

CUDA Course

Victor Eijkhout

2025 COE 379L / CSE 392

Justification

CUDA is the basic programming model for NVidia GPUs . . .
... which TACC is getting a lot of.



Table of contents

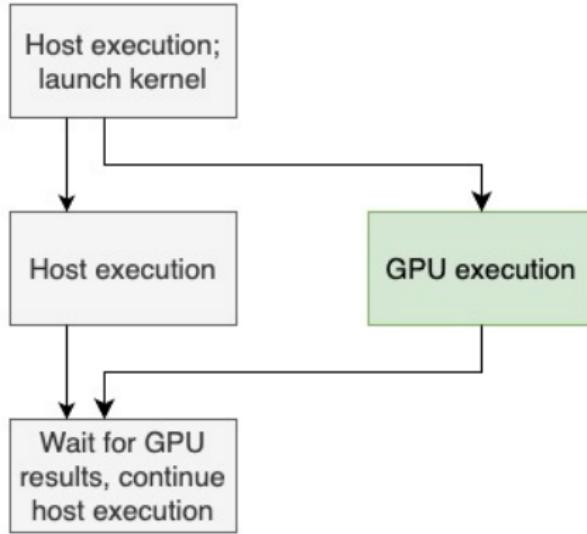
Introduction



Why GPUs

- Massive parallelism (but that's really multicore and SIMD)
- Simpler processor: less power / more computing for the same power.
- Simpler processor: no general programs, only 'data parallel' operations
 - ... such as machine learning
- Latest GPUs have very high speed on low precision
 - ... such as for machine learning

Attached processor



- GPU is attached to regular CPU
- computation is ‘offloaded’
- GPU has its own memory:
you may need to synchronize

Data parallel programming

- Kernel: operation for a single thread;
in effect: body of inner loop
- Grid: multi-dimensional structure of available 'threads'
executed on cores
(not a CPU core, more SIMD vector lane)



Processor structure

- Streaming Multiprocessor: like a core
- Warp: vector instruction
- Thread: like a SIMD-lane
- Single Instruction Multiple Thread (SIMT) ‘massive multi-threading’: you can have way more ‘threads’ than blocks \times blocksize
processor can switch very fast between threads

Loops vs CUDA

```
1 float *x;                                1 __global__ cu_f( float x) {  
2 for ( blockIdx_x < gridsize )           2   size_t linear = threadIdx_x  
3   for ( threadIdx_x < blocksize )         3     + blockIdx_x * blocksize;  
4     size_t linear = threadIdx_x           4   f( x[linear] );  
5     + blockIdx_x * blocksize;           5 }  
6   f( x[linear] );                         6  
7 float *x; // on device                  7  
8 cu_f<<<gridsize,blocksize>>>(x);  
9
```



GPU memory

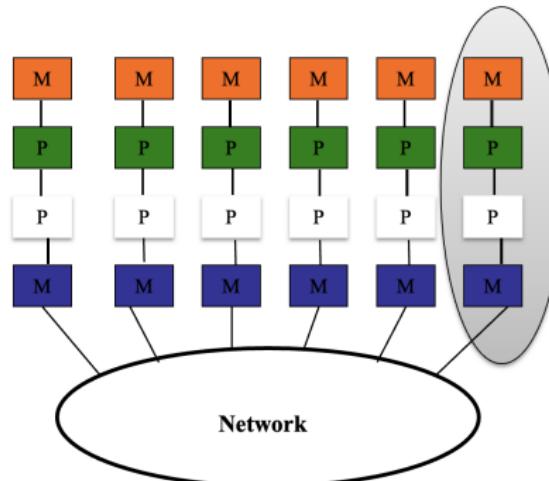
- Very much graphics inspired:
‘constant’ memory, ‘texture’ memory
- Main memory: has grown from 8G to 100G-ish
- Shared memory inside each SM: managed cache
- Actual caches and registers.

