CUDA

Kenjiro Taura

Contents

Contents

Goal

- learn CUDA, the basic API for programming NVIDA GPUs
- learn where it is similar to OpenMP and where it is different

CUDA reference

- official documentation:
  `https://docs.nvidia.com/cuda/index.html`
- book Professional CUDA C Programming
  `https://www.amazon.com/Professional-CUDA-Programming-John-Cheng/dp/1118739329`

Compiling/running CUDA programs with NVCC

- compile with `nvcc` command

```
$ nvcc program.cu
```

- the conventional extension of CUDA programs is `.cu`
- `nvcc` can handle ordinary C/C++ programs too (`.cc`, `.cpp` $\rightarrow$ C+)
- you can have a file with any extension and insist it is a CUDA program (convenient when you maintain a single file that compiles both on CPU and GPU)

```
$ nvcc -x cu program.cc
```

- run the executable on a node that has a GPU(s)

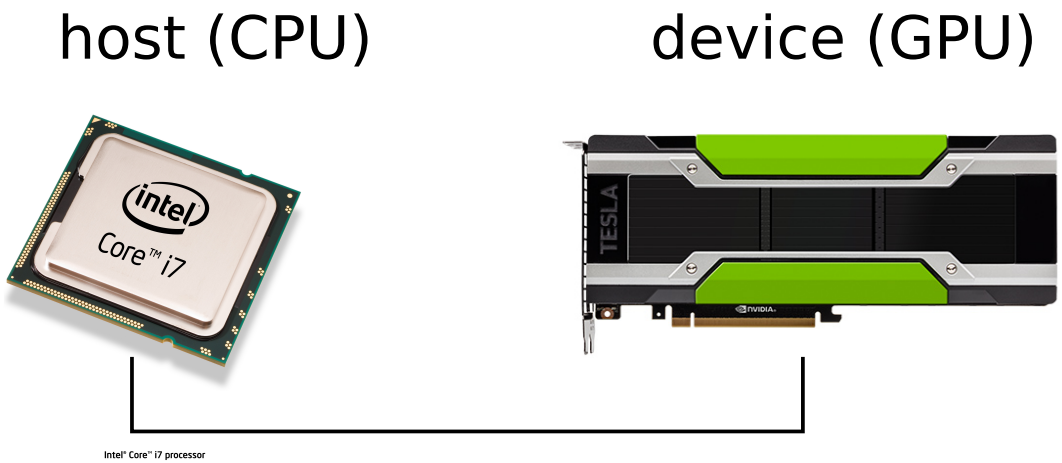
```
$ srun -p p ./a.out
```

Contents

GPU is a device separate from CPU

as such,

- code (functions) that runs on GPU must be so designated
- data must be copied between CPU and GPU
- a GPU is often called a *"device"*,
- and a CPU a *"host"*



Contents

Two things you need to learn first: writing and launching kernels

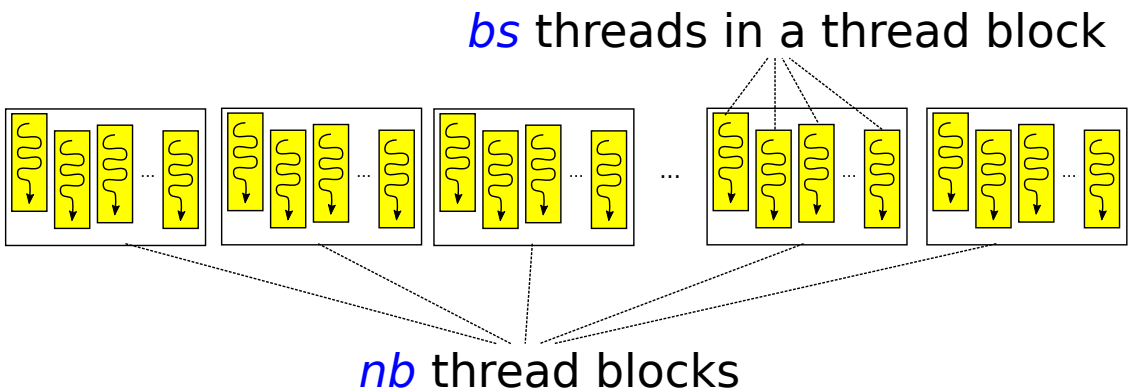
- a "GPU kernel" (or simply a "kernel") is a function that runs on GPU

```
1 __global__ void f(...) { ... }
```

- syntactically, a kernel is an ordinary C++ function that returns nothing (`void`), except for the `__global__` keyword
- a host launches a kernel specifying the number of threads.

```
1 f<<<nb,bs>>>(...);
```

will create $(nb \times bs)$ CUDA threads, each executing f(...)



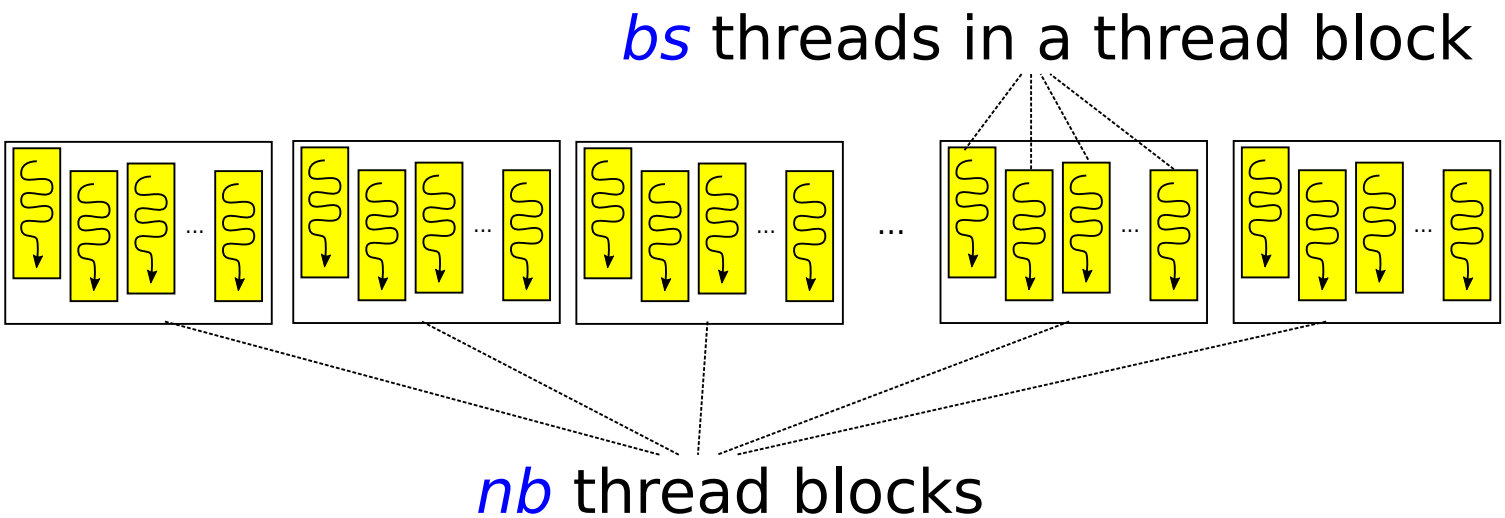
Launching a kernel ≈ parallel loop

- launching a kernel, like

```
1  f<<<nb,bs>>>(...);
```

- ≈ executing the following loop in parallel (on GPU, of course)

```
1  for (i = 0; i < nb * bs; i++) {
2    f(...); // CUDA thread
3  }
```



A simplest example

writing a kernel

```
1  __global__ void cuda_thread_fun(int n) {
2    int i        = blockDim.x * blockIdx.x + threadIdx.x;
3    int nthreads = gridDim.x * blockDim.x;
4    if (i < nthreads) {
5      printf("hello I am CUDA thread %d out of %d\n", i, nthreads);
6    }
7  }
```

and launching it

```
1  int thread_block_sz = 64;
2  int n_thread_blocks = (n + thread_block_sz - 1) / thread_block_sz;
3  cuda_thread_fun<<<n_thread_blocks,thread_block_sz>>>(n);
```

will print hello n times

```
1  hello I am CUDA thread 0 out of n
2     ...
3  hello I am CUDA thread n − 1 out of n
```

note: the order is unpredictable

A CUDA thread is not like an OpenMP thread

- launching 10000 CUDA threads is quite common and efficient

```
1 f<<<1024,256>>>(...);
```

- launching 10000 threads on CPU is almost always a bad idea
- below is "semantically" similar to the above

```
1 #pragma omp parallel
2   f();
```

```
1 OMP_NUM_THREADS=262144 ./a.out
```

but what happens inside is very different

- CPU way of doing this was:

```
1 #pragma omp parallel for
2 for (i = 0; i < 1024 * 256; i++) { f(); }
```

```
1 OMP_NUM_THREADS=a modest number ./a.out
```

a modest number = typically the actual number of cores

A kernel call and the host overlap but two kernel calls do not

- when you call a kernel, the host continues execution without waiting for it to finish
- two kernel calls are serialized on the GPU side, by default
- `cudaDeviceSynchronize()` is an API to wait for the kernel to finish

