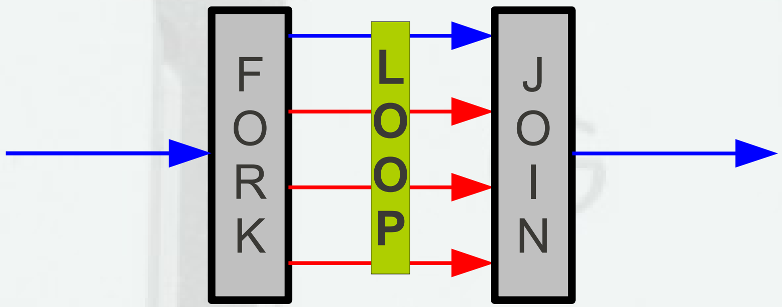
A Brief Introduction to CUDA

2
Types of Parallelism

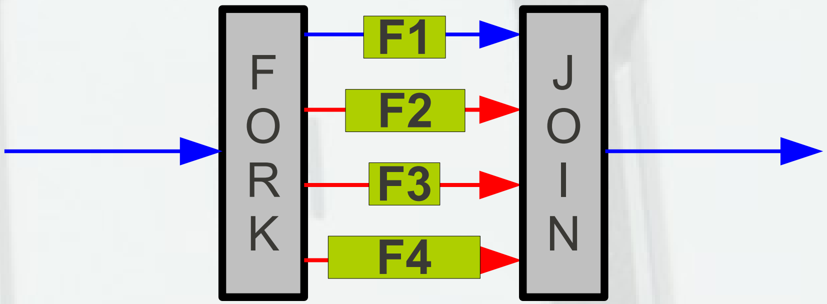
DATA PARALLELISM



Cores execute ***same instructions***

...but operate on ***different data.***

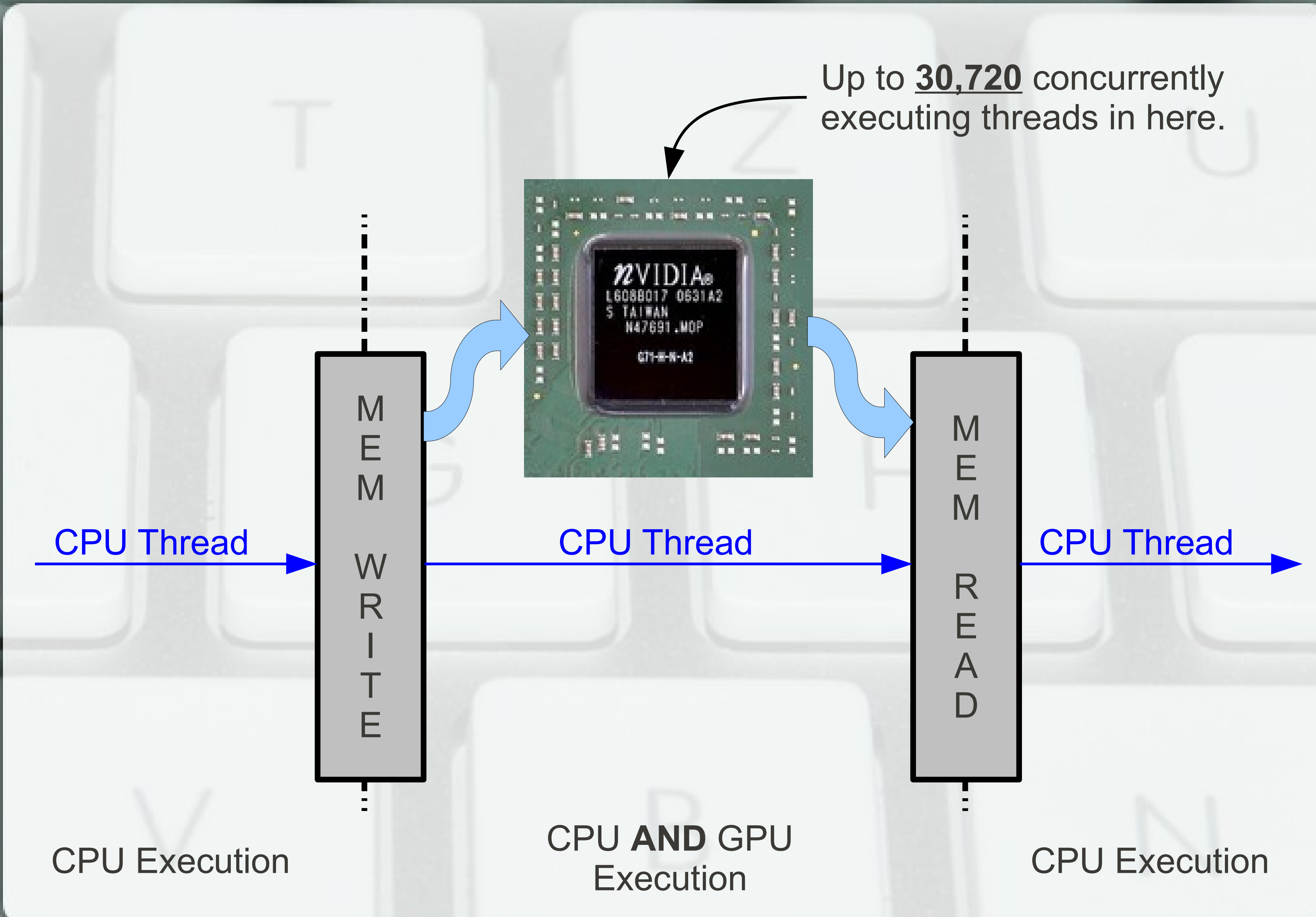
FUNCTIONAL PARALLELISM



Cores execute ***different instructions***

...and *can* <u>read</u> same data &
should <u>write</u> different data.

BASIC IDEA

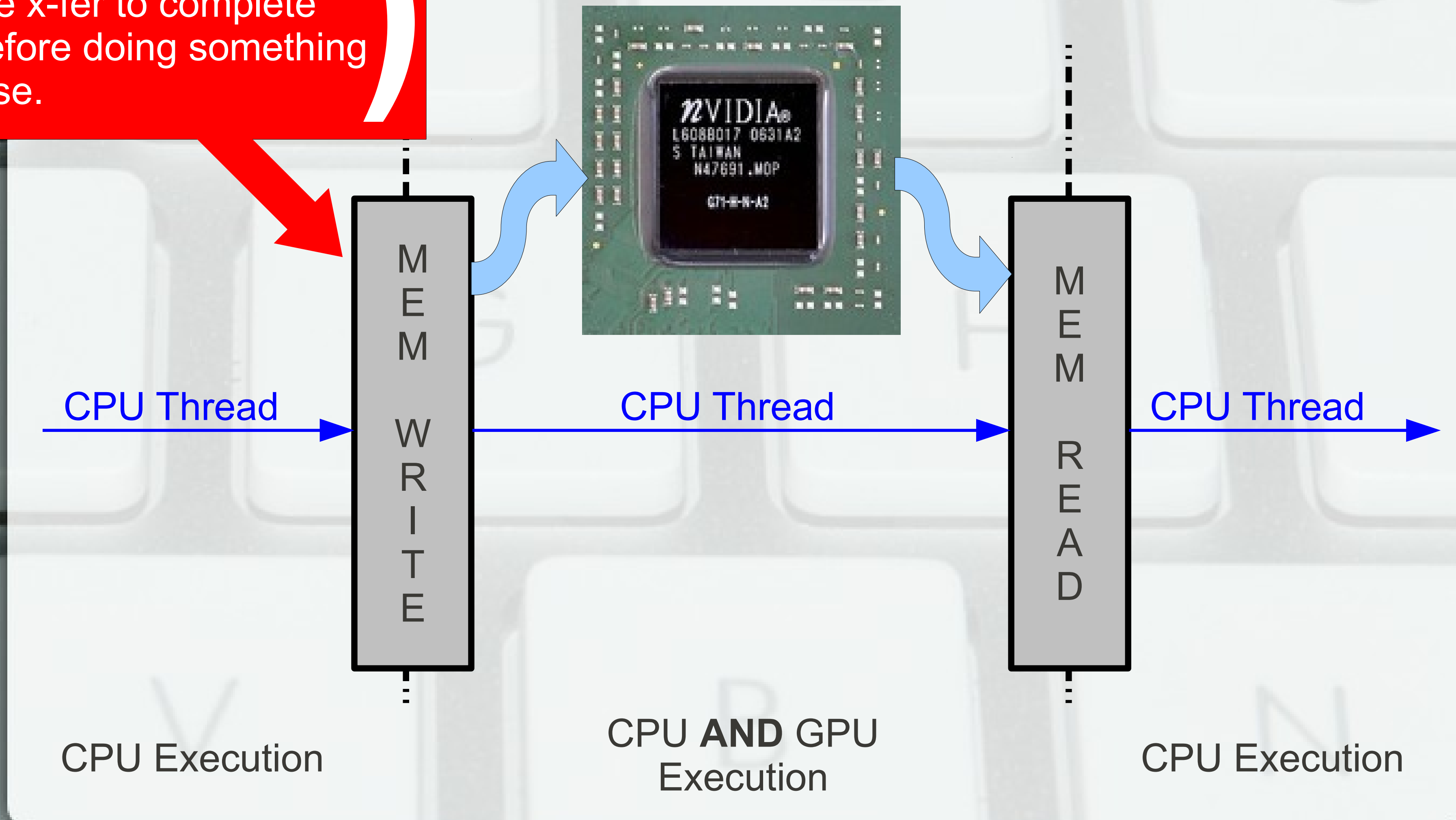


Up to **30,720** concurrently executing threads in here.
NVIDIA
MEM WRITE
MEM READ
CPU Thread
CPU Thread
CPU Thread
CPU Execution
CPU **AND** GPU Execution
CPU Execution

BASIC IDEA

CPU → GPU
Memory transfers are ***blocking***.