Comparison

- Dozens of cores
- Core has vector instructions
- a vector instruction has ‘lanes’
- Latency hiding by caches, prefetching
- Dozens of ‘Streaming Multiprocessors’
- SM has ‘warp’s
- a warp has threads
- Latency hiding by massive multi-threading

GPU cluster

Memory GPU
GPU
CPU
Memory CPU



Step 1:
One node

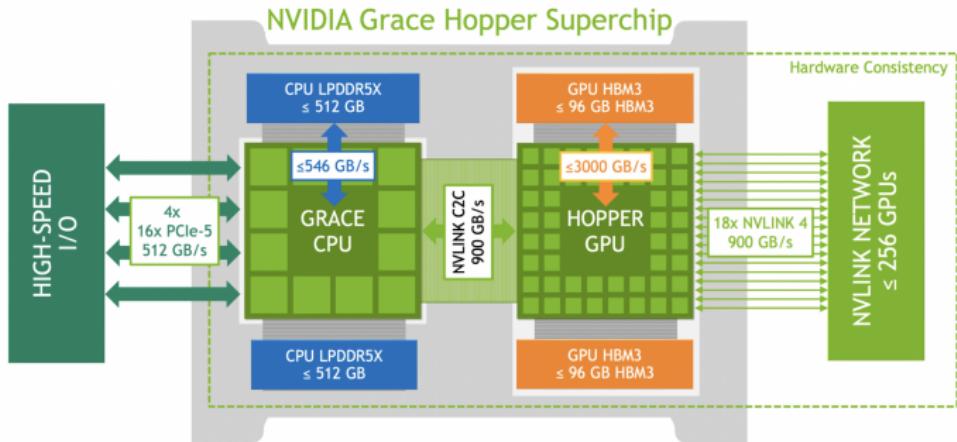
Step 2:
Multiple nodes

- Use GPU on a node
(possible combined with OpenMP or other threading)
- Use MPI between nodes;
NVlink also exists but not on large scale

TACC's Vista

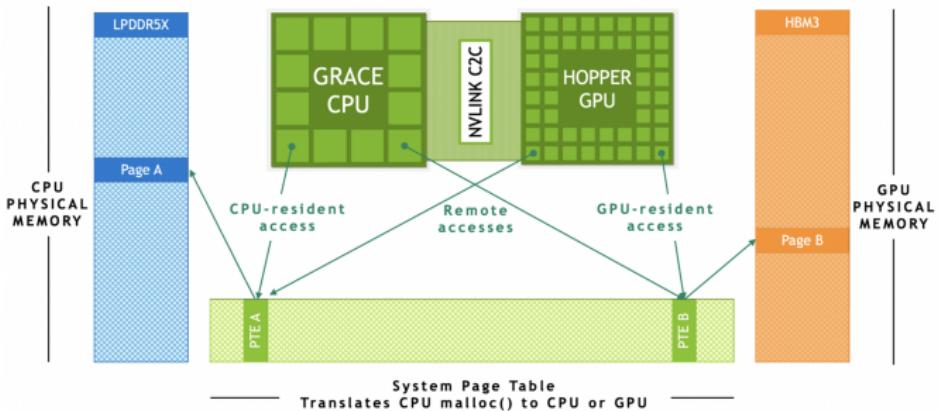
- 256 Grace-Grace two-socket CPU nodes
4 racks of 64 nodes each
- Grace-grace node has twice 72 cores: 144 total per node.
- 600 Grace-Hopper nodes
19 racks of 32 nodes
- GH200 Grace-Hopper superchip combines
each containing one 72-core Grace CPU (120 GiB LPDDR5X)
and H200 GPU (96 GiB HBM3)
- GH node: 34 Tflops of FP64
1979 Tflops of FP16 for Machine Learning.

Grace Hopper superchip



Grace Hopper superchip

Grace-Hopper GPU/CPU Memory Coherence



single page table enabled by address translation services (ATS)

71

- Single page table
- CPU and GPU can access data by paging.



CUDA and others

- CUDA is for C and Fortran;
limited to NVidia
- OpenACC, OpenCL: portable but tough to program
- AMD GPUs: ‘HIP’
- Intel GPUs (hm): Sycl
- OpenMP offload: universal but maybe not highest performance
- Kokkos: portable to OpenMP and all GPUs

Kernels

Translation code to kernel

```
1 for ( int i=0; i<n; ++i ) {  
2     int threadIdx.x = i;  
3     f(i);  
4 }  
5  
6 __global__ computef( /* fargs */ ) {  
7     int i = threadIdx.x;  
8     f(i);  
9 }  
10  
11 int gridsize,blocksize;  
12 computef<<<gridsize,blocksize>>>( fargs );
```

- Kernel is the body of the inner loop;
- Kernel is executed for each thread in the grid;
- Kernel consults `threadIdx` to find the ‘iteration’ number.



