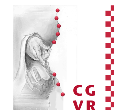
Remarks on Memory (Applies to GPUs and CPUs)



- In our dot product kernel, we could have done everything in global memory, but …
- Global memory bandwidth is sloooow:

Ideal

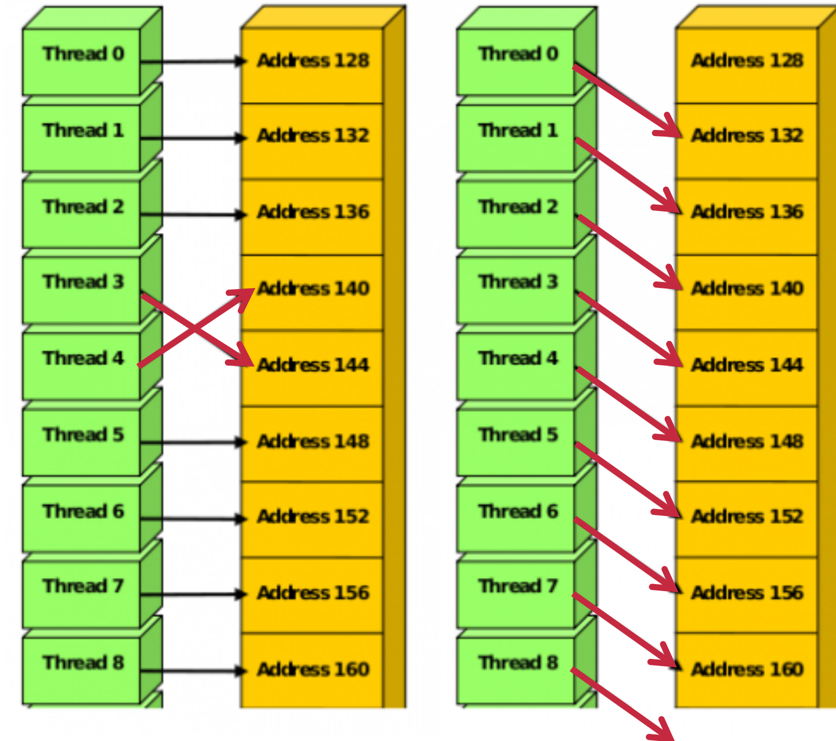
Reality

Coalesced Memory Access

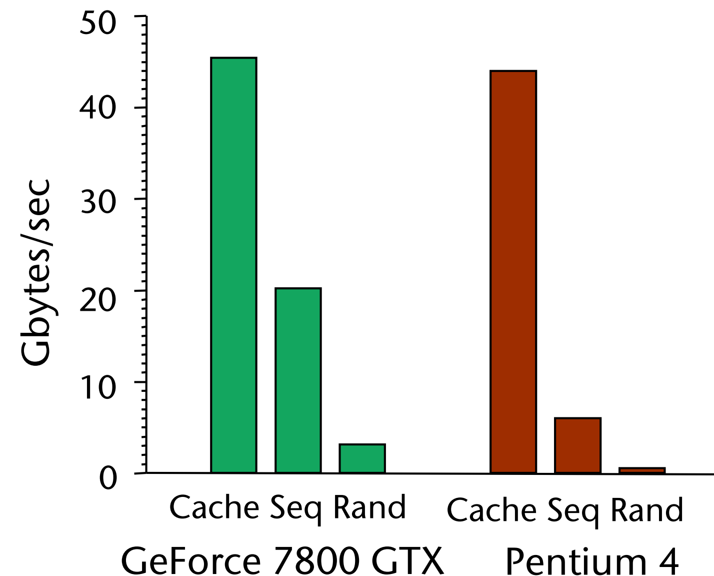
- One of the most important optimization techniques for massively parallel algorithm design (on GPUs and — to some degree — CPUs!)