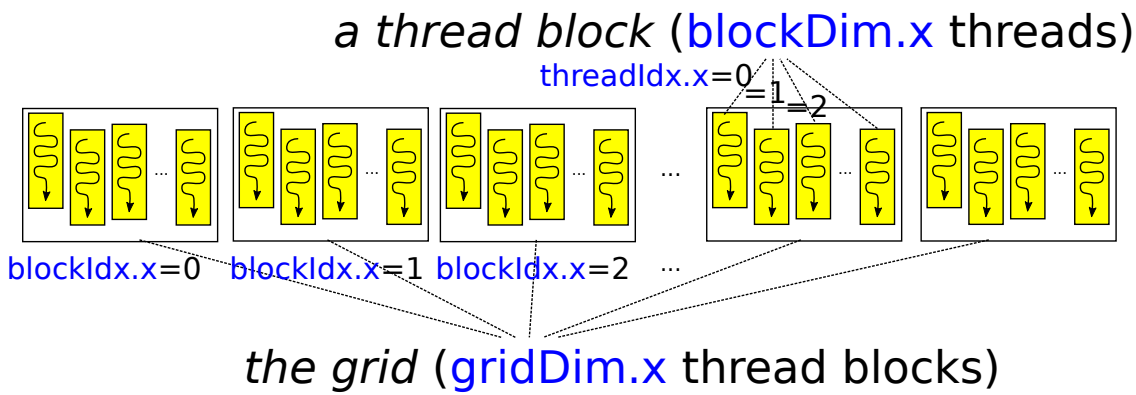
```
h0();
g0<<<...,...>>>();
h1();
g1<<<...,...>>>();
h2();
g2<<<...,...>>>();
cudaDeviceSynchronize();
h3();
```

- `g0` may overlap with `h1` and `h2`
- `g0` and `g1` do not overlap because of GPU serializes them by default
- `h3` does not overlap with anything because of `cudaDeviceSynchronize()`

About thread IDs

- for each thread to determine what to do, it needs a unique ID (the loop index)
- you get it from `gridDim`, `block{Dim,Idx}` and `threadIdx`
- when you launch a kernel by

```
1  f<<<nb,bs>>>(...);
```



  - `blockDim.x` = bs (the thread block size)
  - `gridDim.x` = nb (the number of blocks = the "grid" size)

and

  - `threadIdx.x` = the thread ID within the block ($\in [0, bs)$)
  - `blockIdx.x` = the thread's block ID ($\in [0, nb)$)

Remarks

- as suggested by `.x`, a block and the grid can be multidimensional (up to 3D, of `.x`, `.y`, `.z`) and the previous code assumes they are 1D
- extension to multidimensional block/grid is straightforward
- 1D:

```
1 int nb = 100;
2 int bs = 256
3 f<<<nb,bs>>>(...); // 100*256 threads
```

- 2D:

```
1 dim3 nb(10,10);
2 dim3 bs(8,32);
3 f<<<nb,bs>>>(...); // 10*10*8*32 threads
```

- 3D:

```
1 dim3 nb(10,5,2);
2 dim3 bs(8,8,4);
3 f<<<nb,bs>>>(...); // 10*5*2*8*8*4 threads
```

SpMV in CUDA

- original serial code

```
1  for (k = 0; k < A.nnz; k++) {
2    i,j,Aij = A.elems[k];
3    y[i] += Aij * x[j];
4  }
```

- write a kernel that *works on a single non-zero element*

```
1  __global__ spmv_dev(A, x, y) {
2    k = blockDim.x * blockIdx.x + threadIdx.x; // thread id
3    if (k < A.nnz) {
4      i,j,Aij = A.elems[k];
5      y[i] += Aij * x[j];   }   }
```

- and launch it with $\geq$ **nnz** threads *(we're not done yet)*

```
1  spmv*(A, x, y) {
2    int bs = 256;
3    int nb = (A.nnz + bs - 1) / bs;
4    spmv_dev<<<nb,bs>>>(A, x, y);  }
```

- similarly simple for CSR version

We're not done yet

- this code

```
1  __global__ spmv_dev(A, x, y) {
2    k = blockDim.x * blockIdx.x + threadIdx.x;
3    if (k < nnz) {
4      i,j,Aij = A.elems[k];
5      y[i] += Aij * x[j];
6    }
7  }
```

does not work yet

1. the device cannot access elements of `A`, `x` and `y` on the host
2. there is a race condition when updating `y[i]`

Keywords for functions

- `__global__`, `__device__`, `__host__`

	callable from	code runs on
`__global__`	host/device	device
`__device__`	device	device
`__host__`	host	host

- `__global__` functions cannot return a value (must be `void`)
- you can have both `__host__` and `__device__` in front of a definition, which generates two versions (device and host)

Macros

- convenient when writing a single file that works both on CPU and GPU
- `__NVCC__` : a macro defined when compiled by nvcc

```
#ifdef __NVCC__
  // GPU implementation
#else
  // CPU implementation
#endif
```

- `__CUDA_ARCH__` : a macro defined when copiled for device

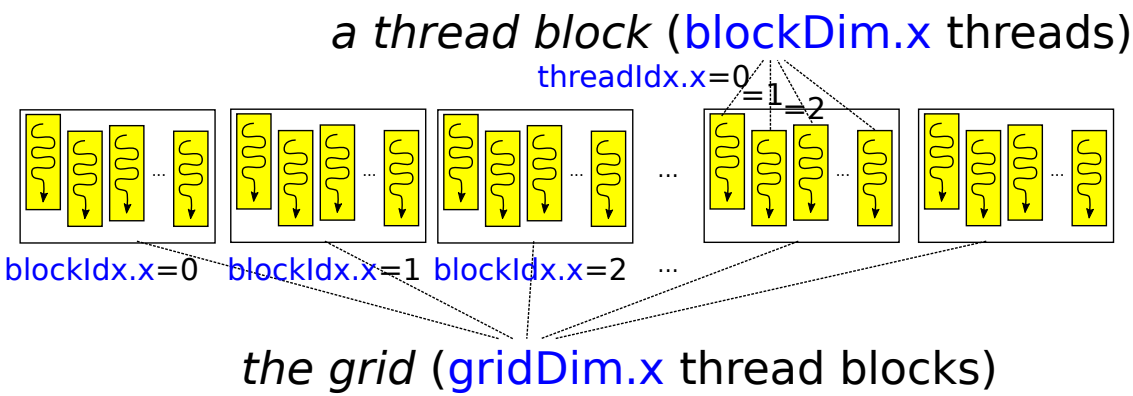
```
__device__ __host__ f(...) {
#ifdef __CUDA_ARCH__
  // device code
#else
  // host code
#endif
}
```

Contents

Threads and thread blocks (recap)

- a kernel specifies the action of *a* CUDA thread
- when you launch a kernel you specify
  - the number of thread blocks (nb) and
  - the thread block size = the number of threads in a single thread block (bs),

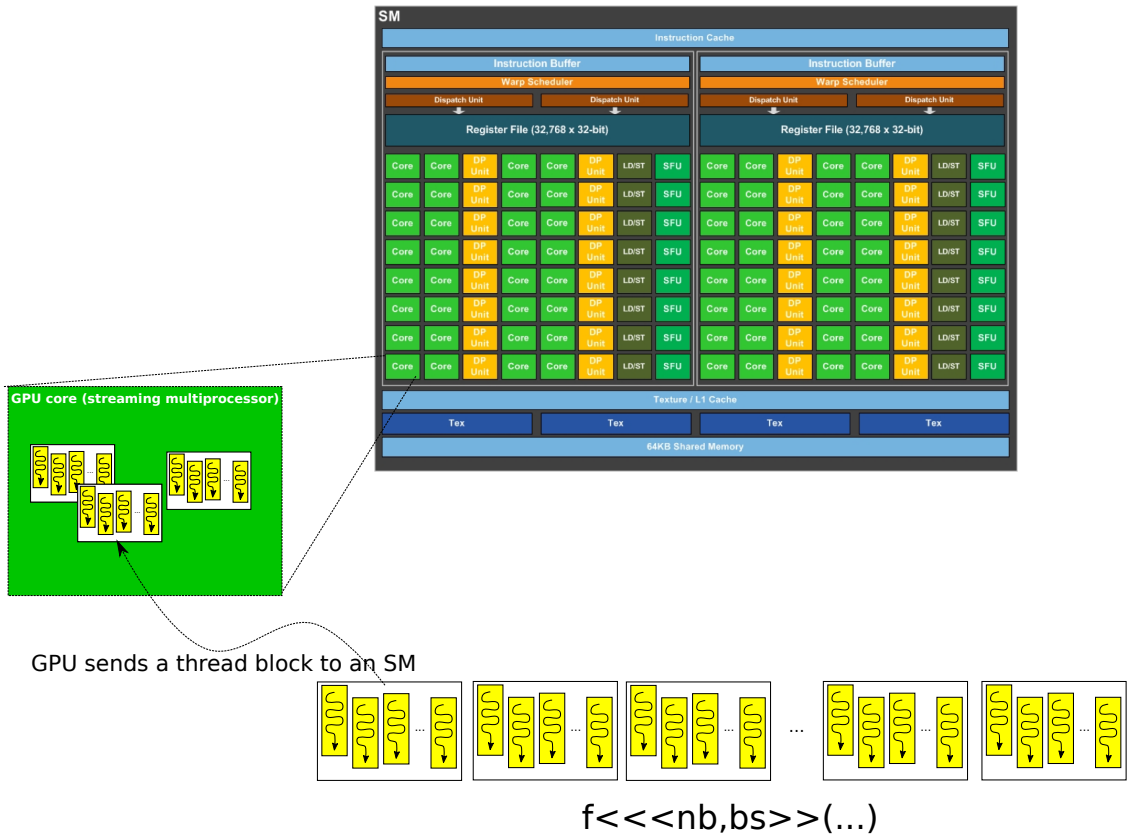
  to effectively create ($nb \times bs$) threads



