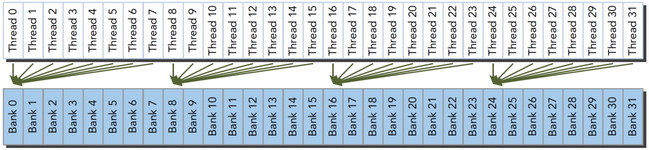
Parallel access (conflict-free)



Random but parallel access (conflict-free)



Irregular access pattern where several threads access the same bank



There are two possible behaviors for such a request:

- Conflict-free broadcast access if threads access same address within a bank
- Bank conflict access if threads access different addresses within a bank

Matrix transpose

Goal: avoid strided global memory access by using shared memory

Consider transpose of a matrix
https://devblogs.nvidia.com/efficient-matrix-transpose-cuda-cc/

- Operates out-of-place, i.e. the input and output are separate arrays in memory.
- Consider only square matrices whose dimensions are integral multiples of 32 on a side.
- Compare transpose and copy of matrix (without transpose)
- Performance measure: memory bandwidth

Main idea: Let each thread operate on a small tile (submatrix)

Simple matrix copy kernel

```
__global__ void copy(float *odata, const float *idata)
{
  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;

  for (int j = 0; j < TILE_DIM; j+= BLOCK_ROWS)
    odata[(y+j)*width + x] = idata[(y+j)*width + x];
}
```

- Each thread copies TILE_DIM/BLOCK_ROWS elements of the matrix in a loop at the end of this routine.
- Note also that TILE_DIM must be used in the calculation of the matrix index y rather than BLOCK_ROWS or blockDim.y (row vs. column first)
- The loop iterates over the second dimension and not the first so that contiguous threads load and store contiguous data, and all reads from idata and writes to odata are coalesced.

DEMO: gpu09.cu $\rightarrow$ this is our baseline case, cannot get faster than this in terms of bandwidth

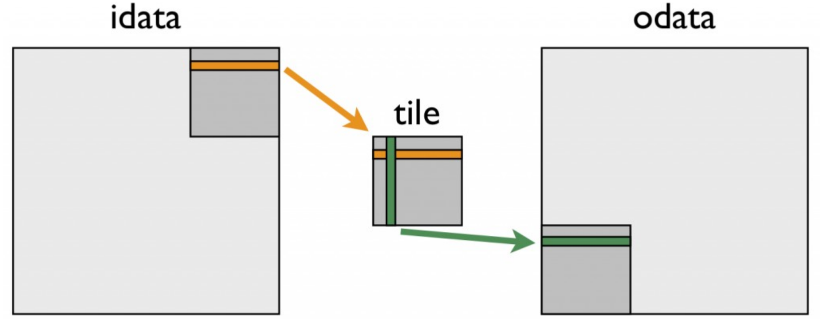
A naive transpose kernel

```
__global__ void transposeNaive(float *odata, const float *idata)
{
  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;

  for (int j = 0; j < TILE_DIM; j+= BLOCK_ROWS)
    odata[x*width + (y+j)] = idata[(y+j)*width + x];
}
```

- In transposeNaive the reads from idata are coalesced as in the copy kernel
- But for our 1024x1024 test matrix the writes to odata have a stride of 1024 elements or 4096 bytes between contiguous threads.
- We expect the performance of this kernel to suffer accordingly.

DEMO: gpu10.cu

One remedy for the poor transpose performance is to use shared memory to avoid the large strides through global memory.



[Figure: https://devblogs.nvidia.com/efficient-matrix-transpose-cuda-cc/]

```
__global__ void copySharedMem(float *odata, const float *idata)
{
  __shared__ float tile[TILE_DIM][TILE_DIM];

  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     tile[threadIdx.y+j][threadIdx.x] = idata[(y+j)*width + x];

  __syncthreads();

  x = blockIdx.y * TILE_DIM + threadIdx.x;  // transpose block
      offset
  y = blockIdx.x * TILE_DIM + threadIdx.y;

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     odata[(y+j)*width + x] = tile[threadIdx.x][threadIdx.y + j];
}
```

- First do loop, a warp of threads reads contiguous data from idata into rows of the shared memory tile.
- Need block-wise barrier synchronization __syncthreads() before reading data (because threads read different data than they have written)
- Then, a column of the shared memory tile is written to contiguous addresses in odata.
- Avoids non-coalesced global memory access

DEMO: gpu11.cu

```
$ nvcc -O3 -o gpu11 gpu11.cu
$ ./gpu11
...
                  Routine         Bandwidth (GB/s)
                     copy              118.48
          naive transpose               28.65
      coalesced transpose               60.69
```