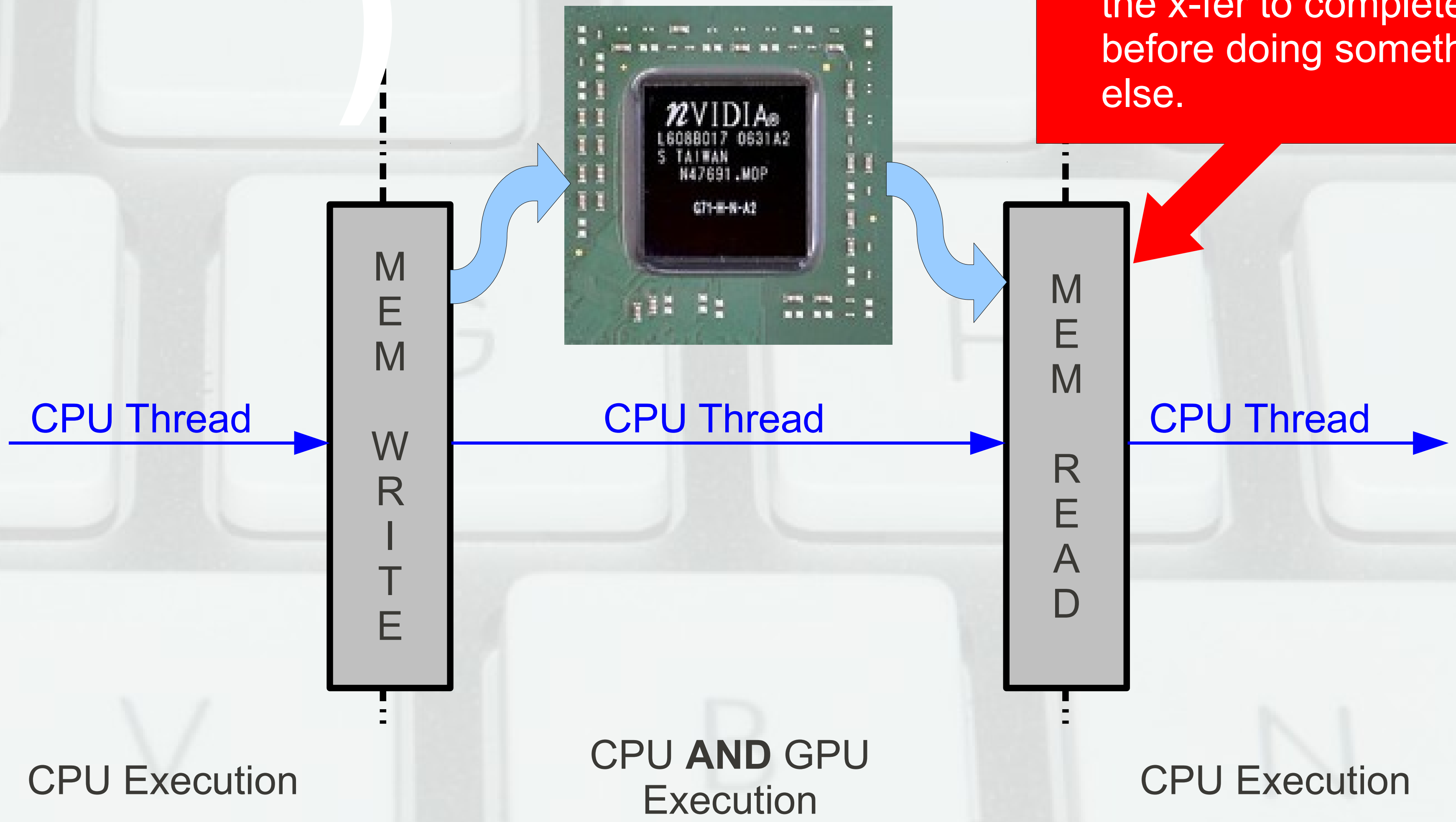
(The CPU must wait for the x-fer to complete before doing something else.)

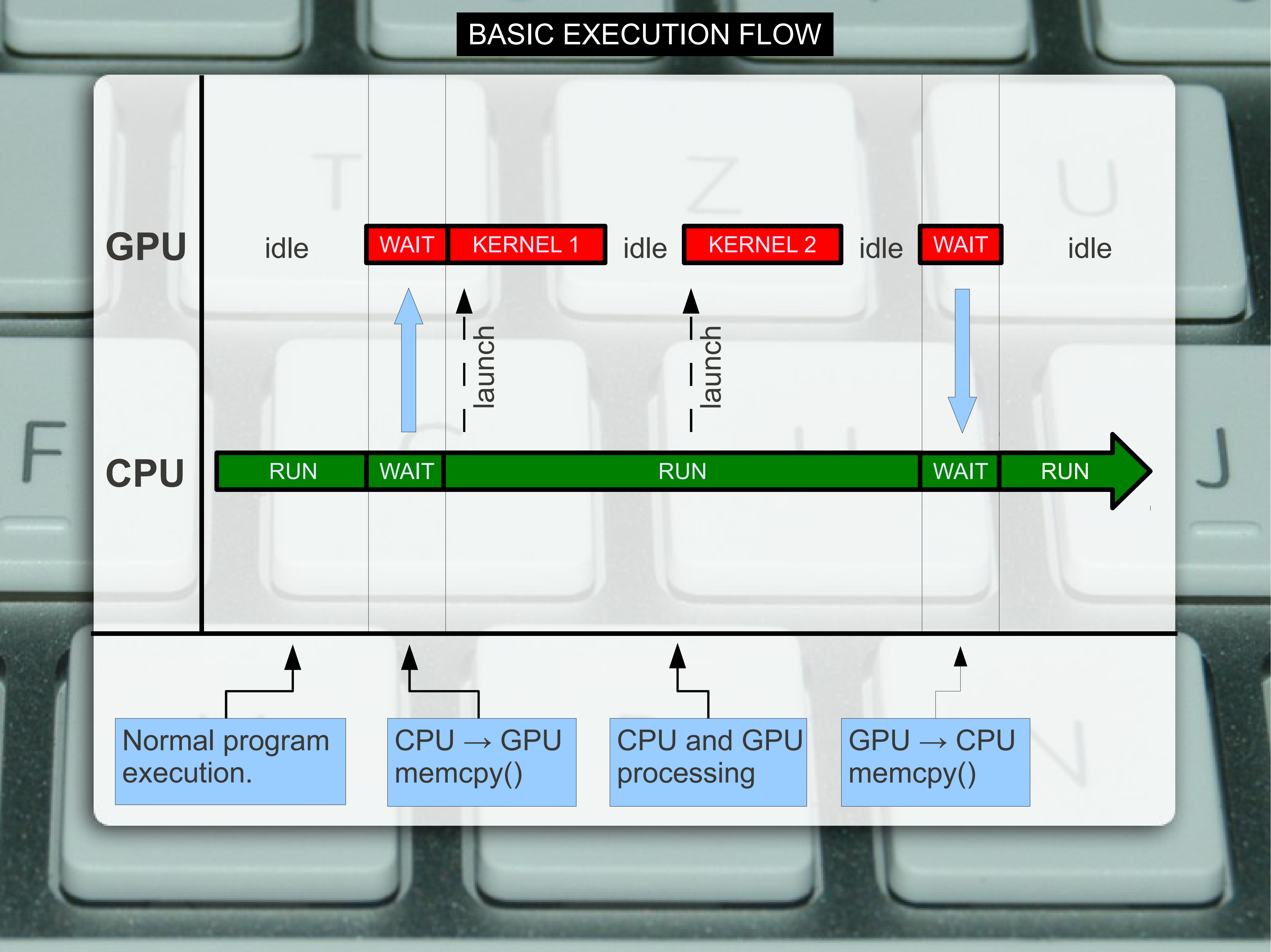


BASIC IDEA

GPU → CPU
Memory transfers are ***also blocking***.

The CPU must wait for the x-fer to complete before doing something else.





BASIC EXECUTION FLOW
GPU
idle
WAIT
KERNEL 1
idle
KERNEL 2
idle
WAIT
idle
launch
launch
CPU
RUN
WAIT
RUN
WAIT
RUN
Normal program execution.
CPU → GPU memcpy()
CPU and GPU processing
GPU → CPU memcpy()

BASIC EXECUTION FLOW

KERNEL 1

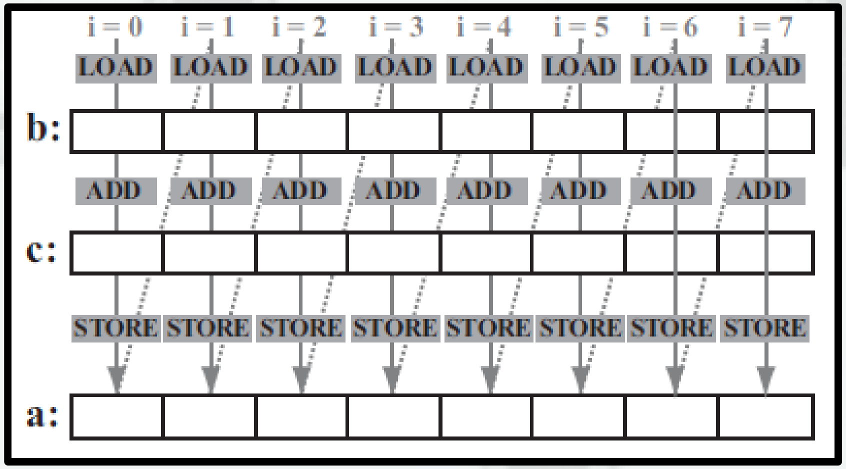
Kernels consist of <u>MANY</u> parallel threads

- Thread IDs are used to index into arrays
- (Most) Array accesses must be *<u>coalesced</u>*.

The Most Basic Example

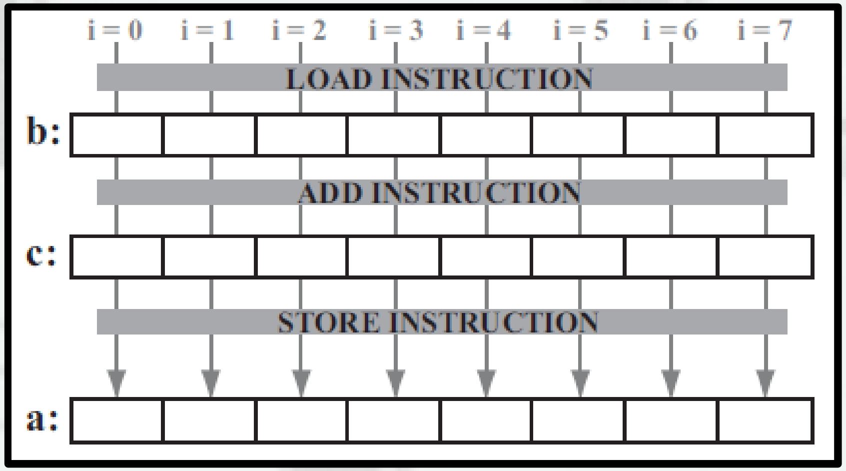
CPU

```
void
func (float* a, float* b, float* c)
{
     int i;
     for (i=0; i<8; i++) {
          a[i] = b[i] + c[i];
     }
}
```



GPU Kernel

```
__global__ void
kernel (float* a, float* b, float* c)
{
     int i = threadIdx.x;
     a[i] = b[i] + c[i];
}
```



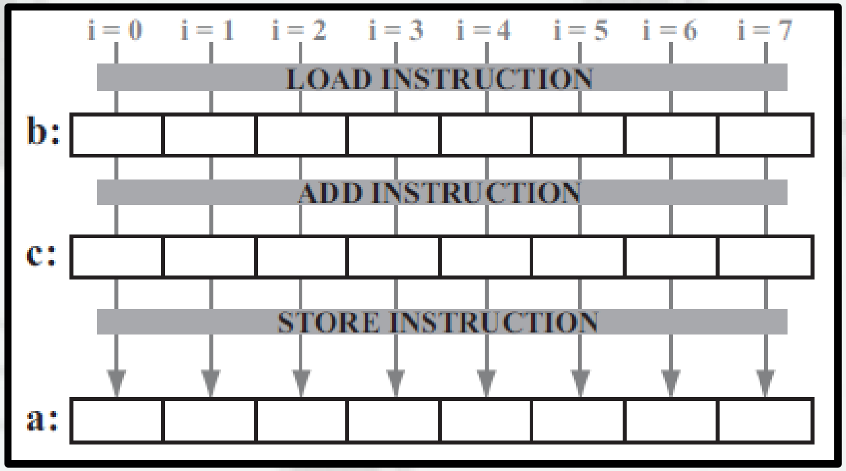
The Most Basic Example

Thread IDs used to index into arrays.