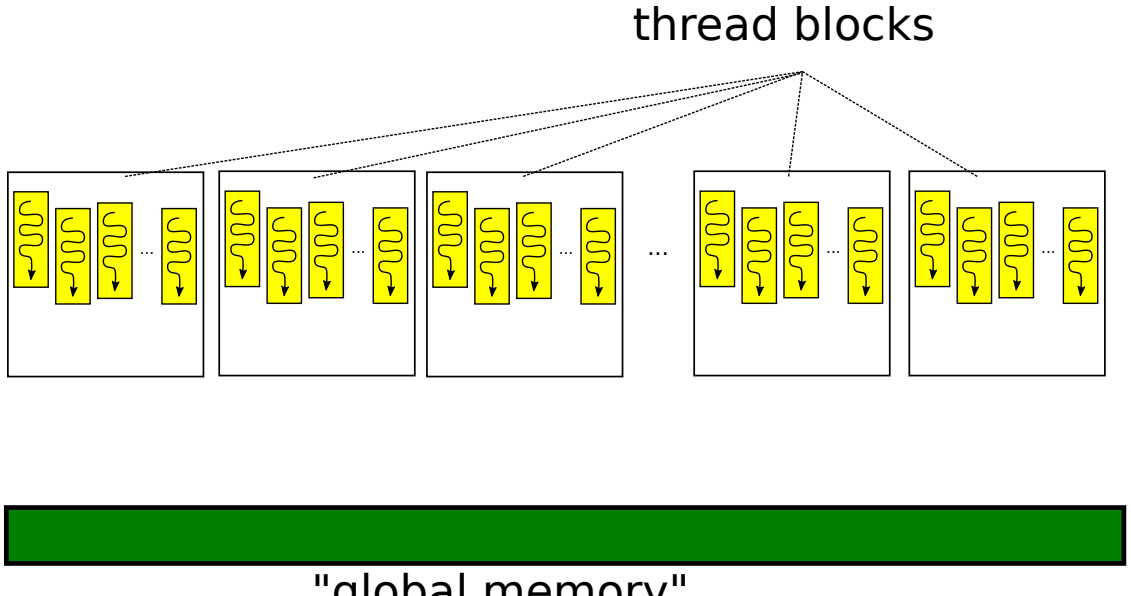
but why you need two separate numbers?

Why two numbers (bs and nb)?

a single thread block is sent to a single SM and stays there until it finishes

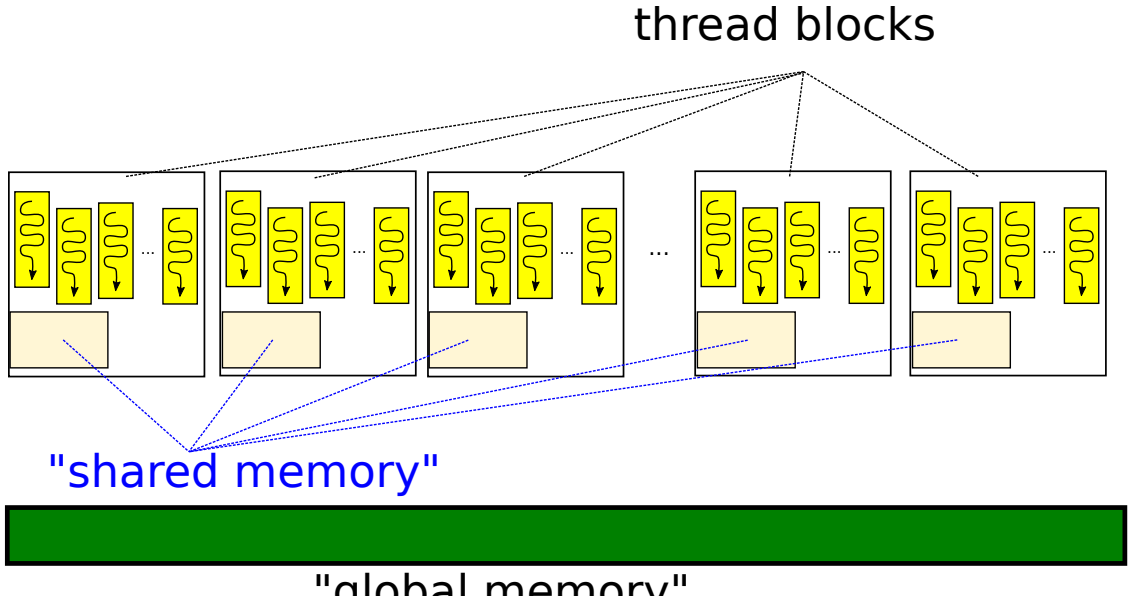


The way thread block boundaries are *semantically* visible



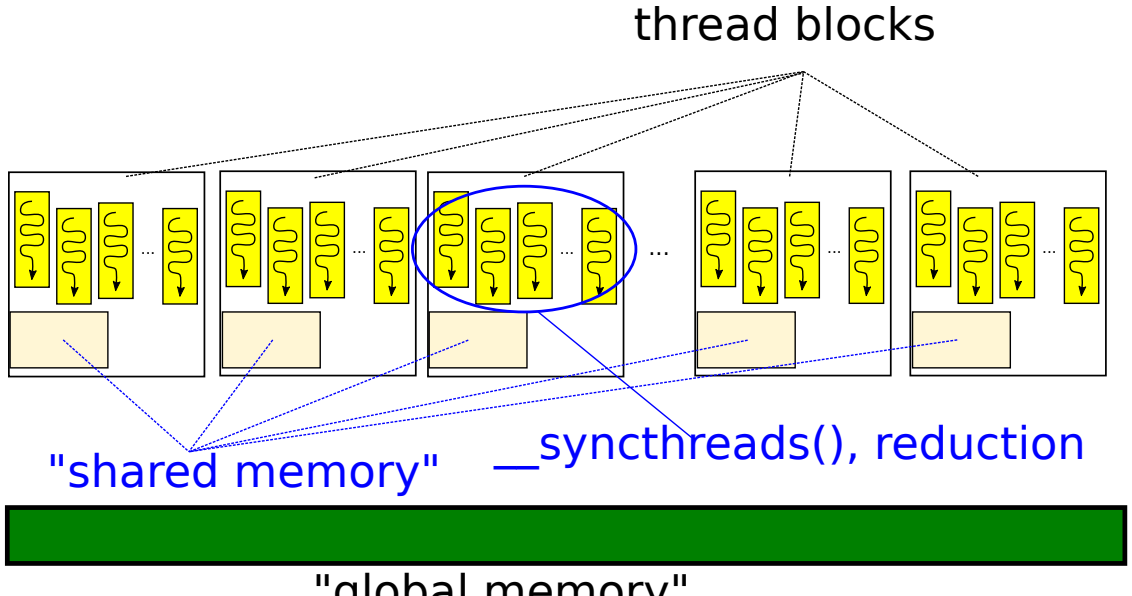
The way thread block boundaries are *semantically* visible

- CUDA API exposes *"shared memory"*, a small cache-like memory only shared within a single thread block



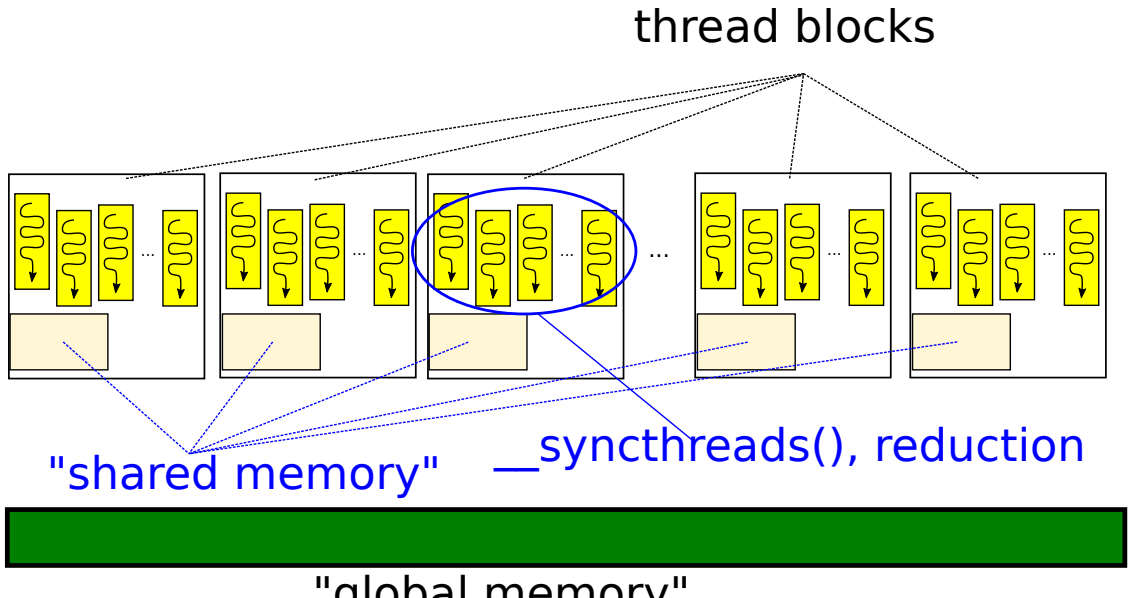
The way thread block boundaries are *semantically* visible

- CUDA API exposes *"shared memory"*, a small cache-like memory only shared within a single thread block
- CUDA API exposes some synchronization/coordination primitives (e.g., `__syncthreads()` or reduction) only usable within a single thread block