- The transposeCoalesced results are an improvement over the transposeNaive case, but they are still far from the performance of the copy kernel.
- What is the cause of the difference/overhead?

```
$ nvcc -O3 -o gpu11 gpu11.cu
$ ./gpu11
...
                  Routine         Bandwidth (GB/s)
                     copy              118.48
          naive transpose               28.65
      coalesced transpose               60.69
```

- The transposeCoalesced results are an improvement over the transposeNaive case, but they are still far from the performance of the copy kernel.
- What is the cause of the difference/overhead?
- We might guess that the cause of the performance gap is the overhead associated with using shared memory and the required synchronization barrier __syncthreads().

```
__global__ void copySharedMem(float *odata, const float *idata)
{
  __shared__ float tile[TILE_DIM * TILE_DIM];

  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     tile[(threadIdx.y+j)*TILE_DIM + threadIdx.x] =
          idata[(y+j)*width + x];

  __syncthreads();

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     odata[(y+j)*width + x] = tile[(threadIdx.y+j)*TILE_DIM +
          threadIdx.x];
}
```

- Copy without tranpose using shared memory
- Comment: Is __syncthreads() needed here?

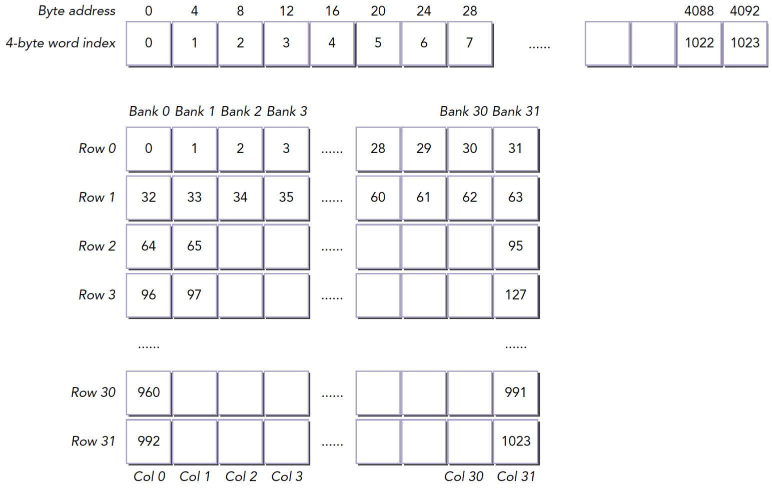
```
__global__ void copySharedMem(float *odata, const float *idata)
{
  __shared__ float tile[TILE_DIM * TILE_DIM];

  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     tile[(threadIdx.y+j)*TILE_DIM + threadIdx.x] =
          idata[(y+j)*width + x];

  __syncthreads();

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     odata[(y+j)*width + x] = tile[(threadIdx.y+j)*TILE_DIM +
          threadIdx.x];
}
```

- Copy without tranpose using shared memory
- Comment: Is __syncthreads() needed here? No, because the operations for an element are performed by the same thread, but we include it here to mimic the transpose behavior.

DEMO: gpu12.cu

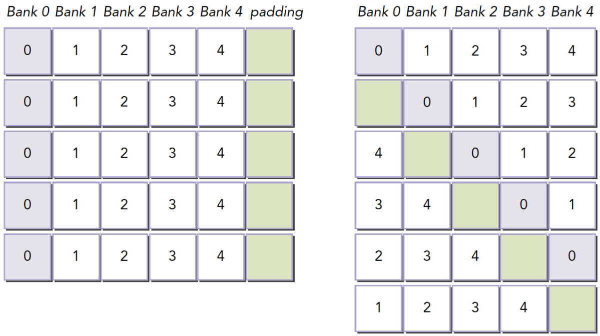
```
__global__ void copySharedMem(float *odata, const float *idata)
{
  ...
  __syncthreads();

  x = blockIdx.y * TILE_DIM + threadIdx.x;  // transpose block
      offset
  y = blockIdx.x * TILE_DIM + threadIdx.y;

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     odata[(y+j)*width + x] = tile[threadIdx.x][threadIdx.y + j];
}
```

- Memory layout for accessing the tile?



- Reading one column of a 32x32 tile leads to a 32-way bank conflict
- Each thread in a warp accesses the same memory bank (worst-case scenario)

Change the memory layout with padding



- If all threads access different locationsin bank 0, a five-way bank conflict occurs.
- Add a word of padding after every N elements, where N is the number of banks.
- This changes the mapping from words to banks, as illustrated on the right.
- The words that used to all belong to bank 0 are now spread across different banks because of the padding.