Coalesced memory accesses

Uncoalesced memory accesses

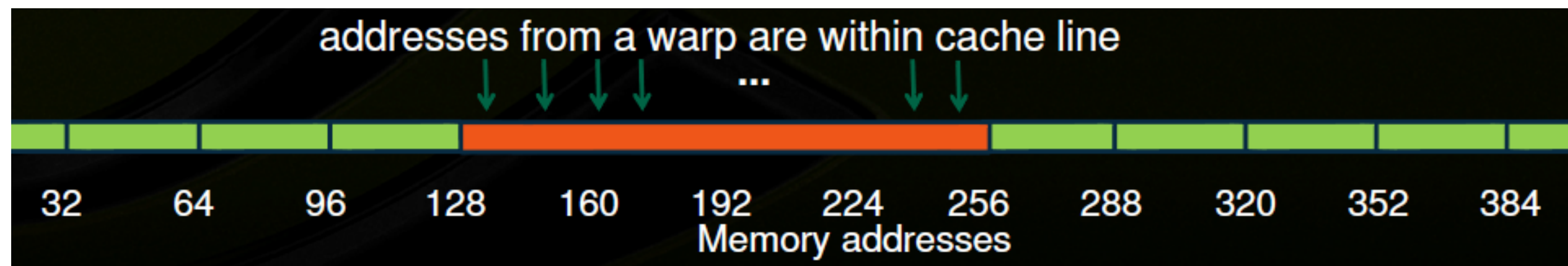


- When does the GPU win over the CPU?
- Arithmetic intensity of an algorithm := $\frac{\text{number of arithmetic operations}}{\text{amount of transferred bytes}}$
  - Sometimes also called computational intensity
- Unfortunately, many (most?) algorithms have a low arithmetic intensity → they are bandwidth limited
- GPU wins if memory access is "streamed" = coalesced
  - Hence, "stream programming architecture"



How to Achieve Coalesced Access

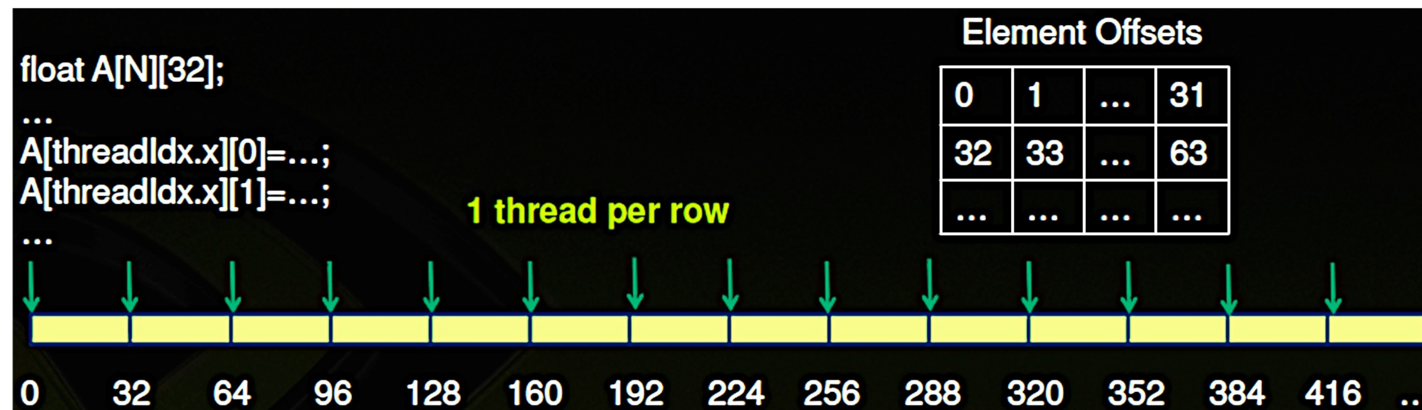
- Addresses from a warp ("thread-vector") are converted into memory line requests
  - Line sizes: 32B (= 32x **char**) and 128B (= 32x **float**)
  - Goal is to maximally utilize the bytes in these lines



2D Array Access Pattern (row major)

- Consider the following code piece in a kernel (e.g., matrix × vector):

```
for ( int j = 0; j < 32; j ++ ) {
   float x = A[treadIdx.x][j];
   ... do something with it ...
```



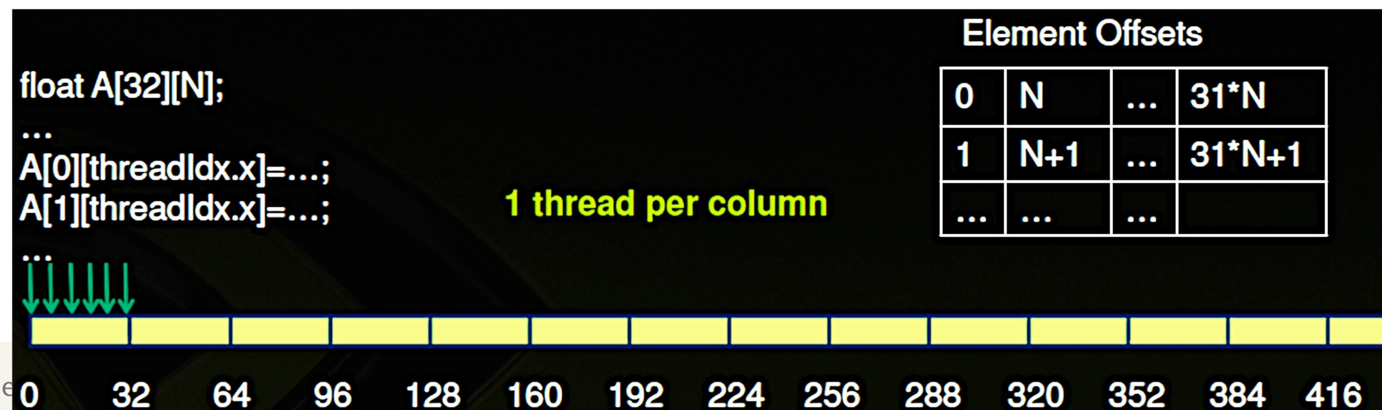
➤ Uncoalesced access pattern:
  - Elements read on 1st SIMT access: 0, 32, 64, ...
  - Elements read on 2nd SIMT access: 1, 33, 65, ...
  - Also, extra data will be transferred in order to fill the cache line size
- Generally, most natural access pattern for direct port of a C/C++ code!