The way thread block boundaries are *semantically* visible

- CUDA API exposes *"shared memory"*, a small cache-like memory only shared within a single thread block
- CUDA API exposes some synchronization/coordination primitives (e.g., `__syncthreads()` or reduction) only usable within a single thread block
- unless you rely on these primitives, choosing the thread block size is largely a performance (not a correctness) issue



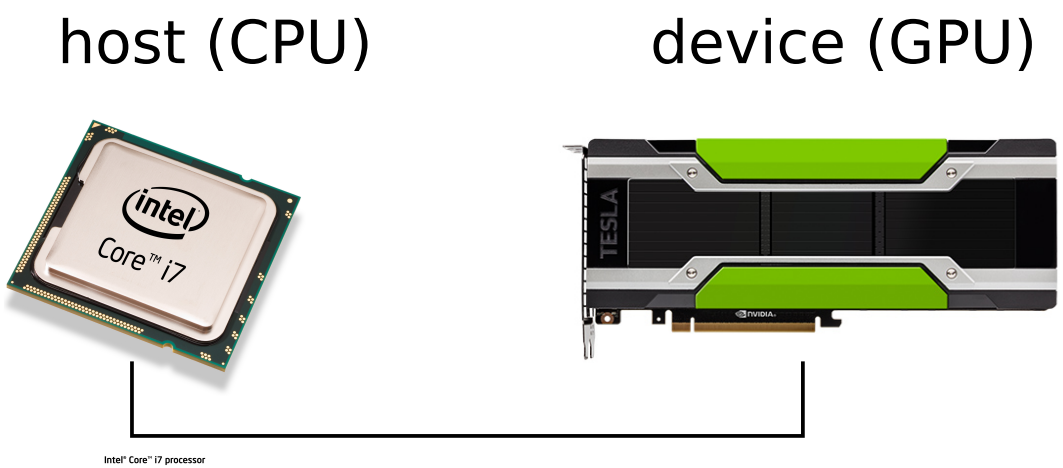
Contents

Moving data between host and device

- host and device memory are *separate*
- the device cannot access data on the host and vice versa (at least not directly by hardware until recently)
- i.e., the following does not work

```
1 double a[n];
2 f<<<nb,bs>>>(a);
```

```
1 __global__ f(double * a) {
2   ... a[i] ... // this will segfault
3 }
```

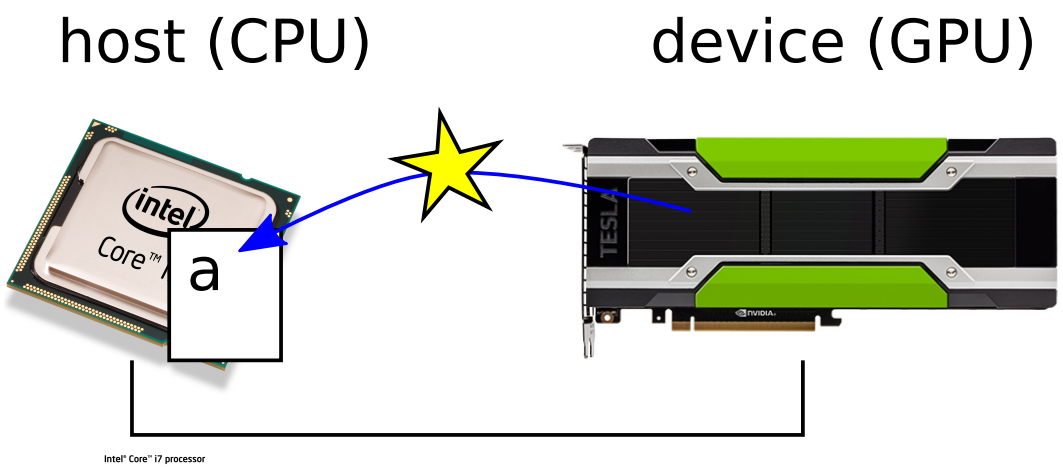


Moving data between host and device

- host and device memory are *separate*
- the device cannot access data on the host and vice versa (at least not directly by hardware until recently)
- i.e., the following does not work

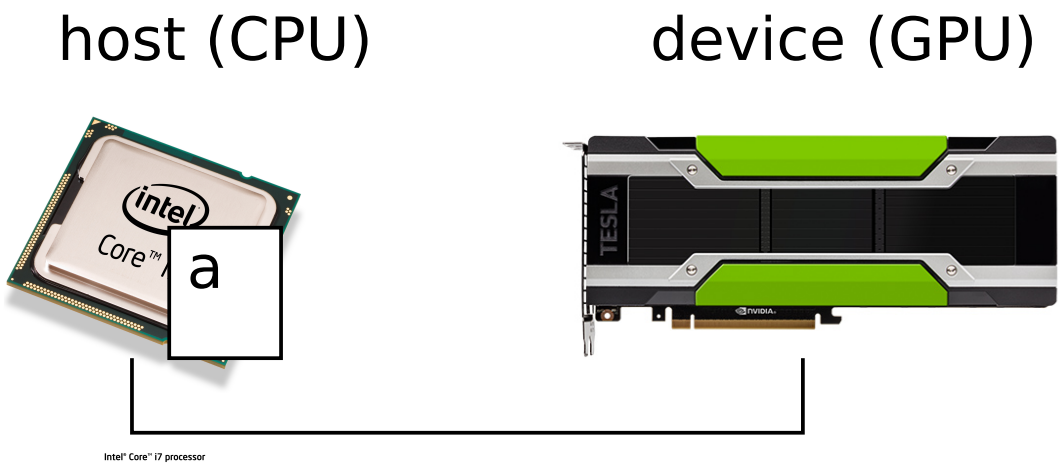
```
1 double a[n];
2 f<<<nb,bs>>>(a);
```

```
1 __global__ f(double * a) {
2   ... a[i] ... // this will segfault
3 }
```



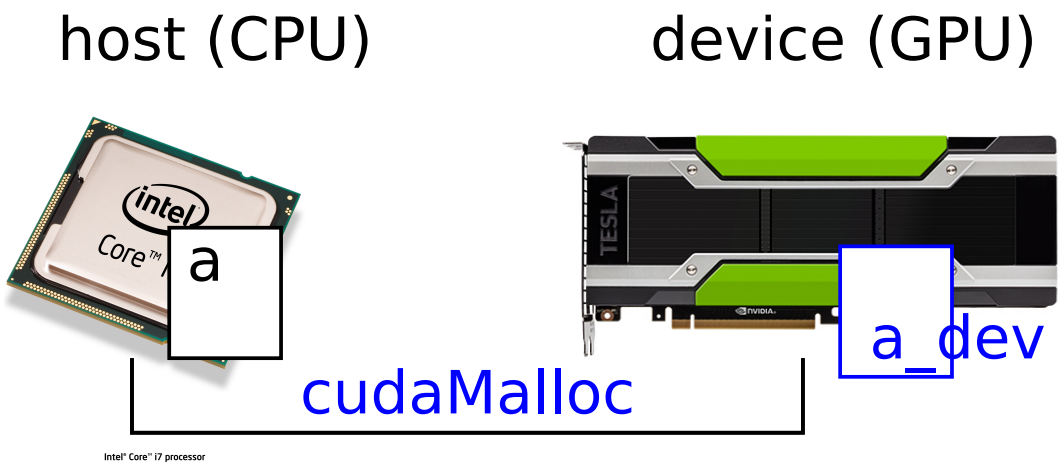
Two more things you must master: `cudaMalloc` and `cudaMemcpy`

- you need to



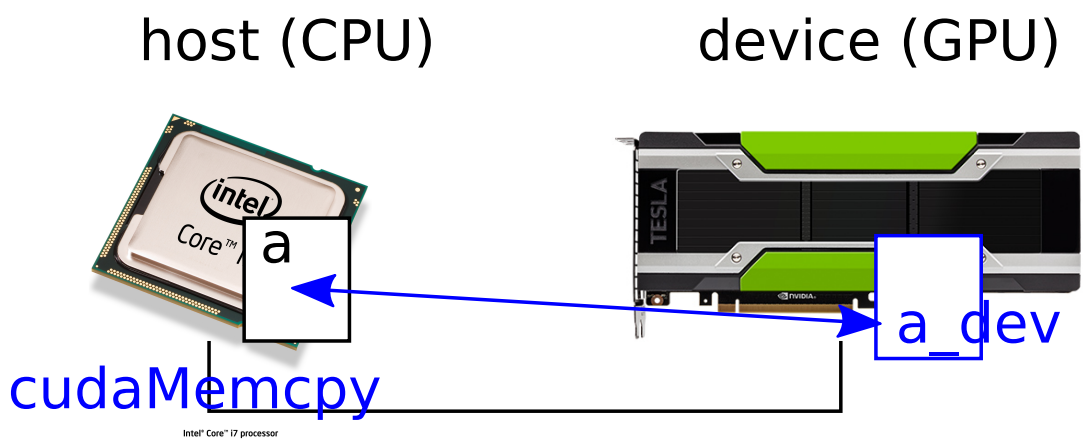
Two more things you must master: `cudaMalloc` and `cudaMemcpy`

- you need to
  1. allocate data on device (by `cudaMalloc`) $\rightarrow$ *device memory*