About threads

- CUDA creates thread blocks until all data has been processed
- You can tune the block size up to some limit;
- Make sure to create enough blocks.

Code:

```
1 // blocksize.cu
2 printf("Device Name: %s\n", prop.name);
3 printf("Max threads per block: %d\n",
4        prop.maxThreadsPerBlock);
5 printf("Max block dimensions:\n %d x %d x
6        %d\n",
7        prop.maxThreadsDim[0],
8        prop.maxThreadsDim[1],
8        prop.maxThreadsDim[2]);
```

Output:

```
1
2 -----
3           Welcome to
4           ↳the Vista
5           ↳Supercomputer
6 -----
7
8 No reservation for this
   ↳job
7 --> Verifying valid
      ↳submit host
      ↳(staff)...OK
8 --> Verifying valid
      ↳jobname...OK
9 --> Verifying valid ssh
      ↳keys...OK
10 --> Verifying access to
      ↳desired queue
      ↳(gh-dev)    OK
      ↳AUSTIN
```



Thread indexing

```
1 int global_thread_id = blockIdx.x * blockDim.x + threadIdx.x; // or 2d, 3d variant
2 if ( global_thread_id >= datasize ) return;
```

- Global thread index from block and thread coordinate;
- Global index often used as index in the data;
- You can have more threads than data;
no problem but you do need to test.

Data on the device

```
1 float
2   *x, *y, *z;
3 size_t nbytes = N*sizeof(float);
4 CU_ASSERT( cudaMallocManaged( &x,nbytes ) );
5 CU_ASSERT( cudaMallocManaged( &y,nbytes ) );
6 CU_ASSERT( cudaMallocManaged( &z,nbytes ) );
```

```
1 // Run kernel on 1M elements on the CPU
2 int blocksize = 1024;
3 int gridsize = N/blocksize+1;
4 add1<<<gridsize,blocksize>>>(N,x,y,z);
5 // Wait for GPU to finish before accessing data on host
6 CU_ASSERT( cudaDeviceSynchronize() );
```

- Managed allocation is accessible on host and device
- Device is asynchronous, so synchronization needed



Utility macro

```
1 #define CU_ASSERT( code ) { \
2     cudaError_t err = code ; \
3     if (err!=cudaSuccess) { \
4         printf("error <<%s>> on line %d\n",cudaGetErrorString(err),__LINE__); \
5         return 1; } }
```

- CUDA routines can return error code
- Check on fatal errors.

Exercise 1

Write a CUDA kernel that adds two arrays into a third.

Using the above snippets, finish the code, test for correctness.

Two-dimensional grid

```
1 dim3 blocksize(16,16);
2 dim3 gridsize( rowsize/blocksize.x+1,colsize/blocksize.y+1 );
3 printf(" .. grid is (%d,%d) blocks of (%d,%d)\n",
4     gridsize.x,gridsize.y,blocksize.x,blocksize.y);
```

- Two-dimensional data calls for two-dimensional thread organization (same for three of course)
- Specify grid/block size with `dim3` objects.

Two-dimensional indexing

```
1 // index2d.cu
2 // Calculate the row and column indices for this thread
3 size_t row = blockIdx.y * blockDim.y + threadIdx.y;
4 size_t col = blockIdx.x * blockDim.x + threadIdx.x;
5
6 // Make sure we don't go out of bounds
7 if (row < height && col < width) {
8 // Calculate the linear index for the matrices
9   size_t idx = row * width + col;
10
11 // Perform the addition
12   d_c[idx] = d_a[idx] + d_b[idx];
```

- Compute row/col from x/y components
- Adjacent threads need to access adjacent data



Thread numbering

0	1	2
3	4	5

6	7	
		11

12		
		17

18		
		23

- Lexicographic ordering of blocks;
- Lexicographic ordering of threads in blocks.