Transposed 2D Array Access Pattern

- This "natural" way to store matrices is called row major order
- Column major := store a *logical* row in a *physical* column
  - I.e., $A_{00} \rightarrow$ A[0][0] , $A_{01} \rightarrow$ A[1][0] , $A_{02} \rightarrow$ A[2][0] , …
    $A_{10} \rightarrow$ A[0][1] , $A_{11} \rightarrow$ A[1][1] , $A_{12} \rightarrow$ A[2][1] , …
    $A_{20} \rightarrow$ A[0][2] , …
- *Transform* the code piece (e.g., row×column) to column major:

```
for ( int j = 0; j < 32; j ++ ){
   float x = A[j][treadIdx.x];
   ... do something with it ...
```

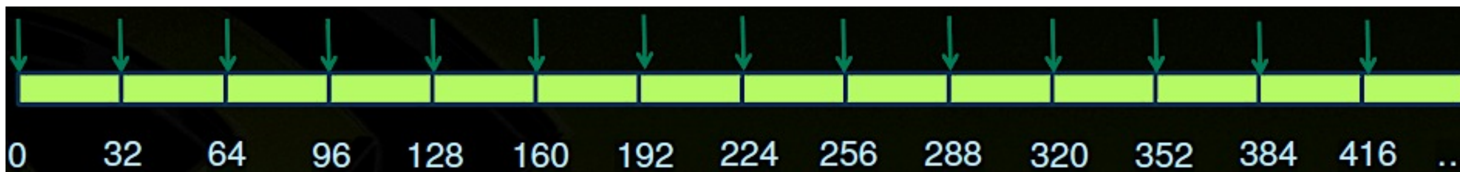
- Now, we have coalesced accesses:
  - Elements read on 1st SIMT access: 0, 1, 2, …, 31
  - Elements read on 2nd SIMT access: 32, 33, …, 63



Array of Structure or Structure of Array?

- An array of structures (AoS) behaves like *row major* accesses:

```
struct Point {
   float x; float y; float z;
};
Point PointList[N];
...
PointList[threadIdx.x].x = ...
```

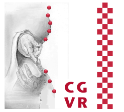


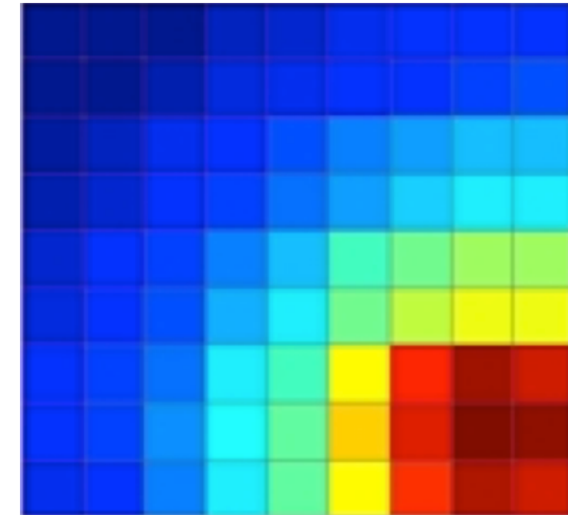
- A structure of arrays (SoA) behaves like *column major* access:

```
struct PointList {
   float x[N];
   float y[N];
   float z[N];
};
...
PointList.x[threadIdx.x] = ...
```

Bremen

Simulating Heat Transfer in Solid Bodies





- Assumptions:
  - For sake of illustration, our domain is 2D
  - Discretize domain → 2D grid (common approach in simulation)
  - A few designated cells are "heat sources" → cells with constant temperature
- Simulation model (simplistic):

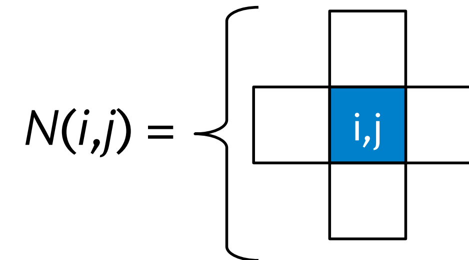
$$T_{i,j}^{n+1} = T_{i,j}^{n} + \sum_{(k,l) \in N(i,j)} \mu \left(T_{k,l}^{n} - T_{i,j}^{n} \right)$$

$$\Leftrightarrow \quad T_{i,j}^{n+1} = (1 - N\mu)\, T_{i,j}^{n} + \mu \sum_{(k,l) \in N(i,j)} T_{k,l}^{n} \qquad (1)$$