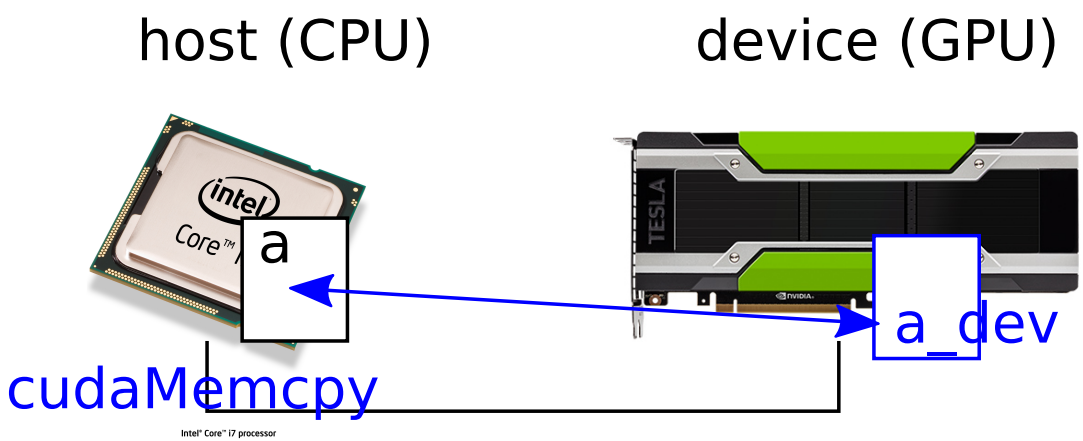
Two more things you must master: `cudaMalloc` and `cudaMemcpy`

- you need to
  1. allocate data on device (by `cudaMalloc`) $\rightarrow$ *device memory*
  2. move data between the host and the device (by `cudaMemcpy`)



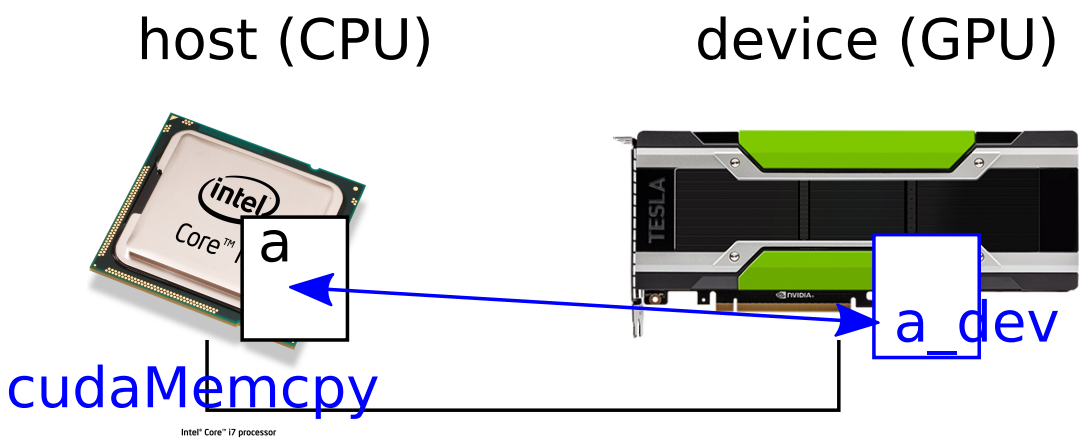
Two more things you must master: `cudaMalloc` and `cudaMemcpy`

- you need to
  1. allocate data on device (by `cudaMalloc`) → *device memory*
  2. move data between the host and the device (by `cudaMemcpy`)
  3. give the kernel the pointer to the device memory



Two more things you must master: `cudaMalloc` and `cudaMemcpy`

- you need to
  1. allocate data on device (by `cudaMalloc`) → *device memory*
  2. move data between the host and the device (by `cudaMemcpy`)
  3. give the kernel the pointer to the device memory
- note: call `cudaMalloc` and `cudaMemcpy` on the host, not on the device



Typical steps to send data to the device

1. allocate data of the same size both on host and device

```
1 double * a = ...;  // any valid address will do (malloc, &variable, etc.)
2 double * a_dev = 0;
3 cudaMalloc((void **)&a_dev, sz);
```

2. the host works on the host data

```
1 for ( ... ) { a[i] = ... } // whatever initialization you need
```

3. copy the data to the device

```
1 cudaMemcpy(a_dev, a, sz, cudaMemcpyHostToDevice);
```

4. pass the device pointer to the kernel

```
1 f<<<nb,bs>>>(a_dev, ...)
```

5. often a good idea to have a struct encapsulating both pointers

```
1 typedef struct {
2   double * a;     // host pointer
3   double * a_dev; // device pointer
4    ...          } my_struct;
```

Typical steps to retrieve the result

1. allocate data of the same size both on host and device

```
1 double * r = ... ;
2 double * r_dev = 0;
3 cudaMalloc((void **)&r_dev, sz);
```

2. pass the device pointer to the kernel

```
1 f<<<nb,bs>>>(..., r_dev);
```

3. copy the data to the host

```
1 cudaMemcpy(r, r_dev, sz, cudaMemcpyDeviceToHost);
```

Unified Memory

- recent NVIDIA GPUs support Unified Memory that eliminate the need for explicit data movement between host and device memory and dual pointer management
- at the heart of it is `cudaMallocManaged`, which is like `cudaMalloc` but is directly accessible from host CPU too

Typical steps to send data to the device with Unified Memory

1. allocate data of the same size both on host and device

```
1 double * a = 0;
2 cudaMallocManaged((void **)&a, sz);
```

2. the host works on the host data

```
1 for ( ... ) { a[i] = ... } // whatever initialization you need
```

3. pass the pointer to the kernel

```
1 f<<<nb,bs>>>(a, ...)
```

Typical steps to retrieve the result with Unified Memory

1. allocate data with `cudaMallocManaged`

```
1 double * r = 0;
2 cudaMallocManaged((void **)&r, sz);
```

2. pass the device pointer to the kernel

```
1 f<<<nb,bs>>>(..., r);
```

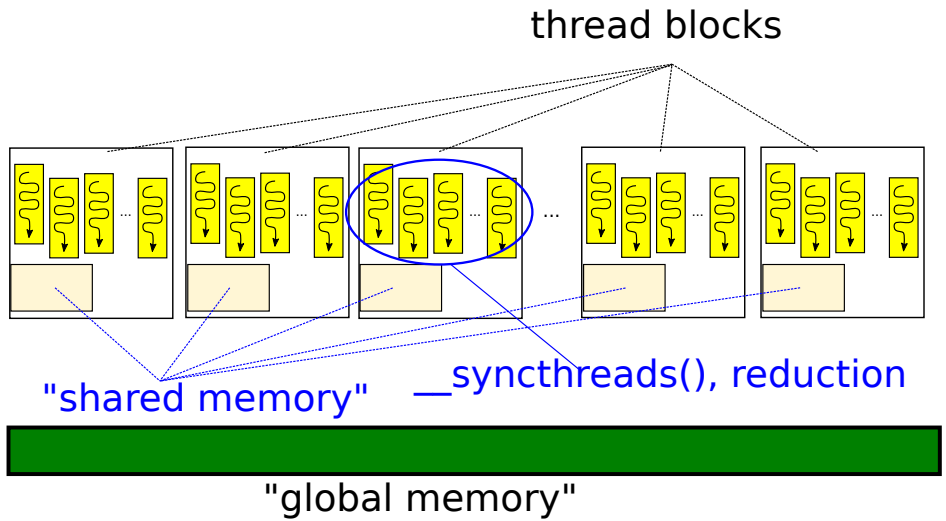
3. make sure threads finished their work

```
1 cudaDeviceSynchronize();
```

Contents

Data sharing among threads in the device

- basics : memory allocated via `cudaMalloc(Managed)?` *are shared among all threads (global memory)*
  - a write by a thread will be visible to all others (sooner or later)
- *shared memory* :
  - hardware terms: a small on-chip memory as fast as caches, not coherent across SMs
  - software view: memory shared only within a thread block
- other weirder memory types not covered in the lecture (constant and texture)



How to resolve race conditions on global/shared memory?

- CUDA threads run concurrently so they are susceptible to race conditions as in CPUs

```
1  __global__ spmv_dev(A, x, y) {
2    k = blockDim.x * blockIdx.x + threadIdx.x; // thread id
3    if (k < nnz) {
4      i,j,Aij = A.elems_dev[k];
5      y[i] += Aij * x[j];
6    }
7  }
```

Resolving race condition on CUDA

- atomic accumulations (`atomicAdd` and other functions)
- forget about mutual exclusion (no `#pragma omp critical` equivalent)
- barrier synchronization, upon which you can built reductions
- reduction, *but only within a single thread block*

	OpenMP	CUDA
atomic accumulation	pragma `atomic`	atomicAdd
mutual exclusion	pragma `critical` or `omp_lock_t`	
barrier	pragma `barrier`	cooperative thread group
reduction	pragma `reduction`	only within a thread block, or DIY with barrier

Atomic accumulations

- consider the following (trivial) example

```
1 int * a;
2 cudaMallocManaged(&a, sizeof(int) * nb * bs);
3 for (i=0; i<nb*bs; i++) a[i] = 1;
4 sum<<<nb,bs>>>(a);
5 cudaDeviceSynchronize();
6 printf("sum = %d\n", a[0]);
```

- the goal is to guarantee it always prints the sum of all elements in the array (= `nb` × `bs`)
- a race-prone version

```
1 __global__ void f(int * a) {
2   int i = thread id;
3   if (i > 0) a[0] += a[i];
4 }
```

Atomic accumulations

- atomic accumulations are supported by the hardware and CUDA API
  - `atomicAdd(`p`, `x`)` $\approx$

```
1  #pragma omp atomic
2    *p += x
```

    in OpenMP
  - search the CUDA toolkit documentation for "atomicAdd"
- there are other primitives, such as compare-and-swap
- fix our example

```
1  __global__ void f(int * a) {
2    int i = thread id;
3    if (i > 0) atomicAdd(&a[0], a[i]);
4  }
```