Exercise 2

Create a two-dimensional array and let each thread write its global number in the element where it is active:

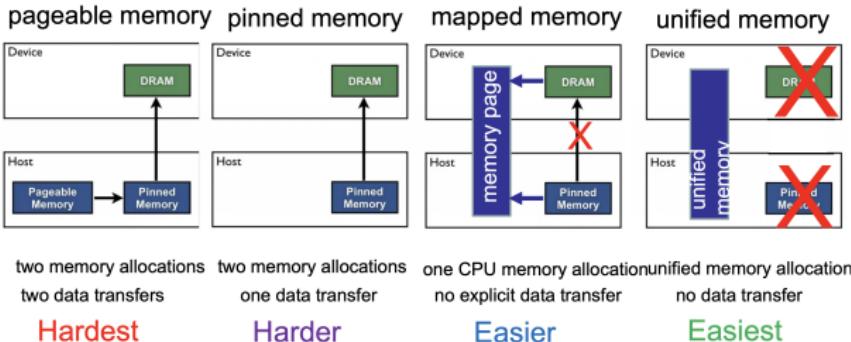
```
1 output[output_loc] = global_thread_num;
```

Your program should recreate the layout of figure 26.

Memory

Host and device memory

Grace/Hopper Memory Allocation Types



72

- Host memory can be copied;
- can be visible on the device;
- can be unified with the device.



Explicit transfer

Allocate input data and copy host values:

```
1 // input allocated on device
2 int *dev_input;
3 cudaMalloc( &dev_input,bytesize );
4 // host values copied
5 cudaMemcpy( dev_input,input,bytesize,cudaMemcpyHostToDevice );
```

After the kernel execution copy output data back:

```
1 cudaDeviceSynchronize();
2 cudaMemcpy( output,dev_output,bytesize,cudaMemcpyDeviceToHost );
```

Free:

```
1 free(input); free(output);
2 cudaFree(dev_input); cudaFree(dev_output);
```

- Host holds pointer to device memory;
- used for copying in/out;
- device memory freed.



Implicit transfer

```
1 // add1m.cu
2 size_t nbytes = N*sizeof(float);
3 CU_ASSERT( cudaMallocManaged( &x,nbytes ) );
4 CU_ASSERT( cudaMallocManaged( &y,nbytes ) );
5 CU_ASSERT( cudaMallocManaged( &z,nbytes ) );
```

- Possibility one:
allocate on the host with `cudaHostAlloc`
using a flag of `cudaHostAllocMapped`.
Pass `cudaHostGetDevicePointer` to the device.
- Easier: `cudaMallocManaged`;
transfer is done through on-demand paging.
- (Implications for timing!)



Shared memory

```
1 __shared__ float[32][32];
```

- Shared between threads in a block:
managed cache per core
- Can be used as fast local storage
- Copy from RAM to 'shared', operate on shared, copy back.

Synchronization

Mechanisms

- Threads in a warp are always synchronized.
- Threads in a block can be synchronized with `__syncthreads`.
- Blocks in a grid can not be synchronized.
- Host and device can be synchronized with `cudaDeviceSynchronize`.

Warps and warp divergence

```
1 if ( threadIdx.%2==0 )           1 bool mask = (threadIdx.x%2==0)
2   functionA();                  2 if (mask) functionA();
3 else                           3 if (not mask) functionB();
4   functionB();
```

- Threads in a warp are synchronized
- Conditionals are serialized: *warp divergence*
- Try to rewrite your code...



Synchronization across blocks

- Threads blocks are not synchronized; can not be.
Does this make sense from the original purpose of Graphics Processing Units (GPUs)?
- Solution 1: synchronize outside CUDA;
- Solution 2: use atomics;
- Solution 3: use a library that does it for you.