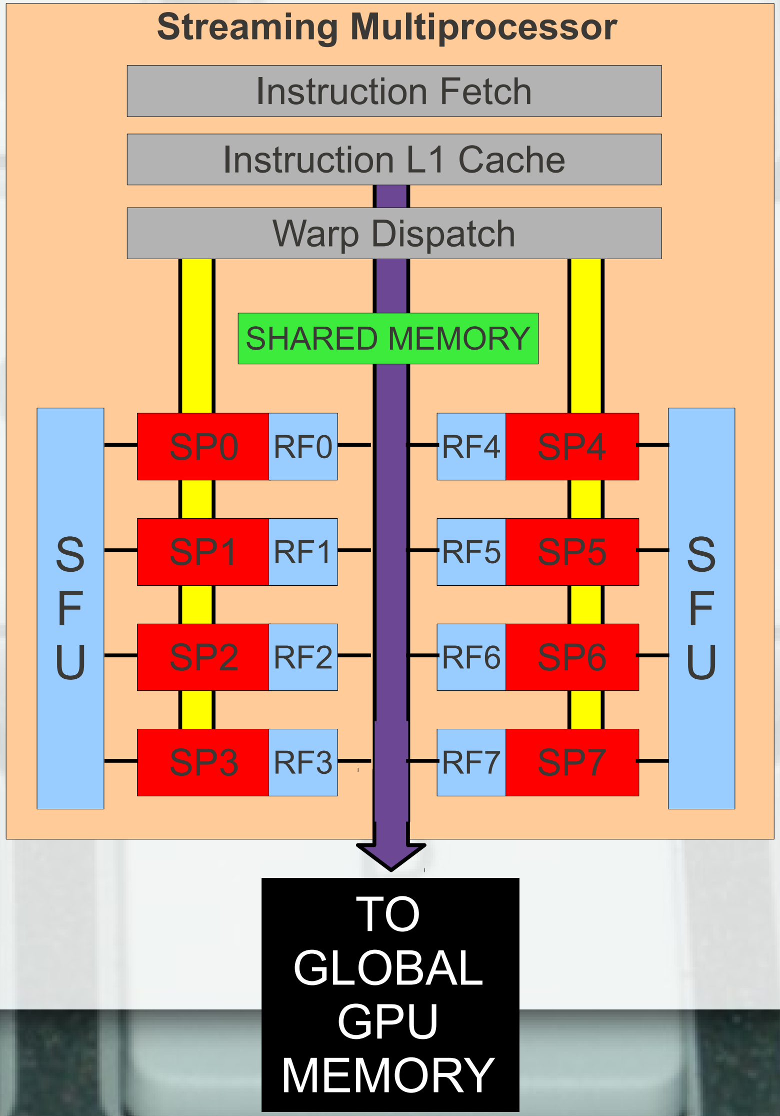
Data reads & writes are *coalesced*.

GPU Kernel

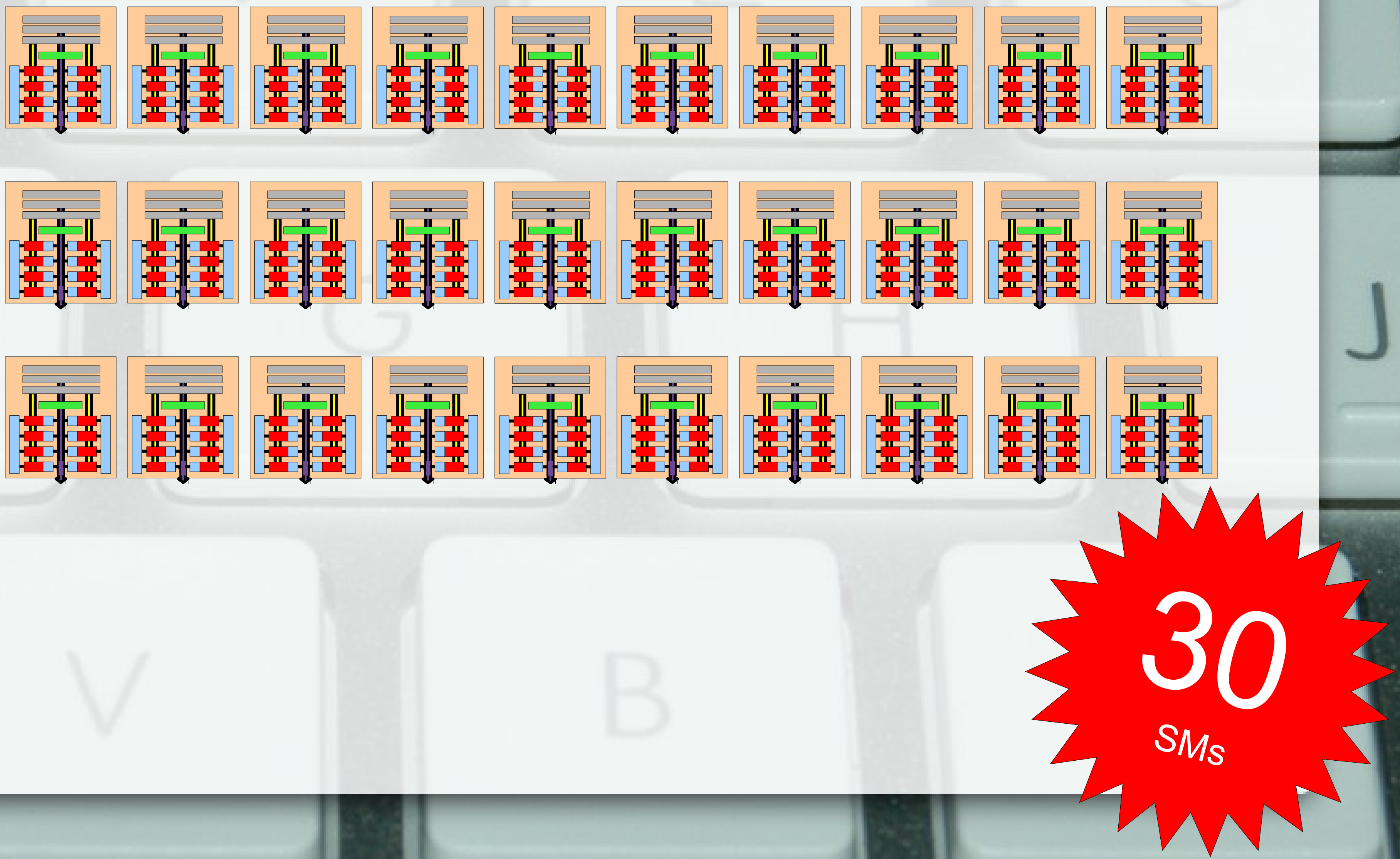
```
__global__ void
kernel (float* a, float* b, float* c)
{
    int i = threadIdx.x;
    a[i] = b[i] + c[i];
}
```



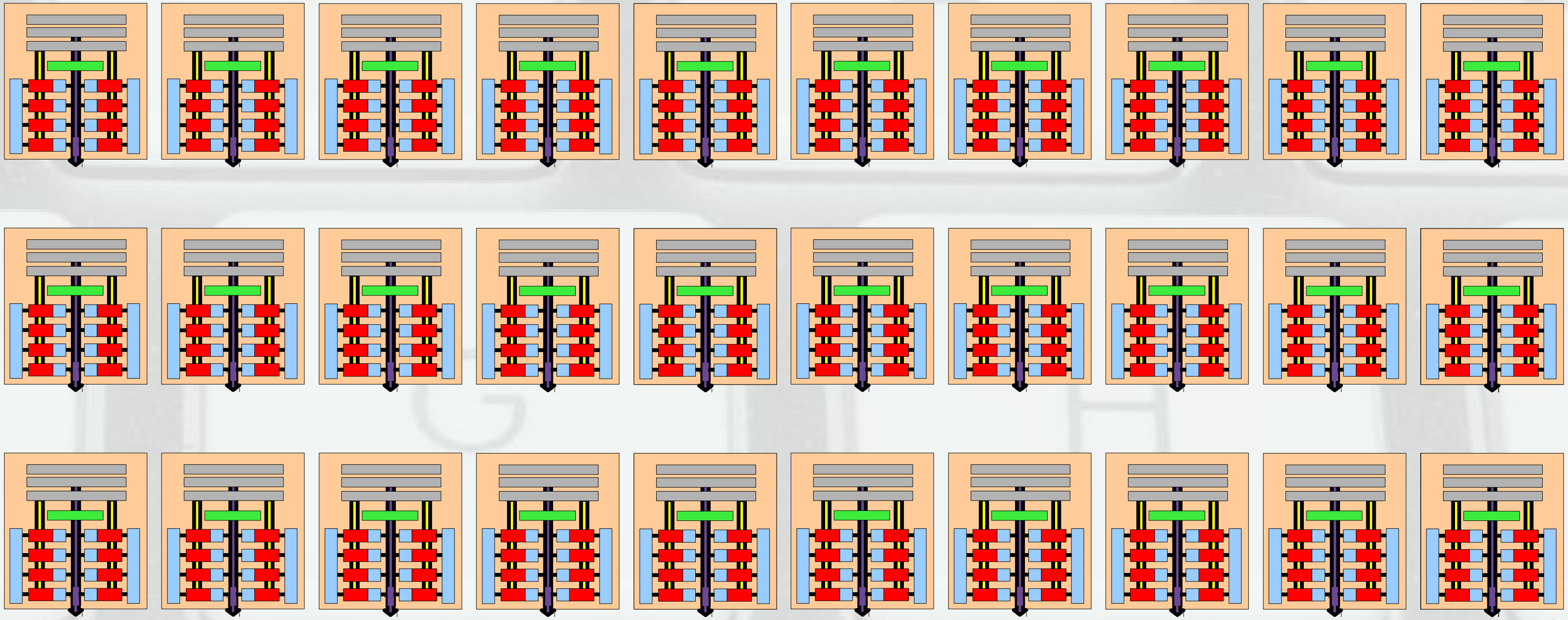
Threading Strategies



Threading Strategies



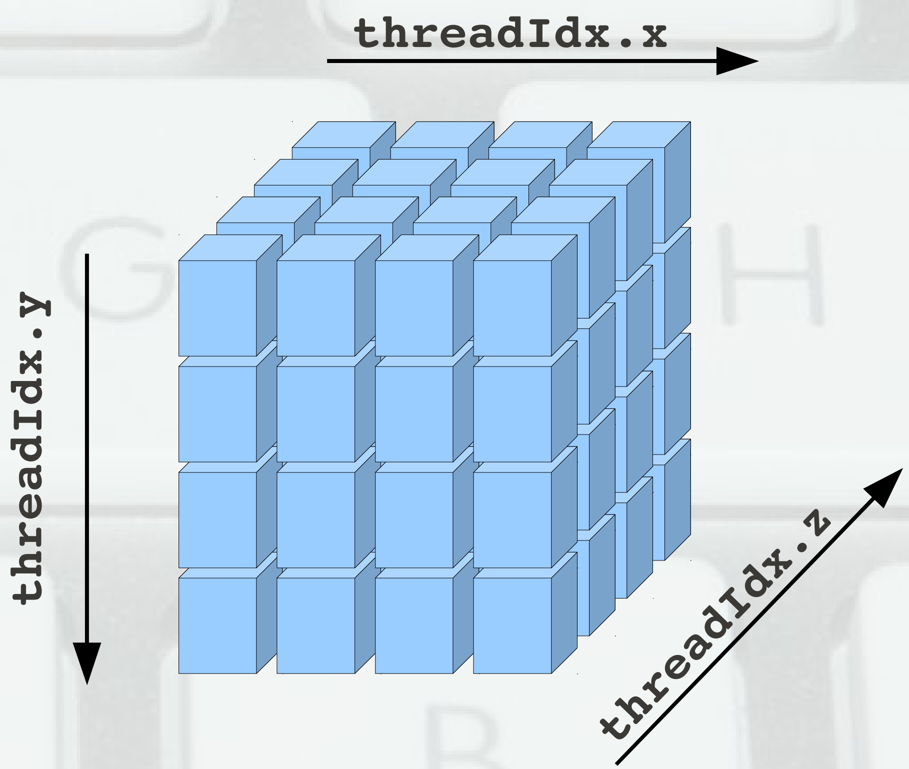
Threading Strategies



*1024 thread capacity per SM. Block assignment subject to SM resource availability.