In our matrix transpose example, we need to change a single line only

```
__shared__ float tile[TILE_DIM][TILE_DIM+1];
```

DEMO: gpu13.cu

In our matrix transpose example, we need to change a single line only

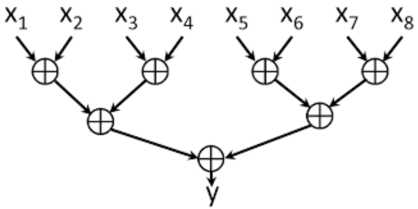
```
__shared__ float tile[TILE_DIM][TILE_DIM+1];
```

DEMO: gpu13.cu

```
                 Routine          Bandwidth (GB/s)
                    copy               118.42
      shared memory copy               119.71
         naive transpose                28.63
     coalesced transpose                60.80
 conflict-free transpose               119.54
```

Reduction

Parallel reduction algorithm:



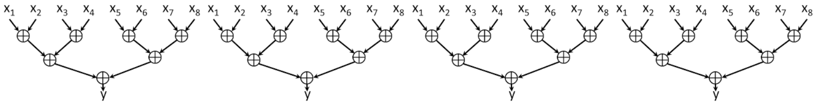
- $\log_2 N$ stages
- Requires coordination between threads!
- In each stage, every thread:
  - reads two numbers,
  - computes the sum and
  - writes its result to memory

Reduction

GPU algorithm

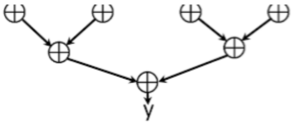
- Compute local reduction within each thread block
- Multiple kernel invocations for global synchronization

Stage 1: kernel with 4 thread-blocks, 8 threads per thread-block



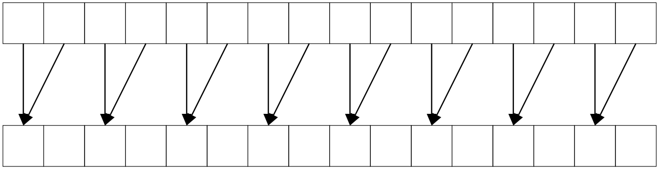
x1 x2 x3 x4 x5 x6 x7 x8 x1 x2 x3 x4 x5 x6 x7 x8 x1 x2 x3 x4 x5 x6 x7 x8 x1 x2 x3 x4 x5 x6 x7 x8
y
y
y
y

Stage 2: launch kernel with 1 thread-block with 4-threads

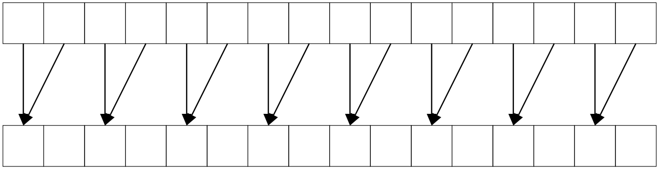


Reduction (within thread block)

Reduction (within thread block)

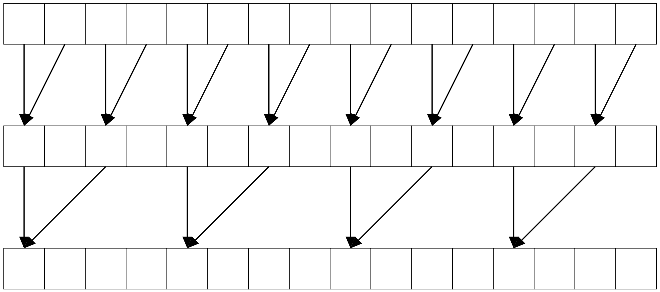


Reduction (within thread block)



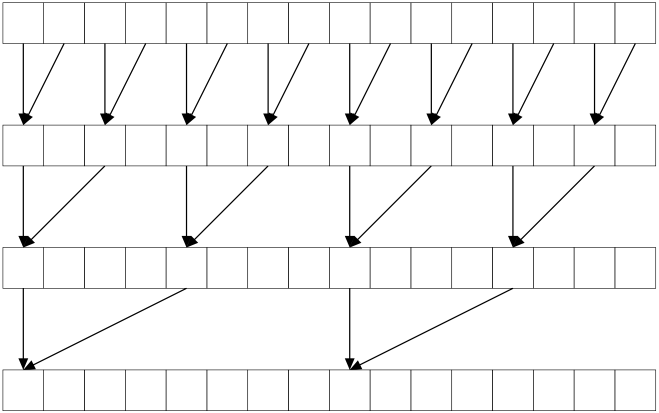
```
__syncthreads();
```

Reduction (within thread block)



```
__syncthreads();
```

Reduction (within thread block)



```
__syncthreads();
```

Reduction (within thread block)