Example: sum reduction

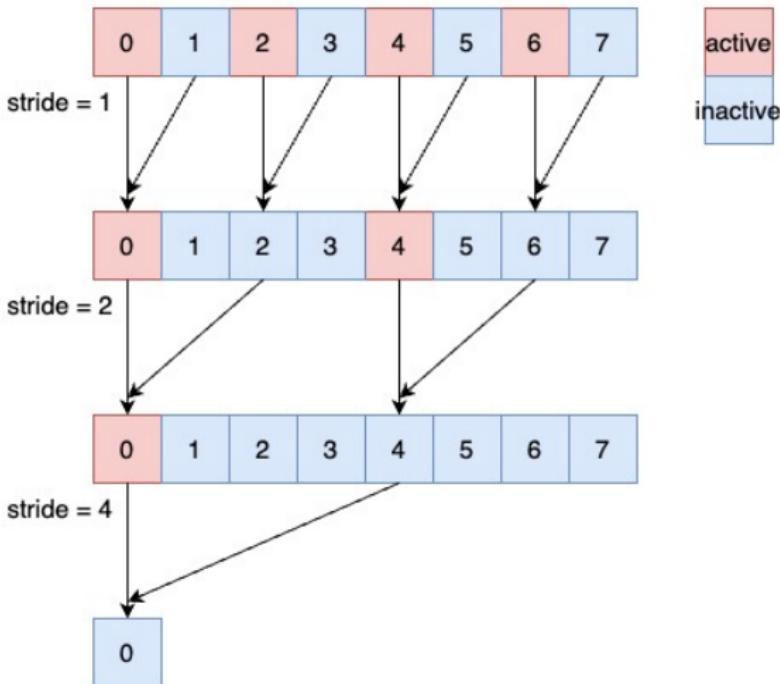
- Recursive summation in each block:

```
1 // redumy.cu
2 __global__
3 void reduce0(size_t n, float *input, float *y);
```

- Summing the blocks outside of CUDA:

```
1 for ( int b=0; b<gridsize; ++b)
2   value += y[b*blocksize];
```

Implementation 1



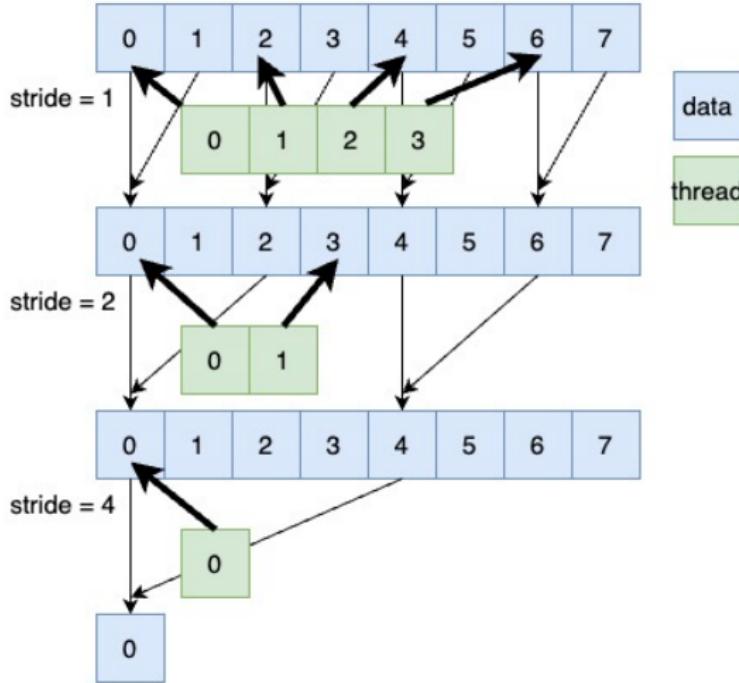
We let threads at distances 2, 4, 8, ... sum values from offsets 1, 2, 4,

Code

```
1 __global__
2 void reduce0(size_t n, float *input, float *y) {
3     size_t
4         thrd = threadIdx.x,
5         global = blockDim.x*blockIdx.x + threadIdx.x;
6     if (global>=n) return;
7     y[global] = input[global];
8     __syncthreads();
9     for ( size_t offset=1; offset<=blockDim.x/2; offset*=2 ) {
10         if (thrd%(2*offset)==0)
11             y[global] += y[global+offset];
12         __syncthreads();
13     }
14 }
```



Implementation 2



- Non-obvious mapping thread \leftrightarrow data

Code

```
1 // pointer to this block
2 float *data = y+blockDim.x*blockIdx.x;
3 for ( size_t offset=1; offset<=blockDim.x/2; offset*=2 ) {
4     size_t index = 2 * offset * thrd;
5     if (index+offset<blockDim.x)
6         data[index] += data[index+offset];
7     __syncthreads();
8 }
9 // we need the following statement really only for thread 0:
10 y[global] = data[thrd];
```