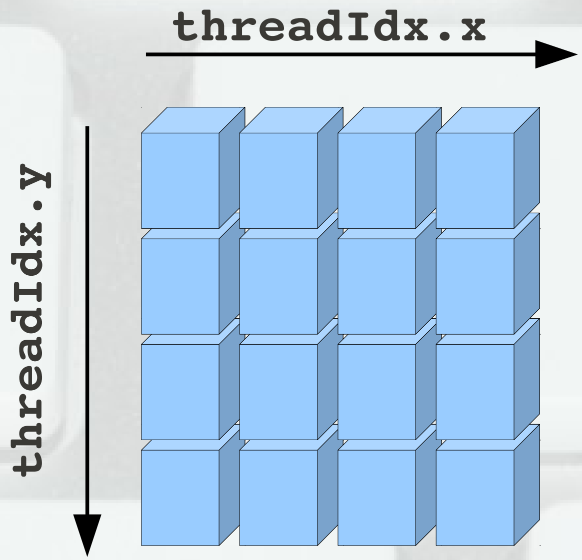
Threading Strategies

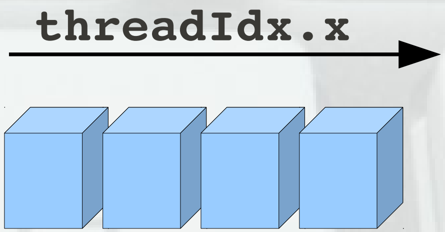
Threads are orgainized into 3D blocks



Threading Strategies

2D and 1D configurations are also possible

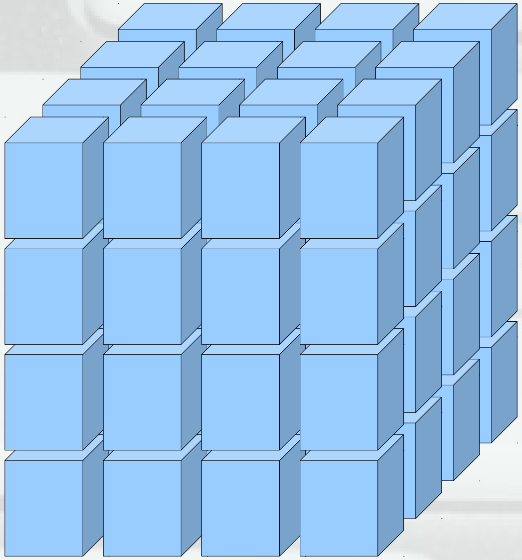




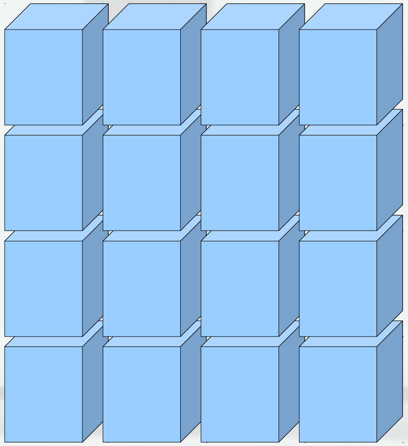
This makes indexing into arrays easier.

Threading Strategies

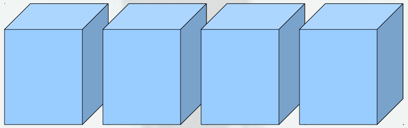
2D and 1D configurations are also possible



Good for 3D data



Good for 2D data



Good for 1D data

Max threads per block is always **512**

Threading Strategies

GPU Kernel – 1D Data

```
__global__ void
kernel (float* a, float* b, float* c)
{
    int i = threadIdx.x;  // thread x coord

    a[i] = b[i] + c[i];
}
```

Threading Strategies

GPU Kernel – 2D Data

```
__global__ void
kernel (float* a, float* b, float* c)
{
    int i = threadIdx.x;  // thread x coord
    int j = threadIdx.y;  // thread y coord

    int d = blockDim.x;   // block x dim

    int index = d.x*j + i;

    a[index] = b[index] + c[index];
}
```

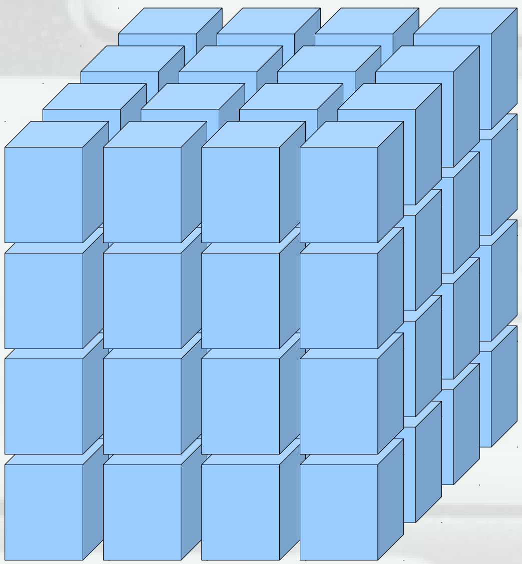
Threading Strategies

GPU Kernel – 3D Data

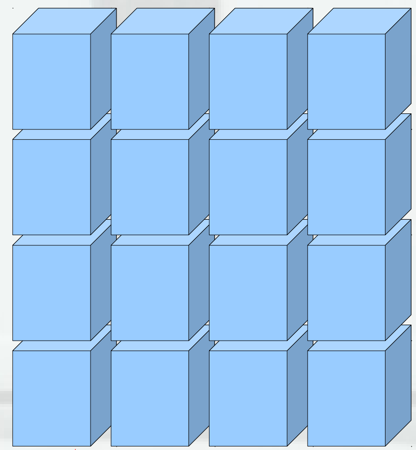
```
__global__ void
kernel (float* a, float* b, float* c)
{
    int i = threadIdx.x;  // thread x coord
    int j = threadIdx.y;  // thread y coord
    int k = threadIdx.z;  // thread z coord

    int2 d;
    d.x = blockDim.x;     // block x dim
    d.y = blockDim.y;     // block y dim

    int index = d.x*d.y*k + d.x*j + i;

    a[index] = b[index] + c[index];
}
```

Threading Strategies

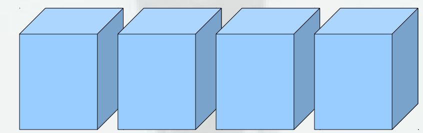
2D and 1D configurations are also possible



Good for 3D data



Good for 2D data



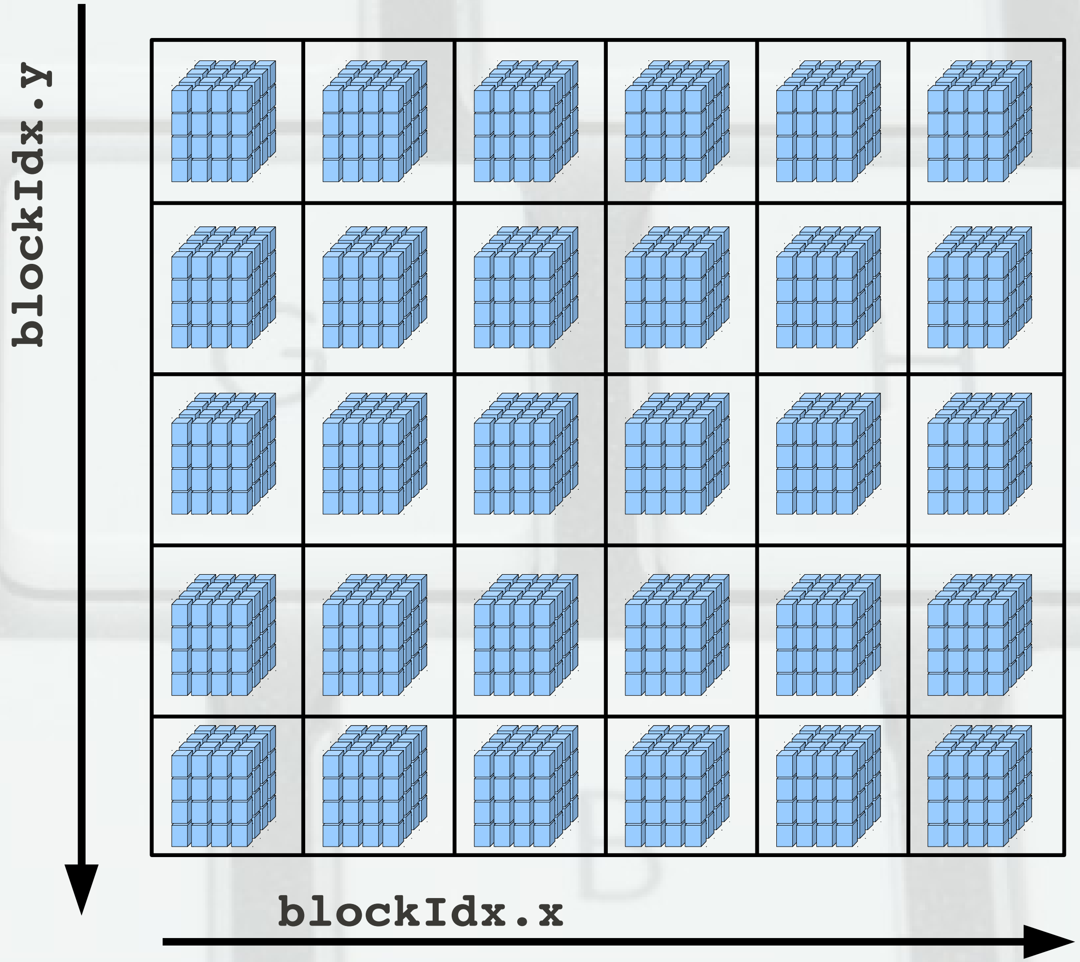
Good for 1D data

What if my input data has **more** than 512 elements?

Max threads per block is always **512**

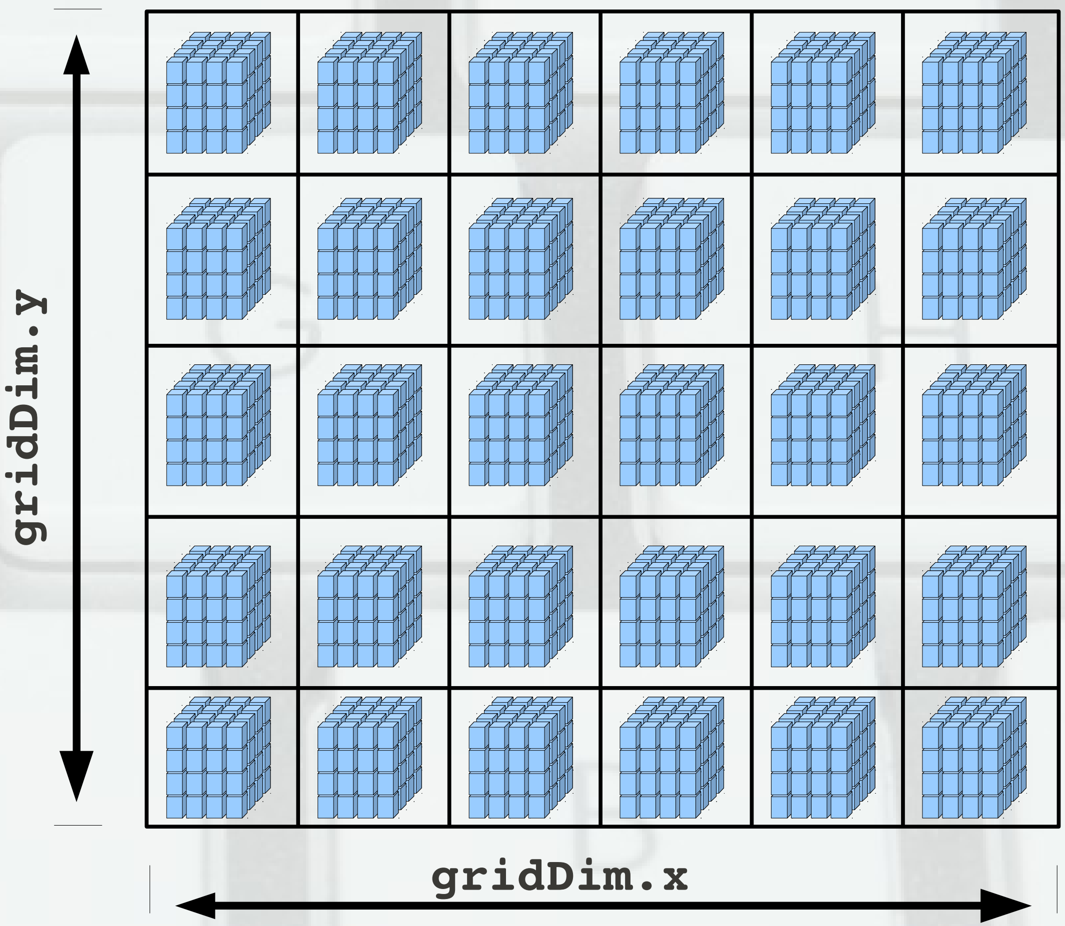
Threading Strategies

Grid of Thread Blocks:



Threading Strategies

Grid of Thread Blocks:



Threading Strategies

GPU Kernel – 1D Data

```
__global__ void
kernel (float* a, float* b, float* c)
{
    int i = threadIdx.x;      // thread x coord

    int block = blockIdx.x;   // block x coord

    int dim = blockDim.x;     // block x dim

    int index = block*dim + i;

    a[index] = b[index] + c[index];
}
```

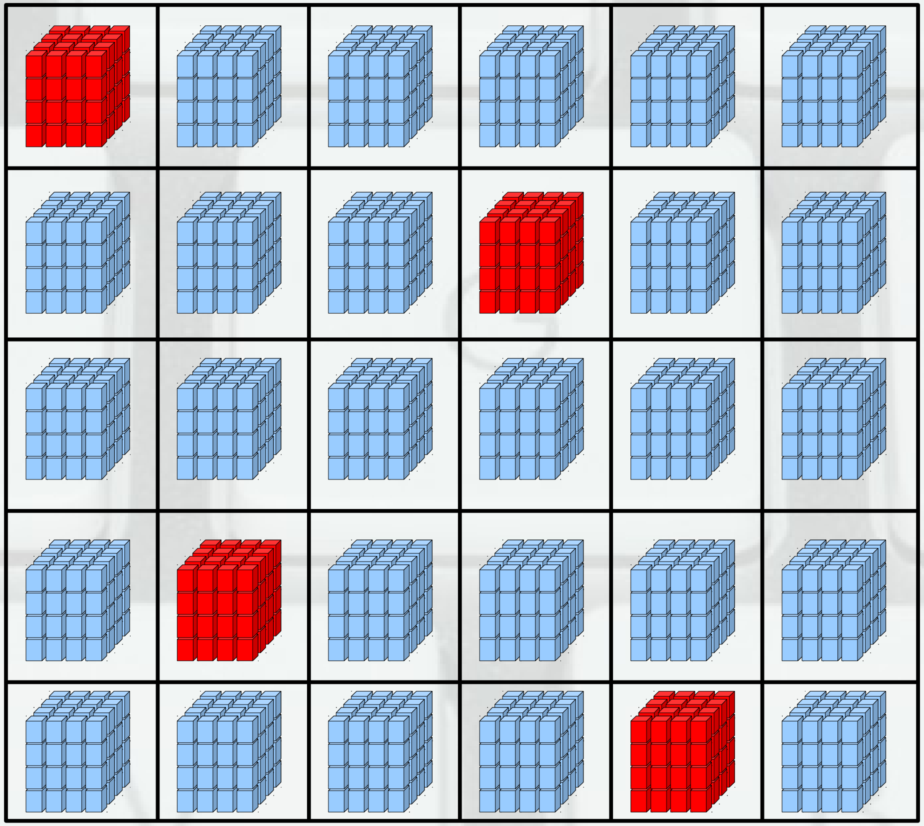
Threading Strategies

GPU Kernel – 2D Data

```
__global__ void
kernel (float* a, float* b, float* c)
{
    int i = threadIdx.x;  // thread x coord
    int j = threadIdx.y;  // thread y coord

    int block_size = blockDim.x * blockDim.y;

    int block_idx =
        blockIdx.y*gridDim.x + blockIdx.x;

    int offset = block_idx*block_size;

    int local_index = blockDim.x*j + i;

    int index = offset + local_index;

    a[index] = b[index] + c[index];
}
```

DATA PARALLELISM

Blocks may be assigned to SMs **out of order.** This is okay.



As a result, there can be no data dependencies between blocks.

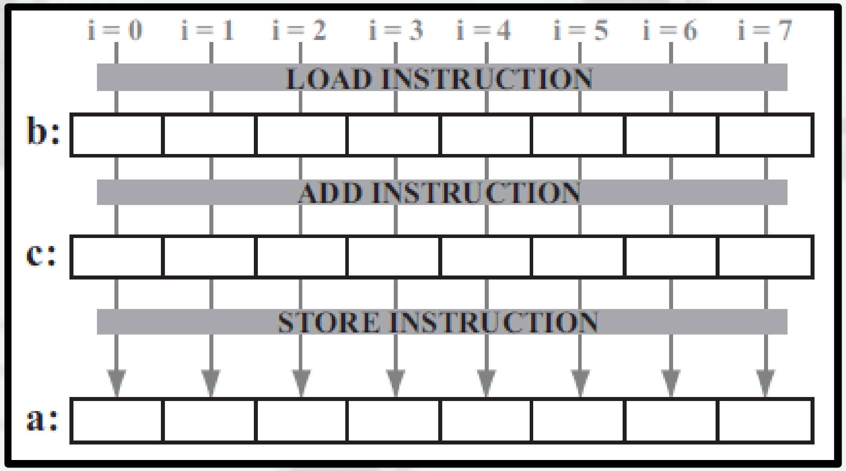
The Most Basic Example

Thread IDs used to index into arrays.

Data reads & writes are *coalesced*.

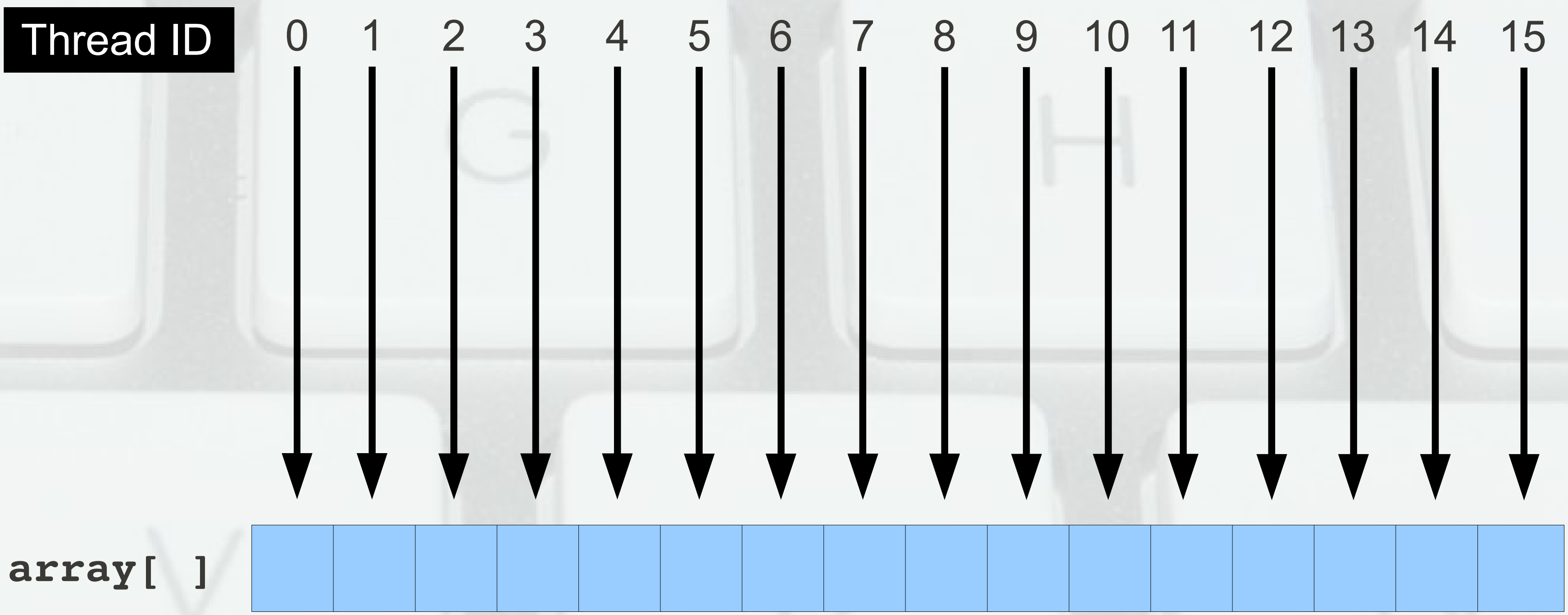
GPU Kernel

```
__global__ void
kernel (float* a, float* b, float* c)
{
    int i = threadIdx.x;
    a[i] = b[i] + c[i];
}
```