N = number of cells in the neighborhood

- Iterate this (e.g., until convergence to steady-state)

- Do we achieve energy conservation?
- For sake of simplicity, assume

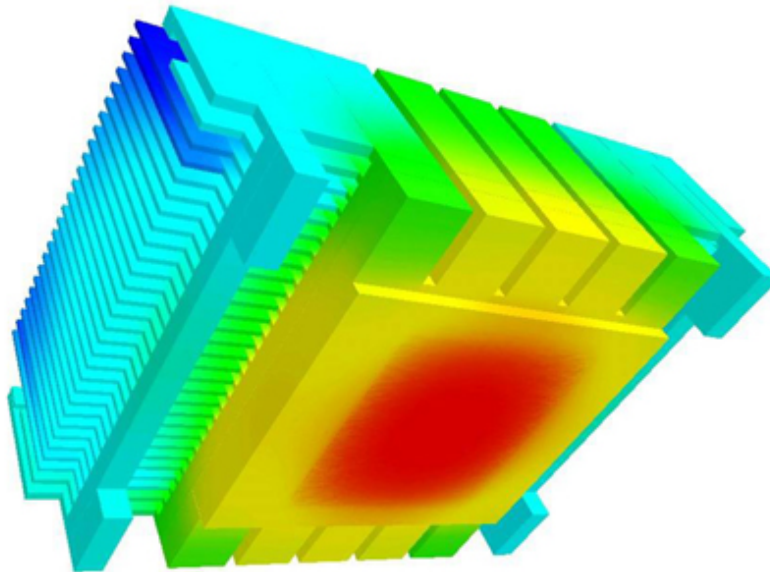
$N(i,j) =$ 

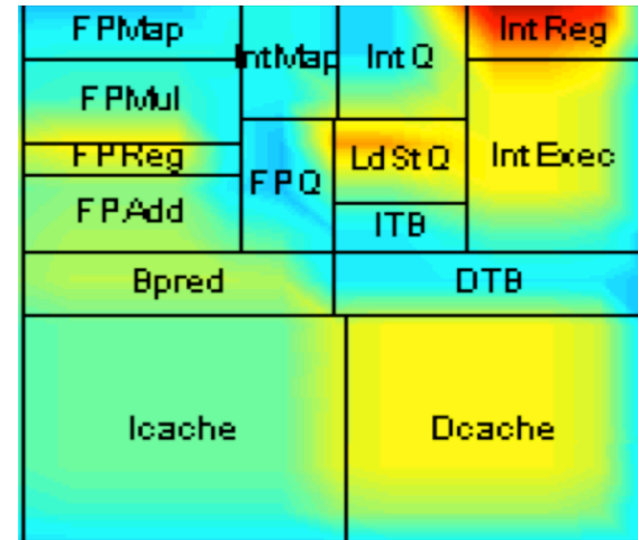
- Energy consumption iff $\sum_{i,j} T_{i,j}^{n+1} \stackrel{!}{=} \sum_{i,j} T_{i,j}^{n}$ (2)
- Plugging (1) into (2) yields

$$\underbrace{(1 - N\mu)\sum_{i,j} T_{i,j}^{n} + \mu \sum_{i,j} \sum_{(k,l)\in N(i,j)} T_{k,l}^{n}}_{=0} \stackrel{!}{=} \sum_{i,j} T_{i,j}^{n}$$

- Therefore, μ is indeed a free material parameter (= "heat flow speed")

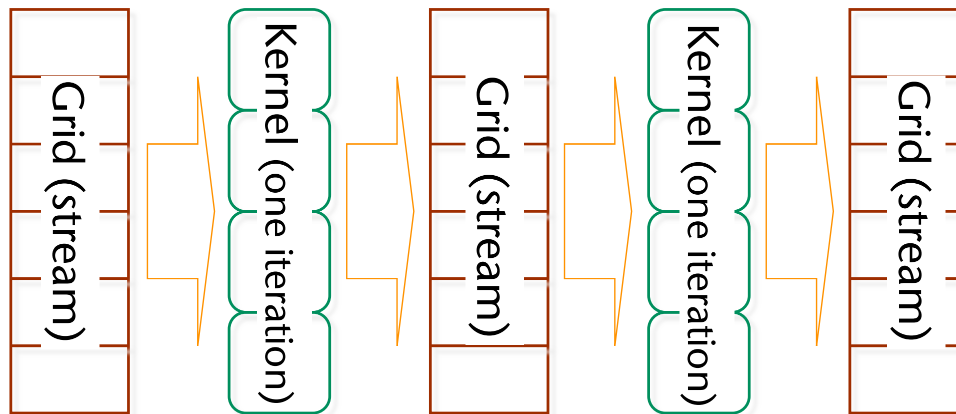
- Example: heat simulation of ICs and cooling elements