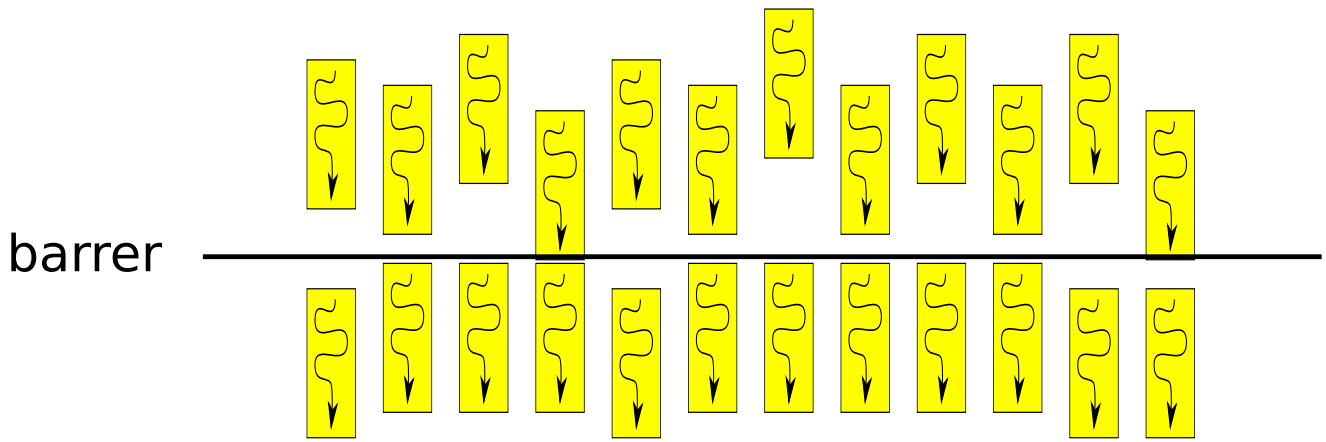
A working version of COO SpMV

```
1  __global__ spmv_dev(A, x, y) {
2    k = thread id;
3    if (k < nnz) {
4      i,j,Aij = A.elems_dev[k];
5      atomicAdd(&y[i], Aij * x[j]);
6    }
7  }
```

- make sure `A.elems_dev`, `x` and `y` point to device memory (not shown)
- note: CSR is simpler to work with if you don't parallelize within a row

Barrier synchronization

- barrier is a mechanism to ensure "all threads reached a point"
- useful to ensure changes made by a thread is visible all others
- CUDA used to support barriers only within a single thread block (`__syncthreads()`)
- it now supports barriers for all threads (C. Cooperative Groups)



Cooperative groups (1)

- (important) when using the following features, launch a kernel by

```
1 void * args[] = { a0, a1, ... };
2 cudaLaunchCooperativeKernel((void *)f, nb, bs, args);
```

  instead of the ordinary

```
1 f<<<nb,bs>>>(a0, a1, ...);
```

- common setup

```
1 #include <cooperative_groups.h>
2 namespace cg = cooperative_groups; // save typing
```

Cooperative groups (2)

- data representing a group

```
1 cg::grid_group g = cg::this_grid(); // all threads
```

```
1 cg::thread_block g = cg::this_thread_block(); // thread block
```

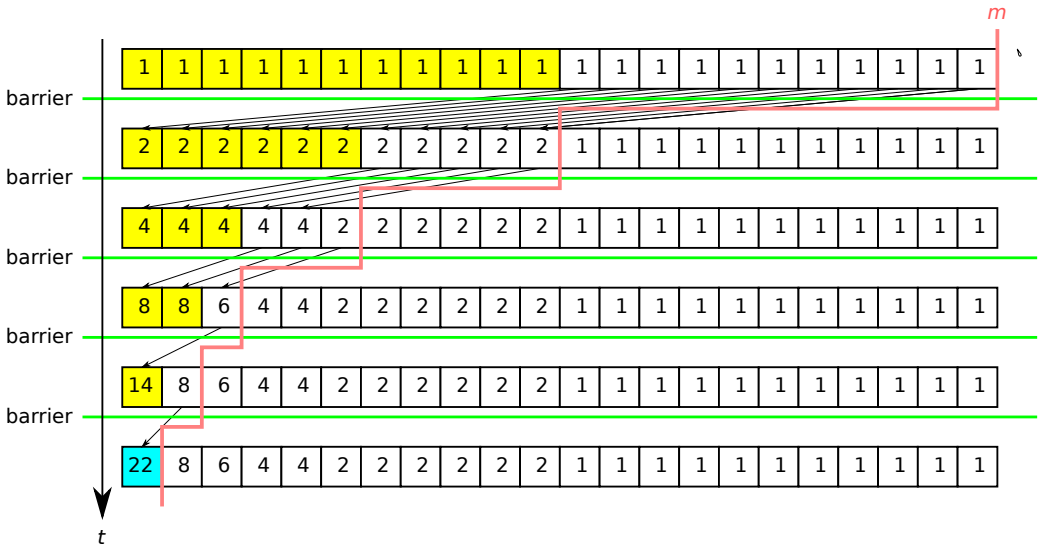
- barrier synchronization

```
1 g.sync(); // barrier all theads in g participate
```

- group actually provides a cleaner way to know thread ID and number of threads

```
1 unsigned long long idx = g.thread_rank(); // my ID in g
2 unsigned long long nth = g.size(); // num threads in g
```

Building reduction on barrier




```
1  __global__
2  void sum(double * c, long n) {
3    // return c[0] + .. + c[n-1]
4    cg::grid_group g = cg::this_grid();
5    ull i = g.thread_rank();
6    ull h; // ull: unsigned long long
7    for (long m = n; m > 1; m = h) {
8      h = (m + 1) / 2;
9      if (i + h < m) c[i] += c[i + h];
10     g.sync();
11   } }
```

- invariant: "`sum(c[0:`m`])` is *the* sum", it repeats halving m
- note: it may not be most efficient; reducing values within a single block first may be better

CUDA shared memory

- CUDA programs can allocate a "shared memory" to each thread block

```
1 f<<<nb,bs,S>>>(...);
```

- from CUDA program's perspective, it is a memory *only shared within a thread block* and *only active during the thread block's lifetime*
  - the term *shared memory* is a misnomer, IMO; ordinary memory you allocate via `cudaMalloc` *is* shared by all threads
  - local memory or something will be a more appropriate name
- physically, it is a cache-like memory faster than global memory
- each SM has a fixed amount of shared memory (A100 : 164KB)

$$S\times \leq \text{shared memory per SM}$$

Accessing CUDA shared memory

- specify the shared memory size on a kernel call and a kernel accesses it by declaring variables or arrays with `__shared__`

```
1 __shared__ int a[n];
2 __shared__ char b[m];
```

- if the data size (n or m above) is not a compile-time constant, obtain the starting address of the shared memory by

```
1 extern __shared__ char whatever[];
```

- it's your responsibility to use appropriate part of it. e.g.,

```
1 int * a = (int*)whatever;
2 char * b = (char *)&a[n];
```

- shared memory is a way to efficiently communicate among threads within a block
- GPU has nowadays processor-managed caches, so how crucial it is to performance is somewhat changing over time

Contents

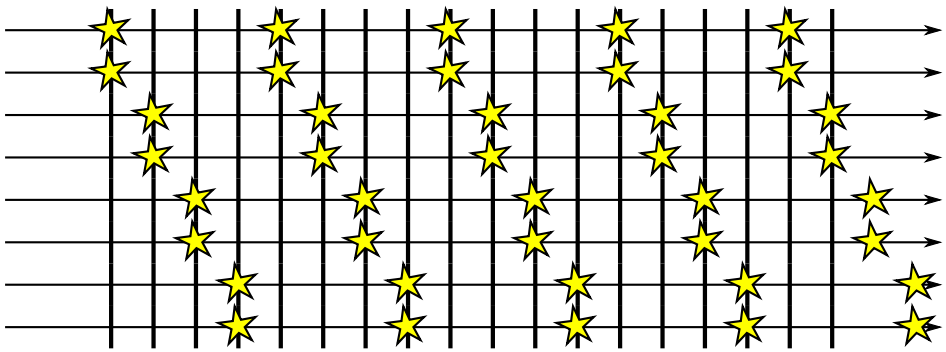
Choosing a good block size for performance

- the question is, when you create a number of, say 10000, threads, how you divide them into thread blocks?

```
1    f<<<1, 10000>>>(x, y, z, ...);
2    f<<<10, 1000>>>(x, y, z, ...);
3    f<<<100, 100>>>(x, y, z, ...);
4    f<<<1000, 10>>>(x, y, z, ...);
5    f<<<10000, 1>>>(x, y, z, ...);
```

  and countless other ways ...