Data Coalescence

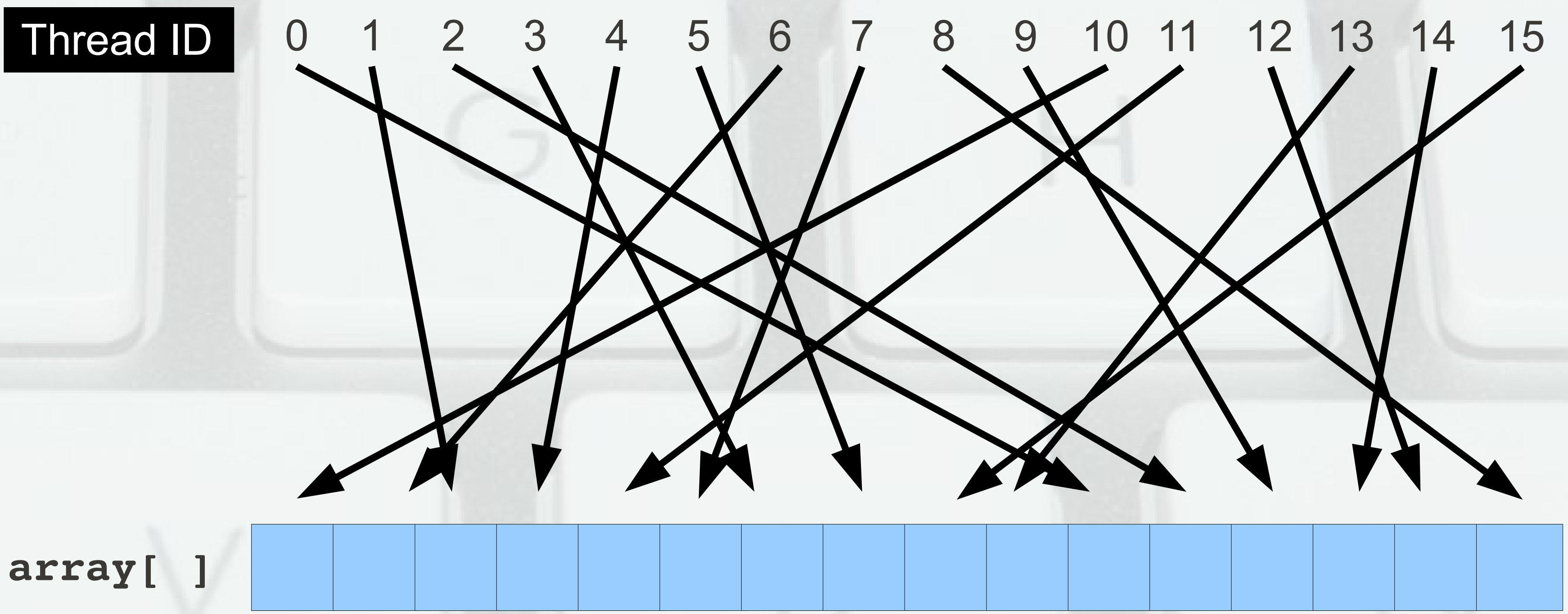
Good ✓



Data Coalescence

Bad ✗

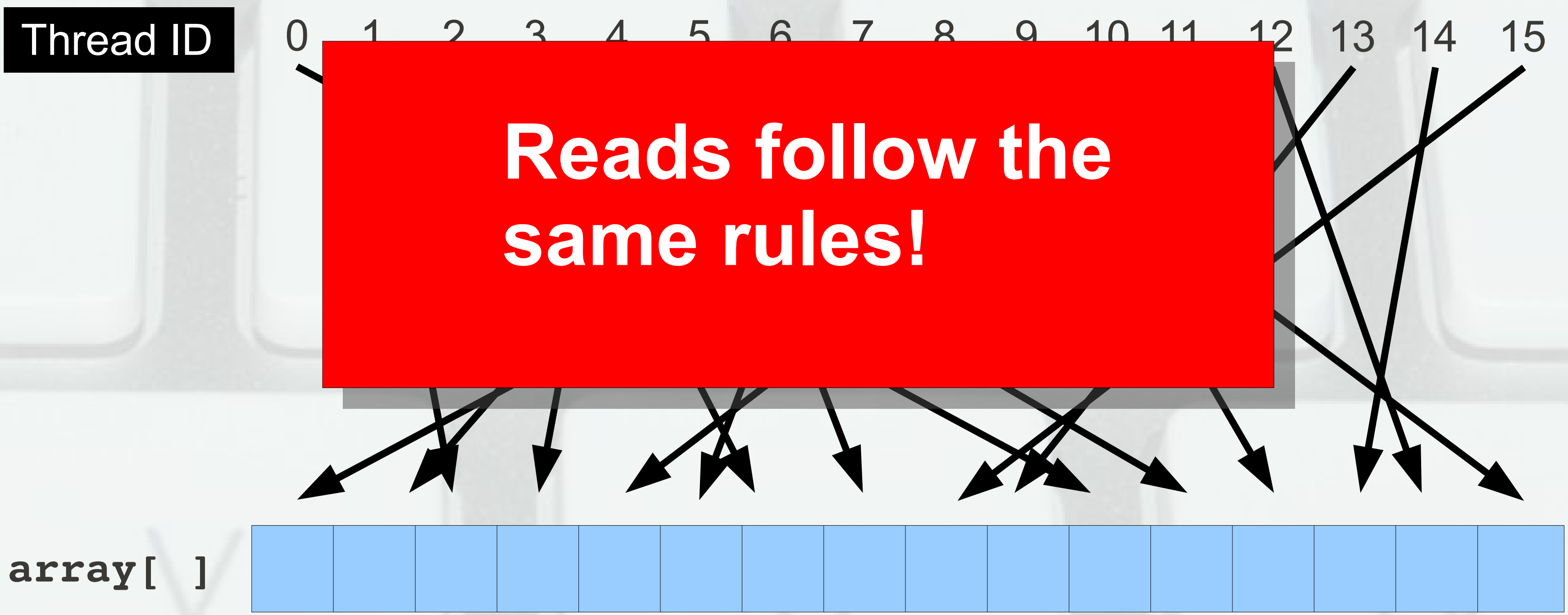
(These writes get serialized)



Data Coalescence

Bad ✘

(These writes get serialized)



Reads follow the same rules!

Essence of Structure

Thread IDs are used to index into data.

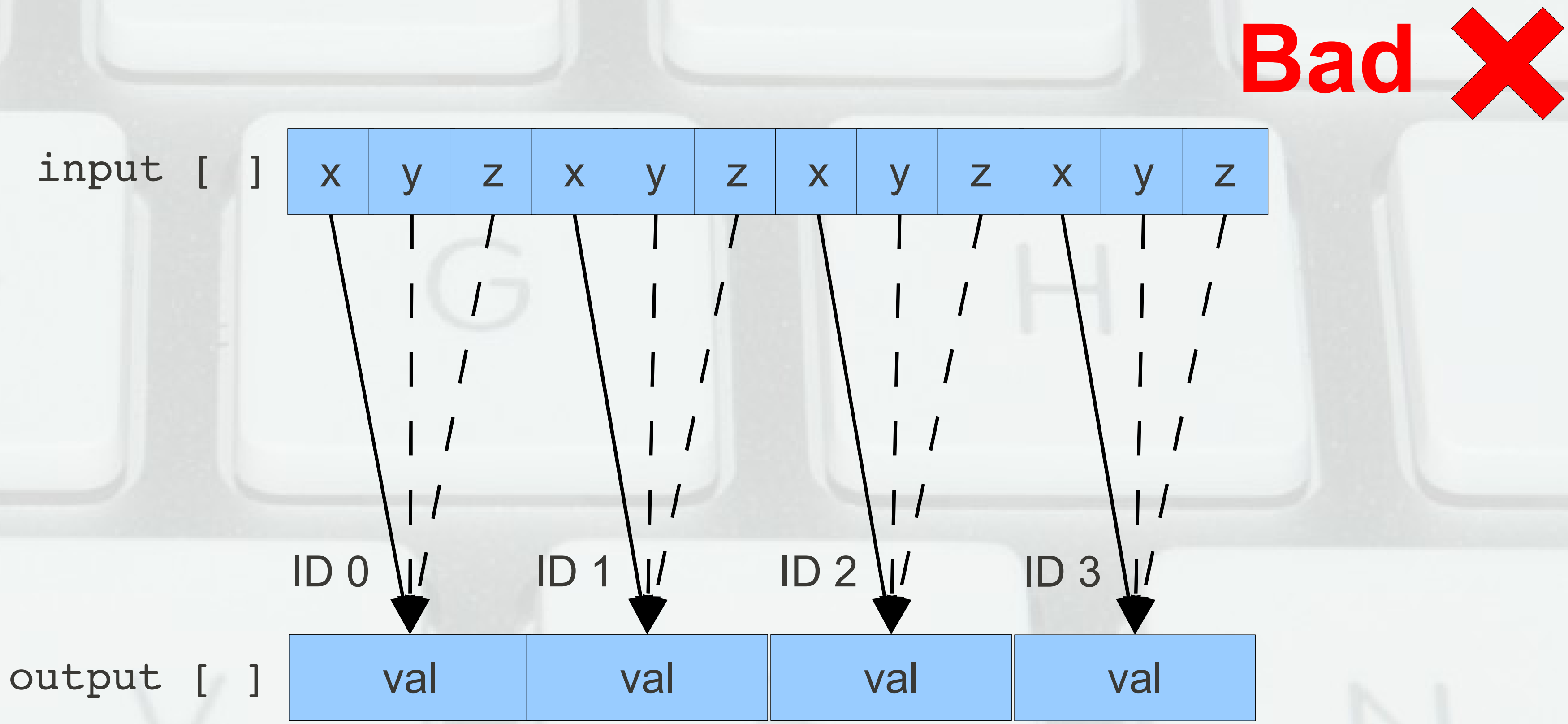
&

Consecutive thread IDs must access consecutive data.