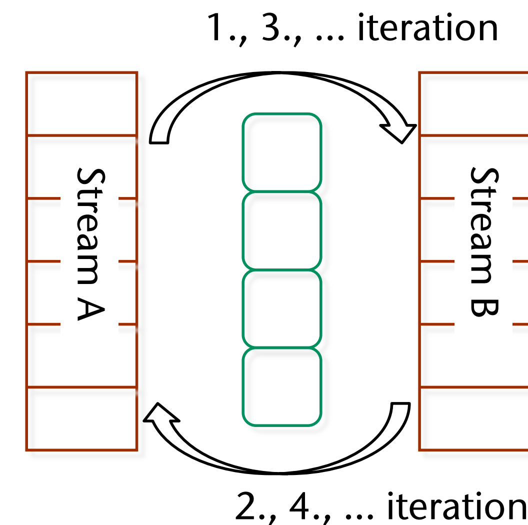


MassPar Algorithm Design Pattern: Double Buffering

- Observations:
  - Each cell's next state can be computed completely independently
- ➢ We can arrange our computations like this:



- General parallel programming pattern: double buffering ("ping pong")



Algorithm

- One thread per cell

1. Kernel for resetting heat sources:

```
if ( cell is heat cell ):
   read temperature from constant "heating stencil"
```

2. Kernel for one transfer step:

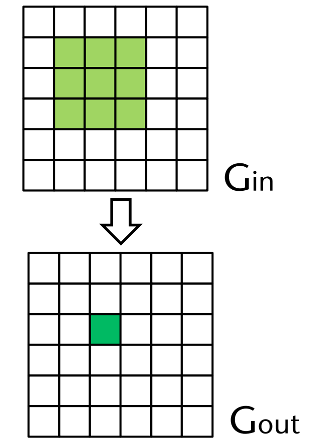
```
Read all neighbor cells:  input_grid[tid.x+-1][tid.y+-1]
Accumulate them
Write new temperature in output_grid[tid.x][tid.y]
```

3. Swap pointers to input & output grid (done on host)

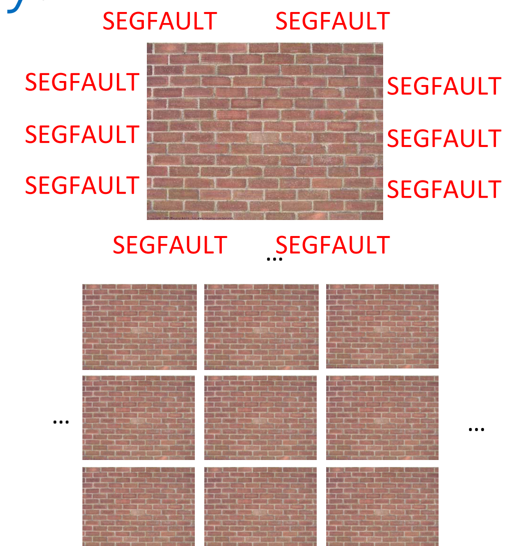
- Challenge: *border cells*! (very frequent problem in sim. codes)
  - Use if-then-else in above kernel?
  - Use extra kernel that is run only for border cells?
  - Introduce padding around domain? Arrange domain as torus?

Texture Memory Optional

- Many computations have the following characteristics:
  - They *iterate* a simple function many times
  - They work on a 2D/3D *grid*
  - We can run *one thread per grid cell*
  - Each thread only needs to look at *neighbor cells*
  - Each iteration transforms an input grid into an output grid