- the goal is to run an enough number of threads *simultaneously* so they *utilize* the hardware capacity of an SM



- to this end, let's understand what a GPU actually does, given a thread block size and the number of blocks

Parallelism within an SM

consists of three levels

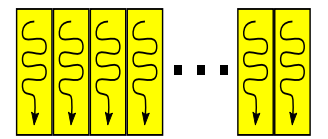
thread $\subset$ warp $\subset$ thread block $\subset$ SM

Parallelism within an SM

consists of three levels

thread $\subset$ warp $\subset$ thread block $\subset$ SM

- a group of 32 CUDA threads makes a *warp*

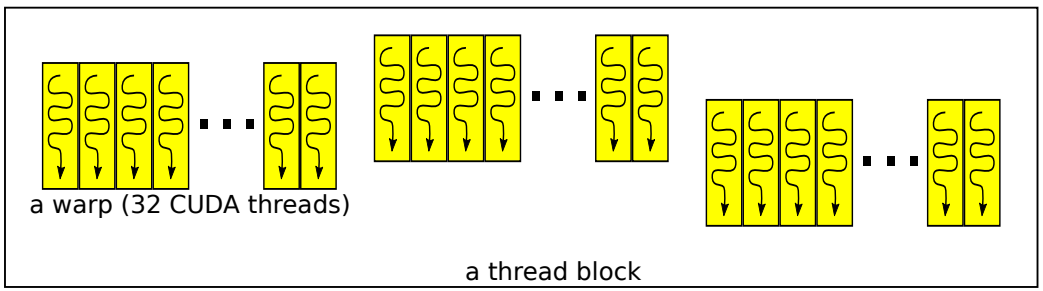


Parallelism within an SM

consists of three levels

thread $\subset$ warp $\subset$ thread block $\subset$ SM

- a group of 32 CUDA threads makes a *warp*
- a group of $\lceil bs/32 \rceil$ warps makes a *thread block*

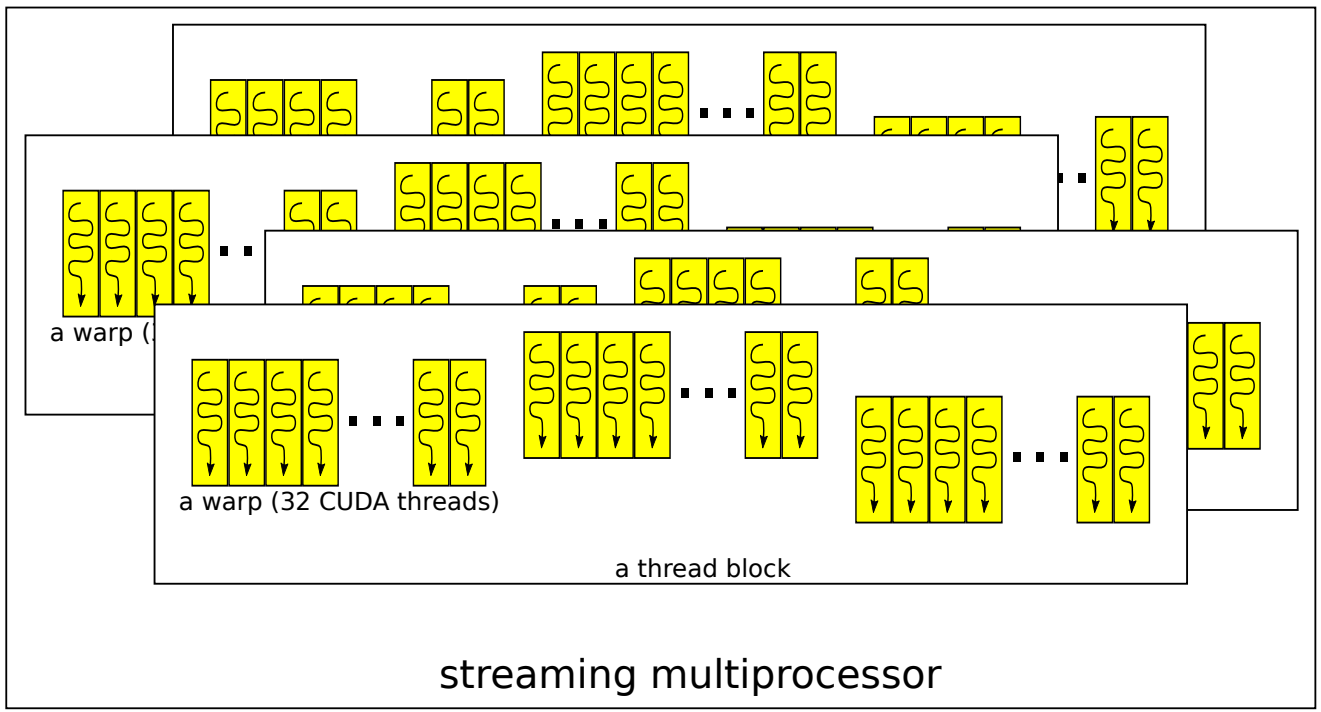


Parallelism within an SM

consists of three levels

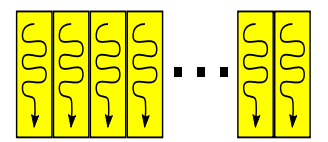
thread $\subset$ warp $\subset$ thread block $\subset$ SM

- a group of 32 CUDA threads makes a *warp*
- a group of $\lceil bs/32 \rceil$ warps makes a *thread block*
- and there are multiple thread blocks executing *simultaneously* on a single SM



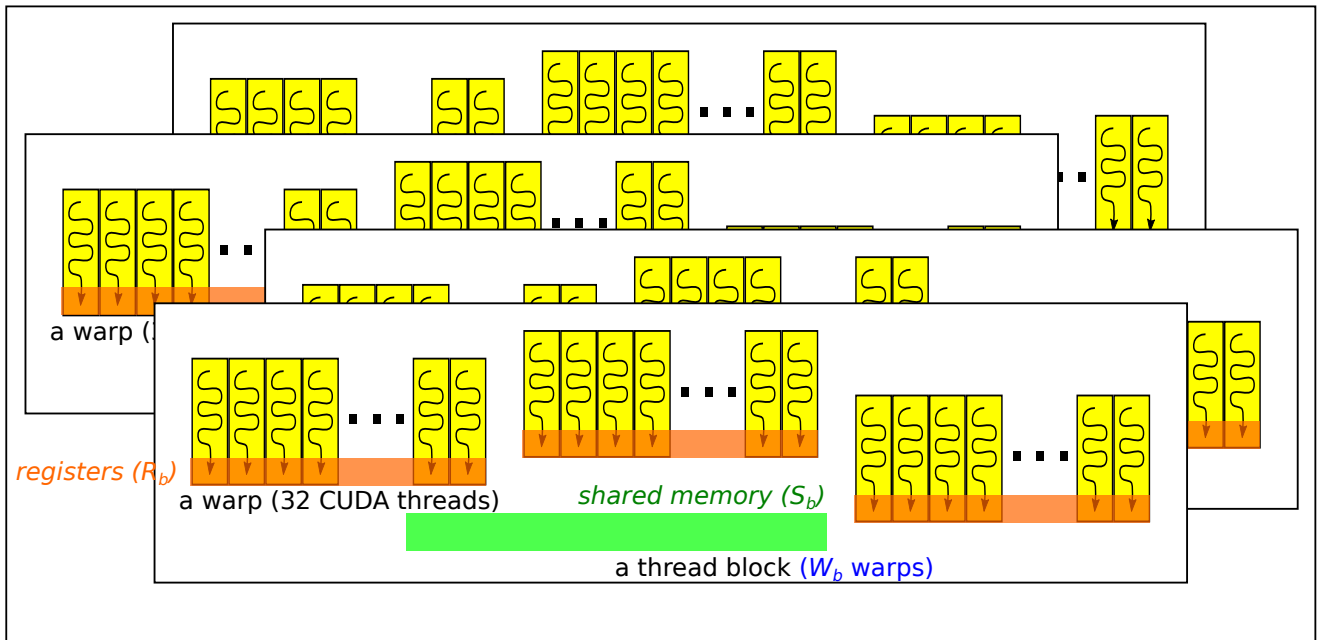
Warps

- a warp is the unit of *instruction execution*
- 32 threads in a single warp share an instruction pointer (a warp $\approx$ a CPU thread executing 32-way SIMD instructions)
- at every cycle, an SM selects a few (actually, ≤ 2) warps and execute them
- $\Rightarrow$
  - there is rarely a point in making $bs < 32$ or not a multiple of 32 (ramainder threads consume resources but perform no useful work)
  - you want to make 32 threads branch in the same way (avoid *warp divergence*)



Thread blocks

- a thread block is *the unit of dispatching to an SM*
- conceptually, a kernel launch `f<<<`nb`, ...>>>(x, y, ...)` puts nb blocks in a queue, which GPU dispatches to SMs, one block at a time
- once a block starts running, they stay on the SM *until it finishes* and occupies *registers* and *shared memory* throughout
- $\Rightarrow$ the number of blocks *simultaneously* running on an SM is limited by registers and shared memory a thread block uses