$\therefore$ **Threading scheme is heavily dependent on data structure.**

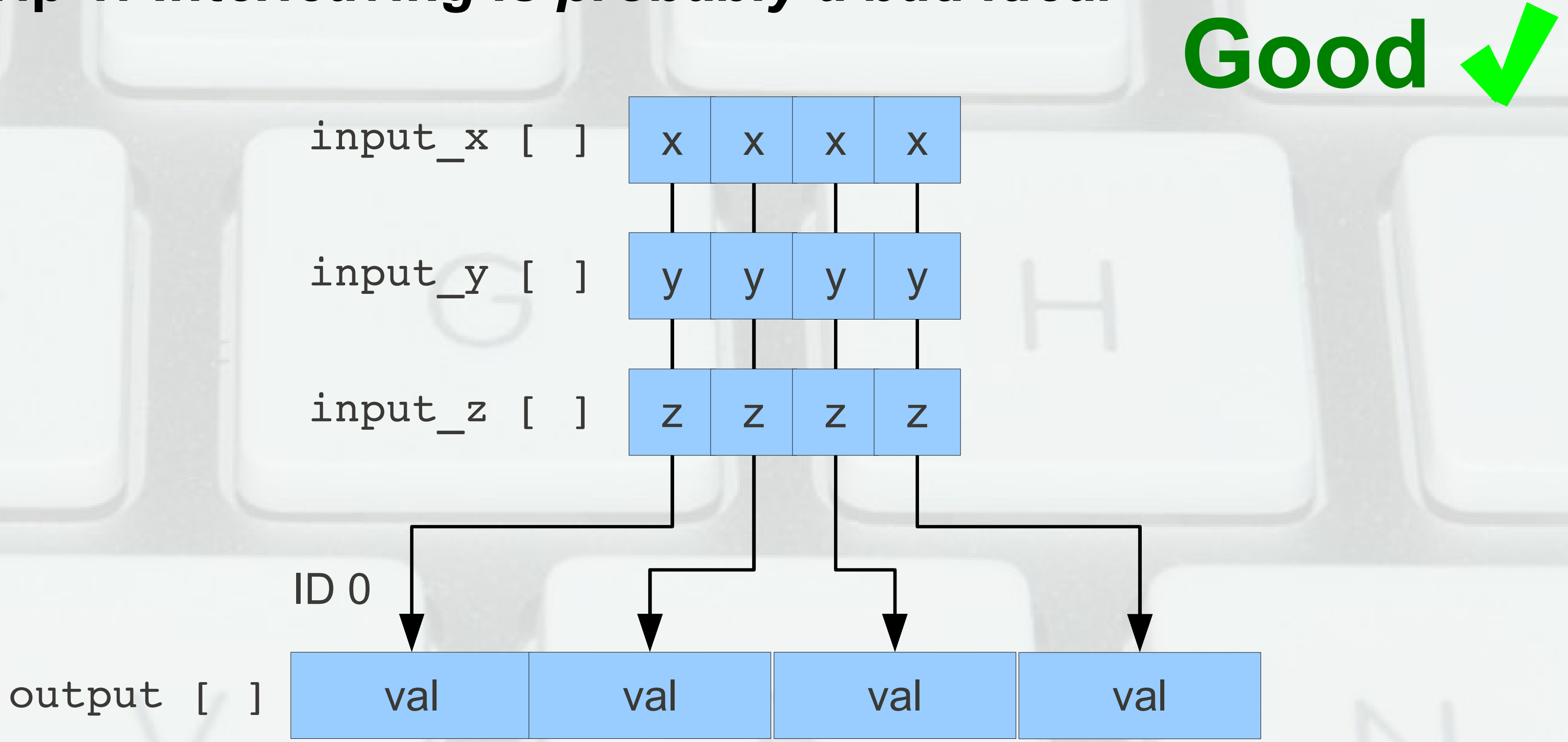
Essence of Structure

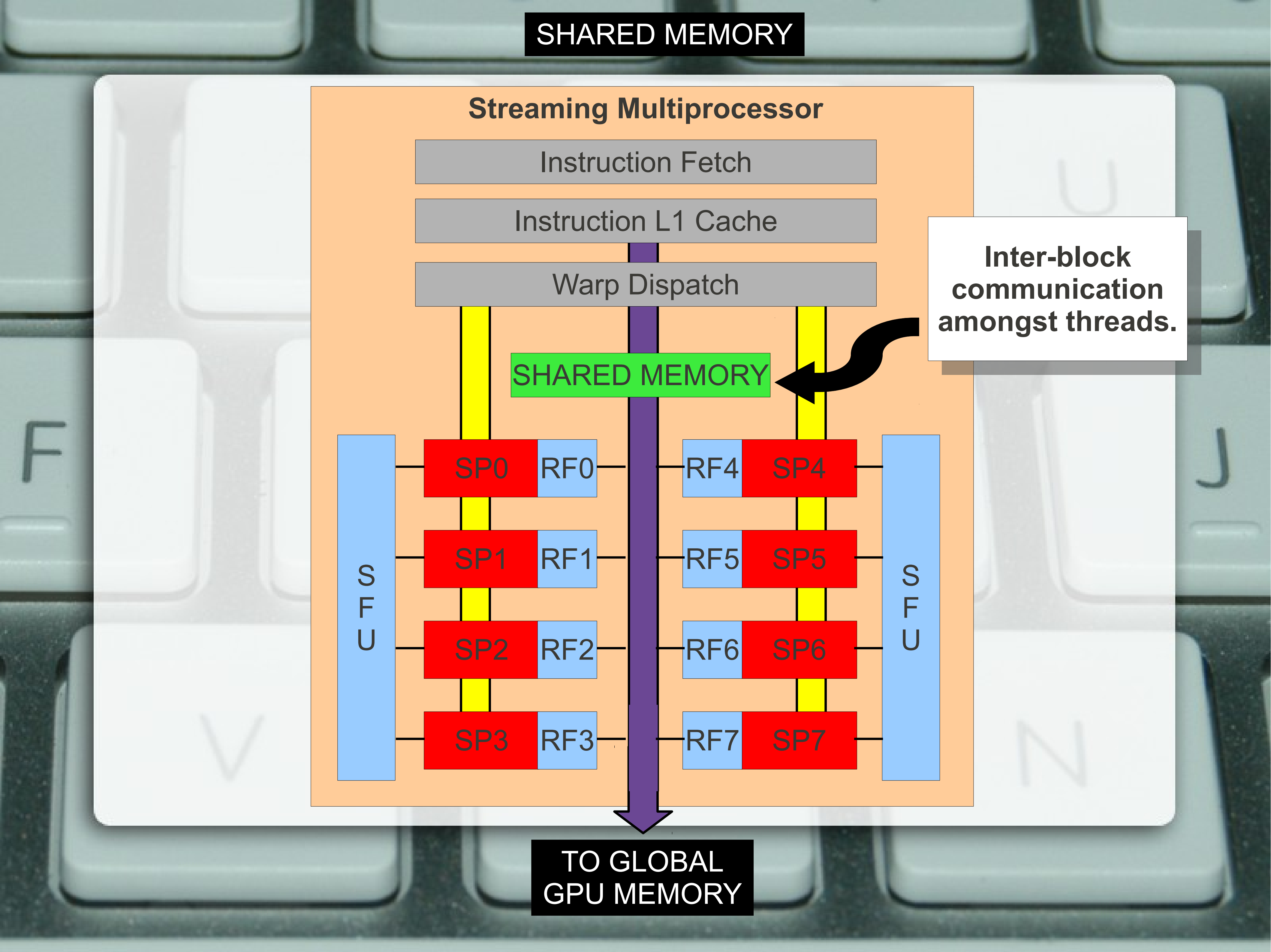
Tip 1: ***Interleaving is probably a bad idea.***



Essence of Structure

Tip 1: ***Interleaving is probably a bad idea.***




SHARED MEMORY
Streaming Multiprocessor
Instruction Fetch
Instruction L1 Cache
Warp Dispatch
SHARED MEMORY
Inter-block communication amongst threads.
SP0 RF0
SP1 RF1
SP2 RF2
SP3 RF3
RF4 SP4
RF5 SP5
RF6 SP6
RF7 SP7
SFU
SFU
TO GLOBAL GPU MEMORY

SHARED MEMORY

Each bank is 1KB
(16KB Shared Memory **TOTAL**)

Accessed by the **HALF WARP**
(16 threads at a time)

Each thread in a half warp should access a different bank!
(Avoid **BANK CONFLICTS**)

MUCH FASTER THAN GLOBAL MEMORY

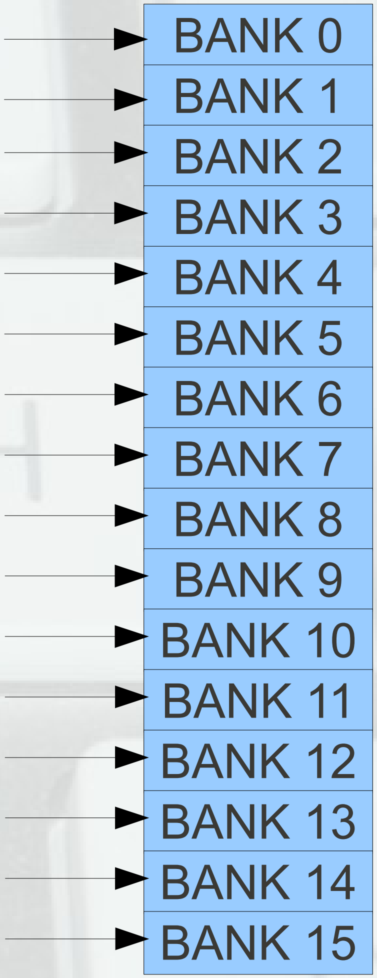
BANK 0
BANK 1
BANK 2
BANK 3
BANK 4
BANK 5
BANK 6
BANK 7
BANK 8
BANK 9
BANK 10
BANK 11
BANK 12
BANK 13
BANK 14
BANK 15

SHARED MEMORY

Can only be populated **DURING** kernel execution.

Has the <u>thread block</u> level **SCOPE**.

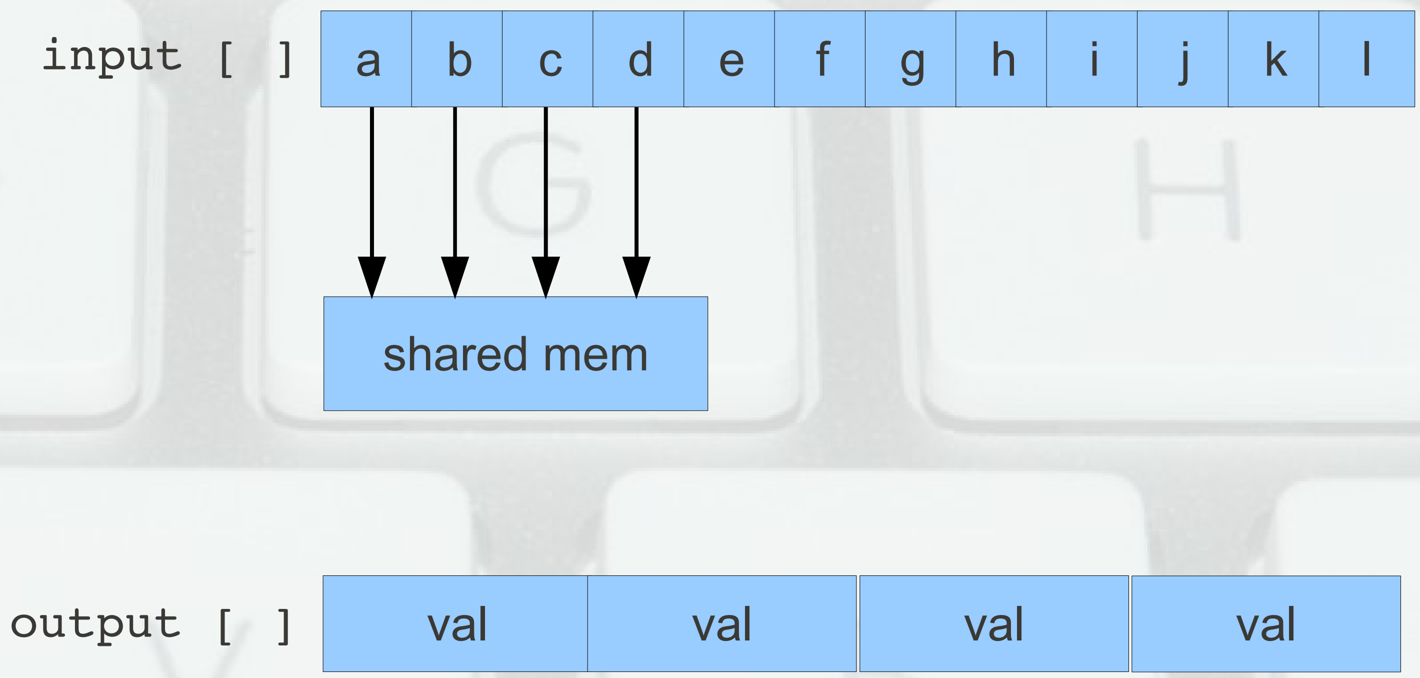
Amount of shared memory for each thread block is specified **AT LAUNCH**



Essence of Structure

Tip 2: *Many to fewer*: *assign thread blocks, not threads.*

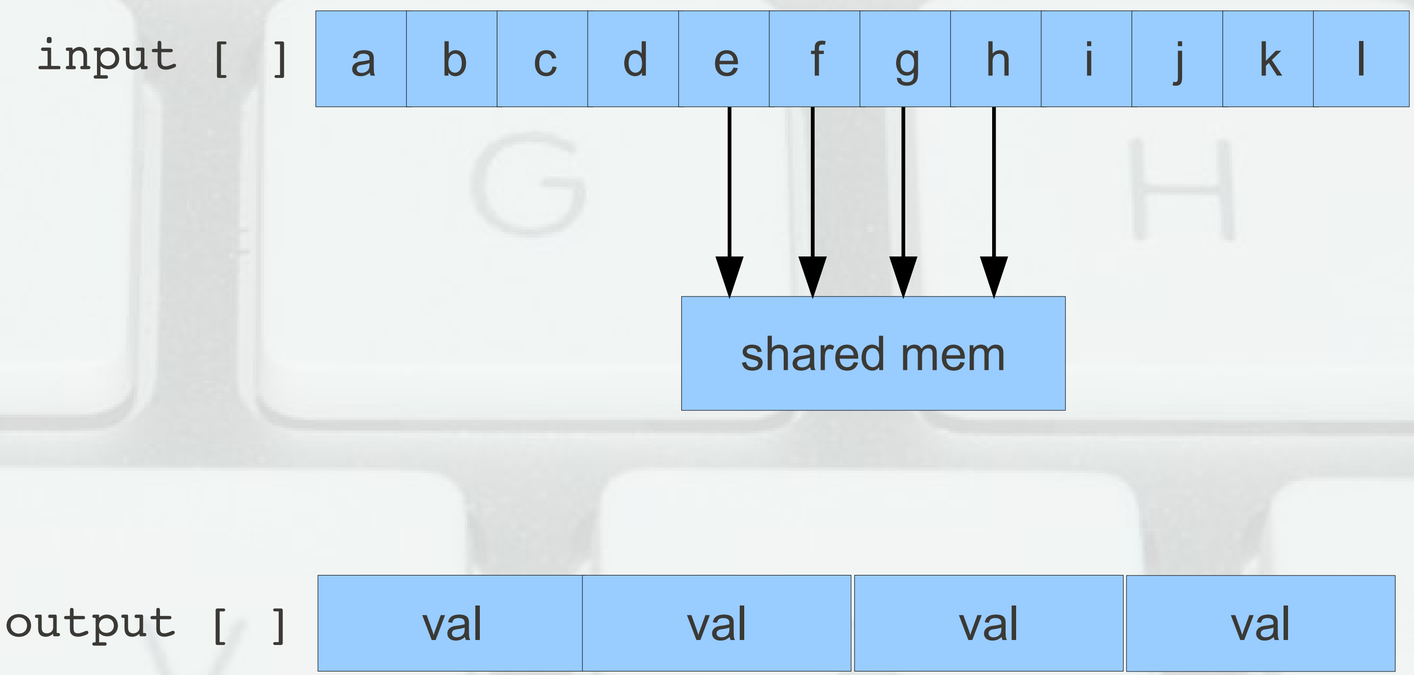
(example – 4 threads per block)