Implementation 3

- Use ‘Cuda UnBound’
- <https://github.com/NVIDIA/cccl/tree/main/cub>

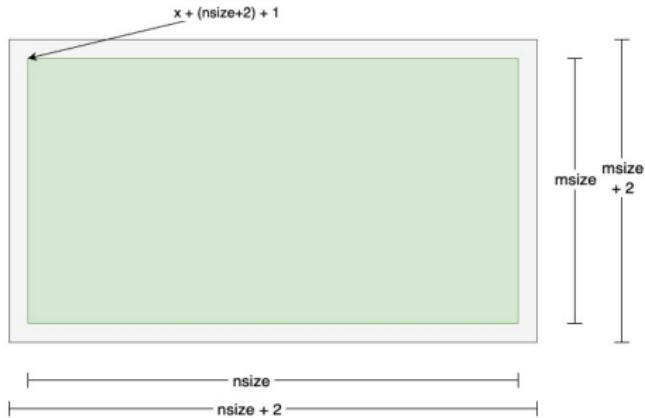
```
1 // compute needed tmp space
2 cub::DeviceReduce::Reduce
3   ( d_temp_storage, temp_storage_bytes,
4     x, &d_nrm, num_items, devplus{}, 0.f );
5 // alloc tmp space
6 cudaMalloc(&d_temp_storage, temp_storage_bytes);
7 // actual reduction
8 cub::DeviceReduce::Reduce
9   ( d_temp_storage, temp_storage_bytes,
10     x, &d_nrm, num_items, devplus{}, 0.f );
11 CU_ASSERT( cudaDeviceSynchronize() );
12 cudaFree(&d_temp_storage);
```



Example: stencil

Five-point stencil

$$y_{ij} \leftarrow 4x_{ij} - x_{i-1,j} - x_{i+1,j} - x_{i,j-1} - x_{i,j+1}$$



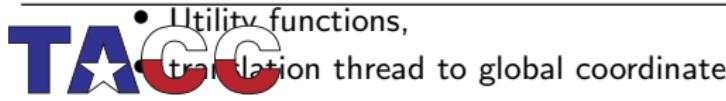
- Streaming kernel, but rhs elements used multiple times
- We use bordered data for easy of programming.

Coordinates

```
1 // smooth2d.cu
2 auto global_row = [=] ( int64_t by,int64_t ty ) {
3     return by * blockDim.y + ty; };
4 auto global_col = [=] ( int64_t bx,int64_t tx ) {
5     return bx * blockDim.x + tx; };
6 auto global_loc = [nsize] ( int64_t row,int64_t col ) {
7 // one blank row, and each row is left & right bordered
8     return (row+1) * (nsize+2) + col+1; };

1 int64_t bx = blockIdx.x, by = blockIdx.y, tx = threadIdx.x, ty = threadIdx.y;
2 int64_t my_global_row = global_row(by,ty);
3 int64_t my_global_col = global_col(bx,tx);
4 if ( my_global_row<0 or my_global_col<0
5     or my_global_row>=msize or my_global_col>=nsize )
6     return;

1 y[my_global_loc] = 4*x[my_global_loc]
2 - x[ global_loc( my_global_row-1,my_global_col ) ]
3 - x[ global_loc( my_global_row+1,my_global_col ) ]
4 - x[ global_loc( my_global_row ,my_global_col-1 ) ]
5 - x[ global_loc( my_global_row ,my_global_col+1 ) ];
```



- Utility functions,

- translation thread to global coordinate

Use of local memory

If elements read multiple times:

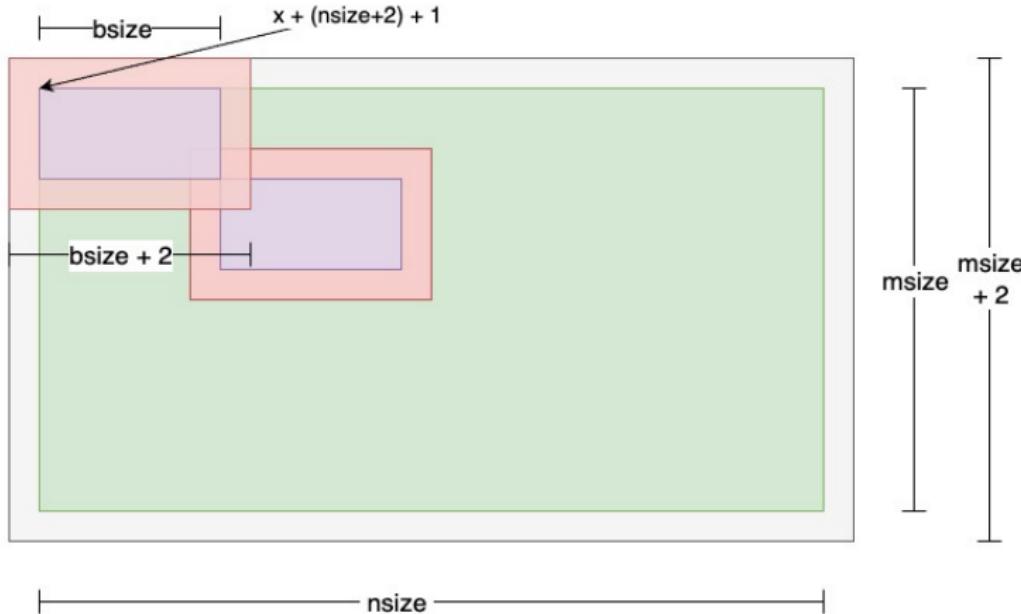
- Copy to local ('shared') data,
- Operate on that
- Write back

Shared memory

```
1 __shared__ float sums[BLOCKSIZE];  
2 sums[thrd_idx] = x[global_loc];  
3 __syncthreads();
```

- Shared memory size of thread block:
static dimensions
- Shared between all threads of block

Shared memory in stencil computation



- Assume bordered global domain,
- Make local block bordered too
- Tricky programming!

Exercise 3

Take your 2D vector addition code;
use shared memory. Do you see a performance difference?
If not, can you find a way to make this visible?

Timing, profiling and such

Device queries

```
1 // device.cu
2 int ndev;
3 auto status = cudaGetDeviceCount(&ndev);

1 cudaDeviceProp properties;
2 cudaGetDeviceProperties(&properties, idev);

1 cout << "Device " << idev << "=" << properties.name << '\n';
2 cout << "  async: " << properties.asyncEngineCount << '\n';
3 cout << "  unified: " << properties.unifiedAddressing << '\n';
4
5 cout << "  capability: " << properties.major << "." << properties.minor << '\n';
6 cout << "  multiprocs: " << properties.multiProcessorCount << '\n';
7 cout << "  clock rate: " << properties.clockRate << '\n';
8
9 cout << "  global memory: " << properties.totalGlobalMem << '\n';
10 cout << "  shared mem/block: " << properties.sharedMemPerBlock << '\n';
```



Block size

Code:

```
1 // blocksize.cu
2 printf("Device Name: %s\n", prop.name);
3 printf("Max threads per block: %d\n",
4        prop.maxThreadsPerBlock);
5 printf("Max block dimensions:\n %d x %d x
6        %d\n",
7        prop.maxThreadsDim[0],
8        prop.maxThreadsDim[1],
8        prop.maxThreadsDim[2]);
```

Output:

```
1
2 -----
3           Welcome to
4           ↳the Vista
5           ↳Supercomputer
4 -----
5
6 No reservation for this
    ↳job
7 --> Verifying valid
    ↳submit host
    ↳(staff)...OK
8 --> Verifying valid
    ↳jobname...OK
9 --> Verifying valid ssh
    ↳keys...OK
10 --> Verifying access to
    ↳desired queue
    ↳(gh-dev)...OK
11 --> Checking available
    ↳allocation
    ↳(A-ccsc).TEXAS
12 --> Quotas are not AT AUSTIN
    ↳currently
```



```
1 cudaEvent_t start, stop;
2 cudaEventCreate(&start);
3 cudaEventCreate(&stop);
4 float milliseconds = 0.0f;
5
6 cudaEventRecord(start);
7 cudaEventSynchronize(start);
8 // device operation goes here
9 cudaEventRecord(stop);
10 cudaEventSynchronize(stop);
11 cudaEventElapsedTime(&milliseconds, start, stop);
```

- You can time on the host, but make sure to synchronize
- Time on the device with events, act like barriers.





Exercise 4 (code)