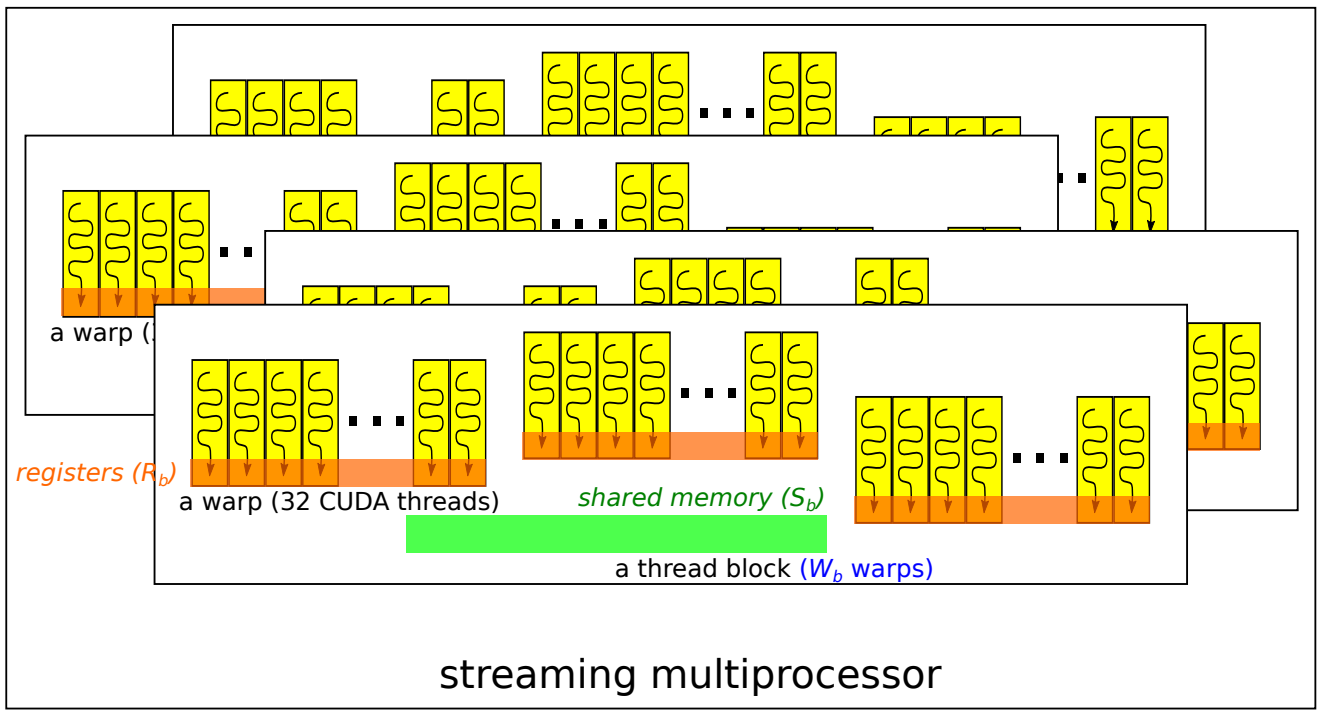


Registers and shared memory

- registers
  - hold *local and temporary variables* of threads
  - the size is determined by your program and the compiler
- shared memory
  - can be allocated when launching a kernel by

```
1  f<<<nb, bs, S_b>>>(x, y, z, ...)
```

  and is shared within a thread block



Hardware limits

- all numbers are per SM
- A100 (compute capability 8.0)

registers	65336 × 32 bits
shared memory	164 KB
warps that can simultaneously run	64
thread blocks that can simultaneously run	32

- V100 (compute capability 7.0)

registers	65336 × 32 bits
shared memory	96 KB
warps that can simultaneously run	64
thread blocks that can simultaneously run	32

Putting them together

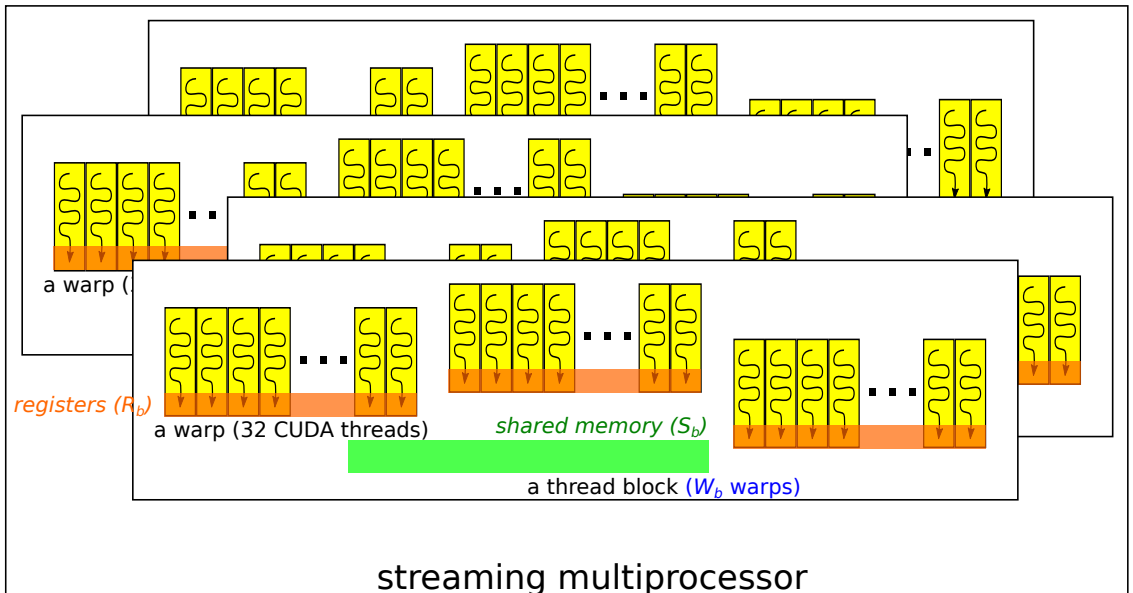
blocks that will simultaneously run on an SM

given (from the programmer or the compiler)

- T_b : the number of threads per block,
- S_b : shared memory size per block, and
- R_1 : registers per thread,

calculate various resources *per block*

- *warps* per block : $W_b = \lceil T_b/32 \rceil$
- *registers* per block : $R_b = 32R_1 \times W_b$



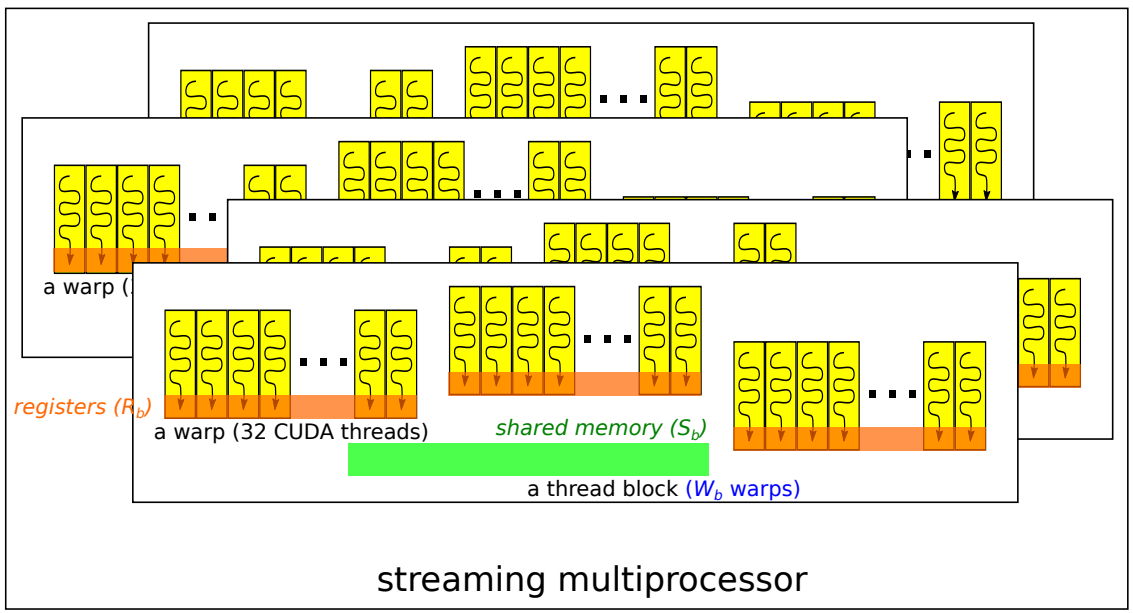
Putting them together

- the number of blocks that simultaneously run on an SM (nb)

$$\begin{aligned} nb &= \min(\lfloor 65536/R_b \rfloor, \lfloor 164\text{K}/S_b \rfloor, \lfloor 64/W_b \rfloor, 32) \\ &= \min(\lfloor 2048/(R_1 W_b) \rfloor, \lfloor 164\text{K}/S_b \rfloor, \lfloor 64/W_b \rfloor, 32) \end{aligned}$$

- the number of warps simultaneously run on an SM (nw)

$$\begin{aligned} nw &= W_b \cdot nb \\ &= W_b \cdot \min(\lfloor 2048/(RW_b) \rfloor, \lfloor 164\text{K}/S_b \rfloor, \lfloor 64/W_b \rfloor, 32) \end{aligned}$$



Takeaways (often good thread block sizes)

$$W_b \cdot \min(\lfloor 2048/(R_1 W_b) \rfloor, \lfloor 164\text{K}/S_b \rfloor, \lfloor 64/W_b \rfloor, 32)$$

if we ignore factors that come from R_1 and S_b, a guideline is to run the maximum 64 warps simultaneously and it can be accomplished by

- putting at least two warps in a block (so $64/W_b \leq 32$) and
- chooseing the number of warps per block that divides 64
- that is, $W_b = 2, 4, 8, 16, 32$ (or $T_b = 64, 128, 256, 512, 1024$)

Remarks

- 64 warps is merely *an* upper bound that
  - may not be necessary to get the maximum performance (e.g., floating point performance, whose limit is 2-warp (= 64) FMAs per cycle) and
  - may not be achievable due to other constraints (registers and shared memory)
- the above takeaway is a rule of thumb to eliminate bad thread block sizes

Occupancy calculator

- NVIDIA used to provide a simple Excel to give you how many warps can run simultaneously given block size (T_b), shared memory per block (S_b), and registers per thread (R_1)
- a small web page doing the same at `https://xmartlabs.github.io/cuda-calculator/`