Essence of Structure

Tip 2: *Many to fewer*: *assign thread blocks, not threads.*

(example – 4 threads per block)



Essence of Structure

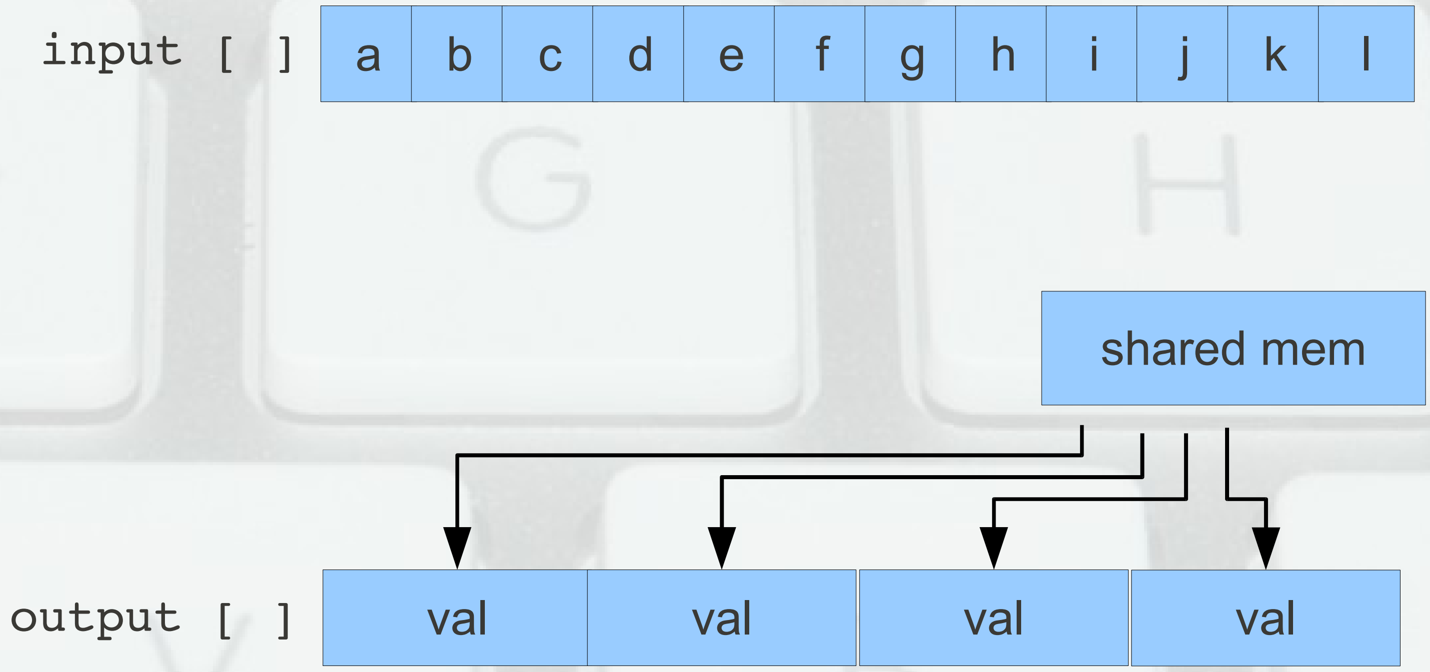
Tip 2: *<u>Many to fewer</u>: assign thread blocks, not threads.*

(example – 4 threads per block)

Essence of Structure

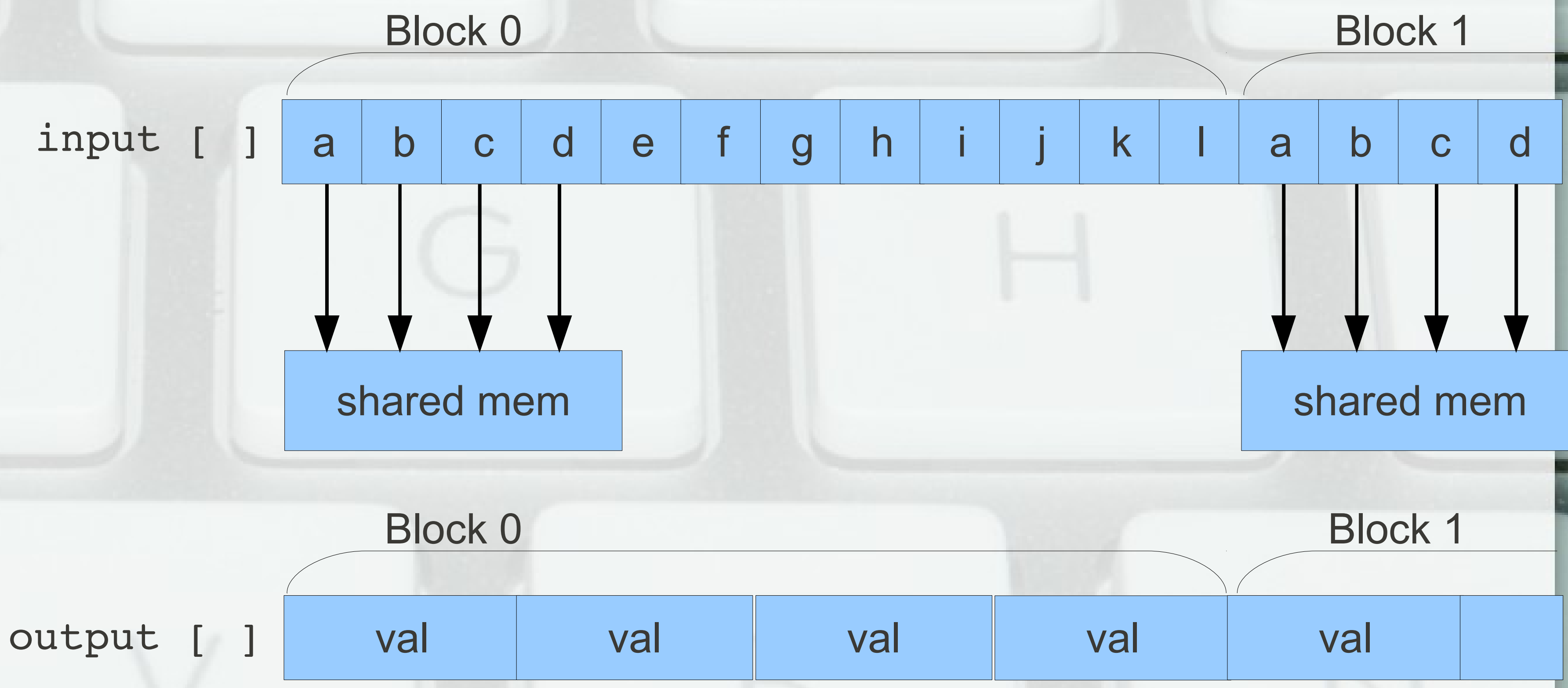
Tip 2: *<u>Many to fewer</u>: assign thread blocks, not threads.*

(example – 4 threads per block)



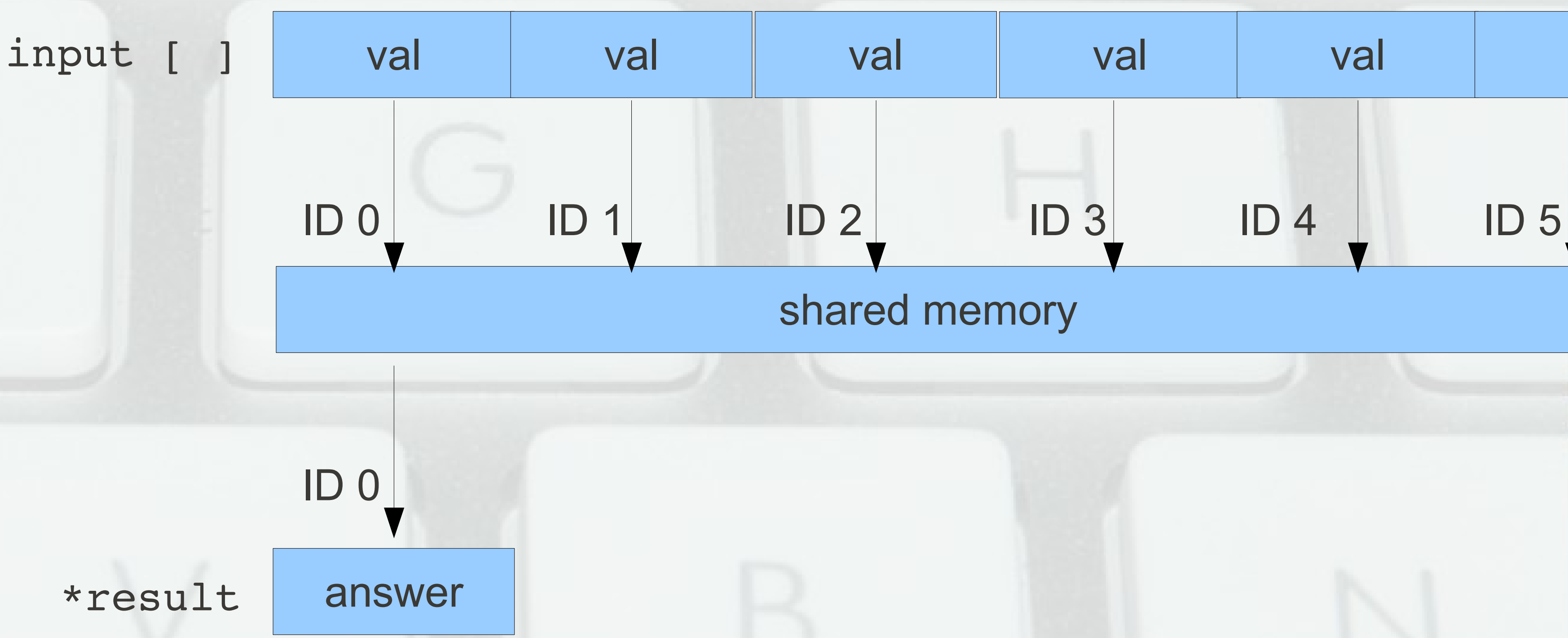
Essence of Structure

Tip 2: *<u>Many to fewer</u>*: *assign thread blocks, not threads.*



Essence of Structure

Tip 3: *<u>Many to one</u>: parallel sum reductions*



cuda-gdb command reference

`thread <<<(b.x, b.y), (t.x, t.y, t.z)>>>`

switch to specified CUDA thread

`info cuda threads`

reports which line # each active thread is currently on

`info cuda threads all`

reports line #s for all CUDA threads

`info cuda state`

reports memory allocation and symbols on GPU

`break` *`function`* `OR` *`file:line#`*

set breakpoint. pauses execution at specified point.

`info break`

list all breakpoints and show their numbers

`disable` *`breakpoint#`*

temporarily disable a breakpoint

`enable` *`breakpoint#`*

enable a previously disabled breakpoint

cuda-gdb command reference

`delete breakpoint#`

deletes specified breakpoint

`print variable`

displays the current contents of the specified variable

`print array[start]@end`

displays a slice of an array from element *start* thru element *end*

`watch variable`

set watchpoint. program will break when *variable* is modified

`run`

begin program execution

`step`

execute next line then break

`list`

show code listing around current breakpoint

`kill`

terminal program execution

`quit`

exit debugger