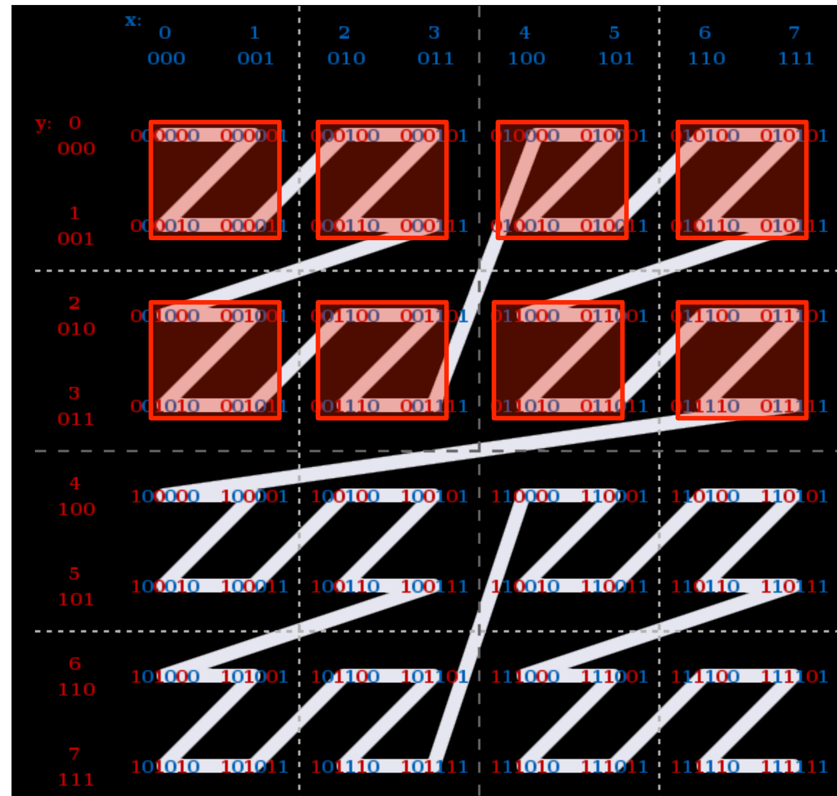


- For this kind of algorithms, there is texture memory:
  - Special cache with optimization for spatial locality
  - Access to neighbor cells is very fast
  - Important: can handle out-of-border accesses automatically by clamping or wrap-around!
- For the technical details: see "Cuda by Example", Nvidia's "CUDA C Programming Guide",



Optional

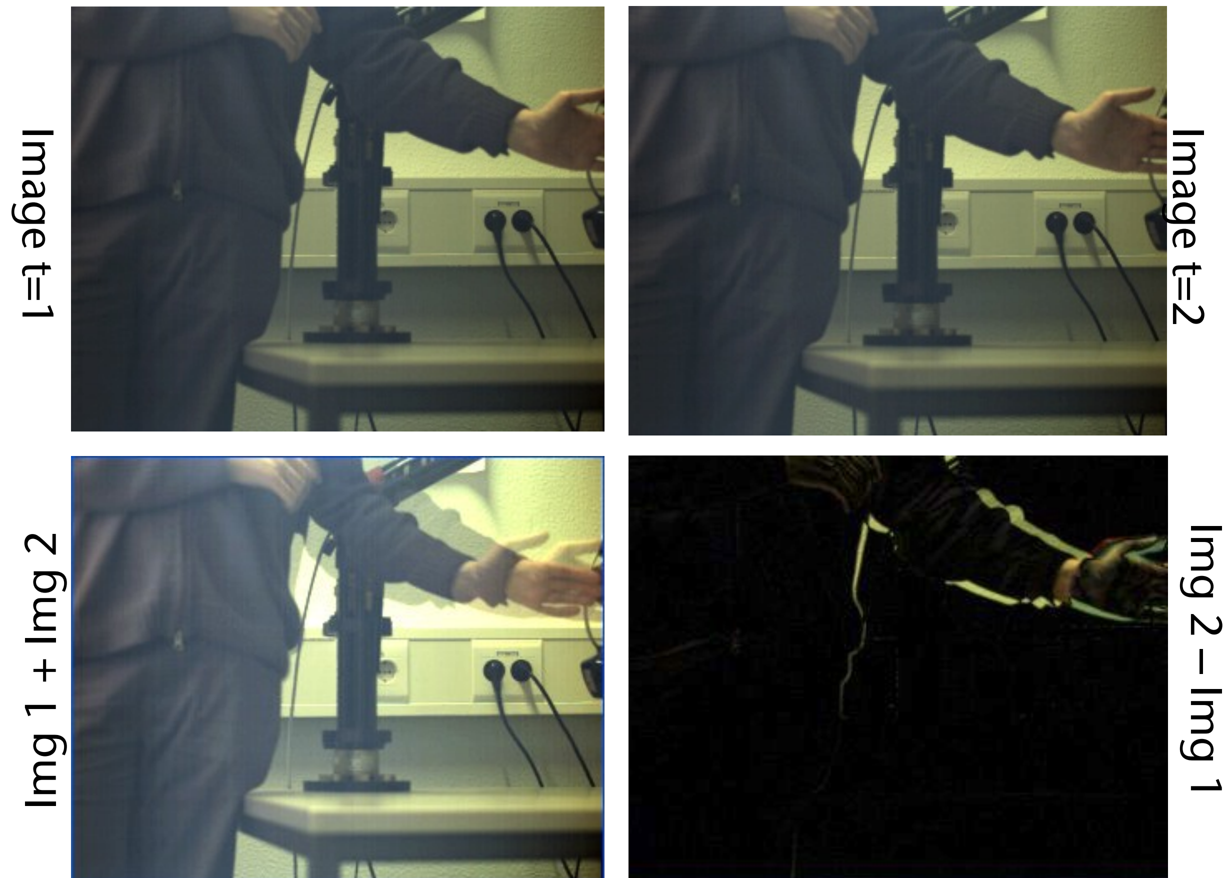
- The locality-preserving cache is probably achieved by arranging data via a space-filling curve:



Other Applications of Texture Memory

Optional

- Most image processing algorithms exhibit this kind of locality
- Trivial example: image addition / subtraction ⟶ neighboring threads access neighboring pixels

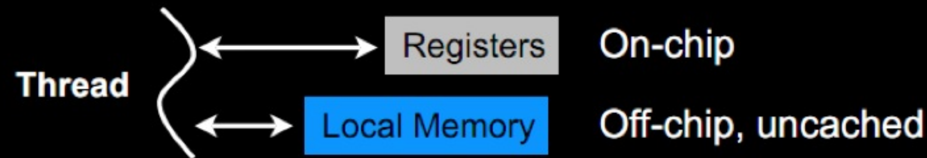


CUDA's Memory Hierarchy

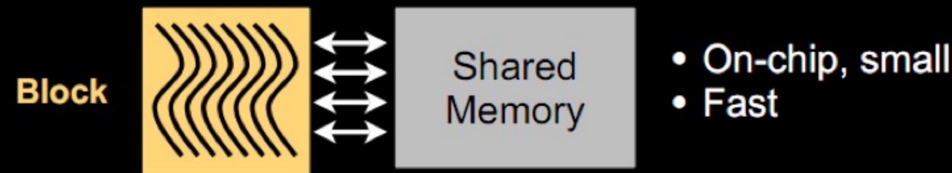
Kernel Memory A

Review

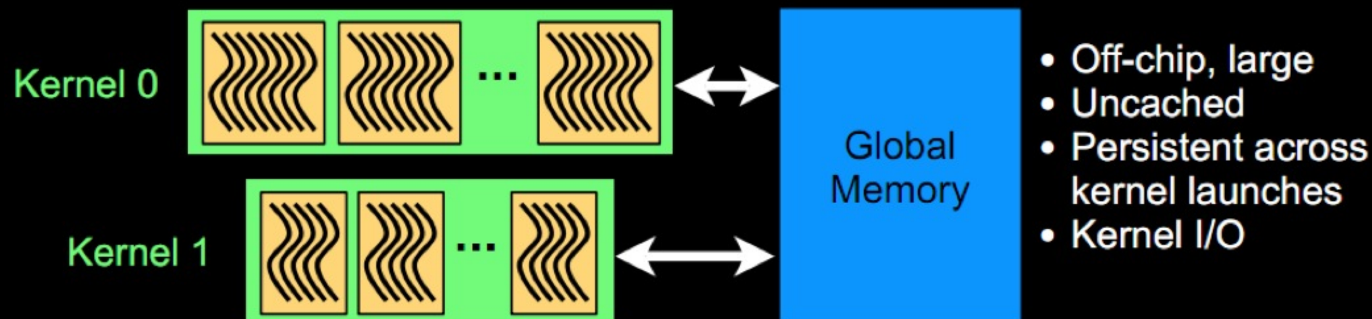
- Per-thread

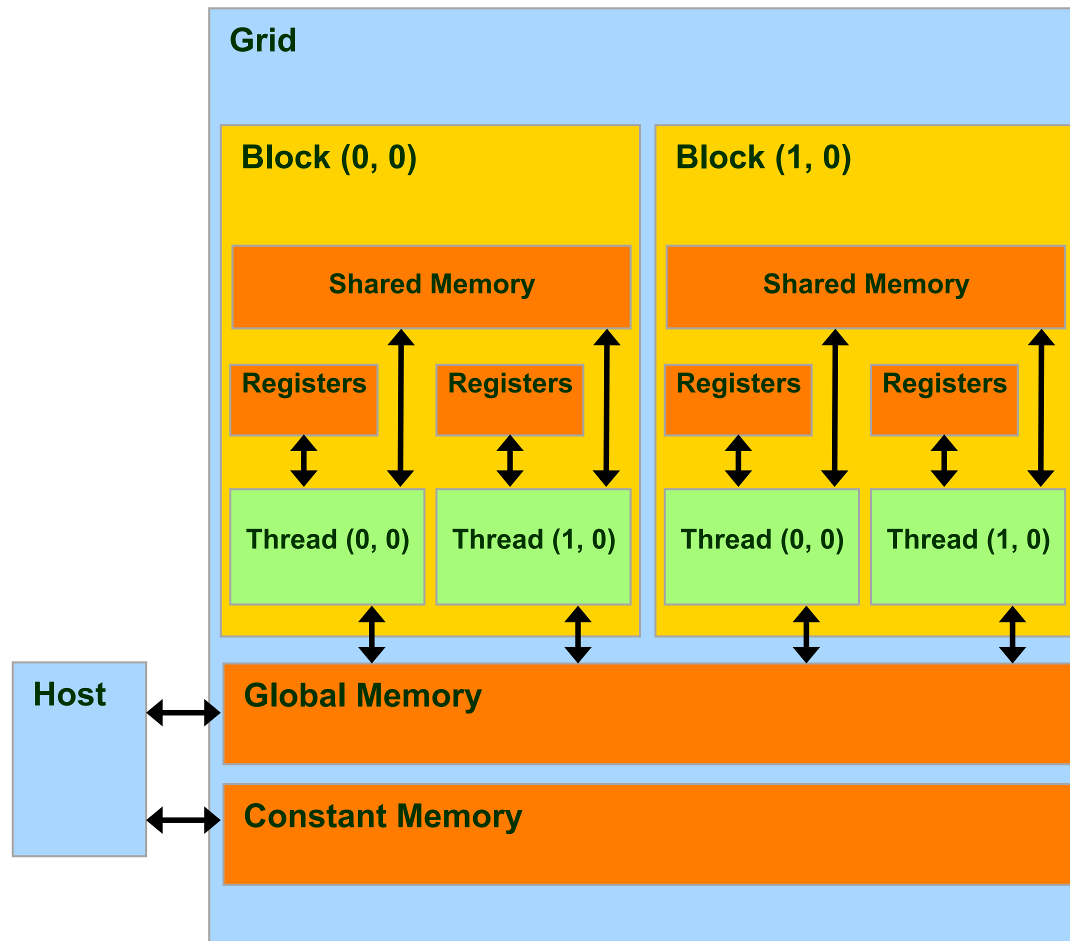


- Per-block



- Per-device




Grid
Block (0, 0)
Block (1, 0)
Shared Memory
Shared Memory
Registers
Registers
Registers
Registers
Thread (0, 0)
Thread (1, 0)
Thread (0, 0)
Thread (1, 0)
Host
Global Memory
Constant Memory

CUDA Variable Type Qualifiers

Variable declaration			Memory	Access	Lifetime
`__device__`	`__local__`	`int LocalVar;`	local	thread	thread
`__device__`	`__shared__`	`int SharedVar;`	shared	block	block
`__device__`		`int GlobalVar;`	global	grid	application
`__device__`	`__constant__`	`int ConstantVar;`	constant	grid	application

- Remarks:
  - `__device__` is optional when used with `__local__`, `__shared__`, or `__constant__`
  - Automatic variables without any qualifier reside in a register
    - Except arrays, which reside in local memory (slow)

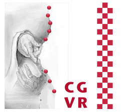
Bremen

CUDA Variable Type Performance

Variable declaration		Memory	Penalty
	`int var;`	register	1x
	`int array_var[10];`	local	100x
`__shared__`	`int shared_var;`	shared	1x
`__device__`	`int global_var;`	global	100x
`__constant__`	`int constant_var;`	constant	1x

- Scalar variables reside in fast, on-chip registers
- Shared variables reside in fast, on-chip memories
- Thread-local arrays & global variables reside in uncached off-chip memory
- Constant variables reside in cached off-chip memory

Where to Declare Variables?



Can host access it?

yes → global or constant → Outside of any function

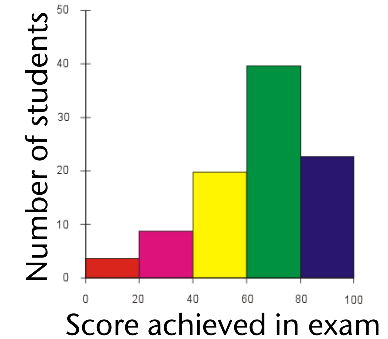
no → register (auto), or shared, or local → In the kernel

Massively Parallel Histogramm Computation

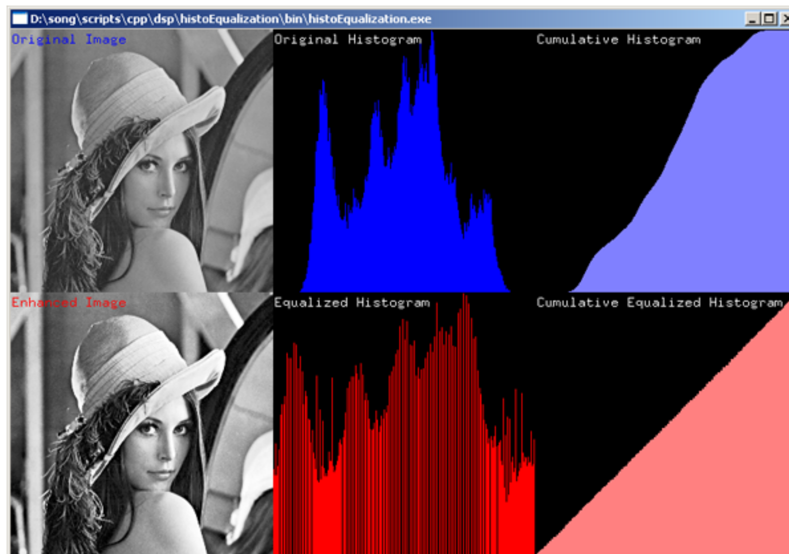
- Definition (for images):

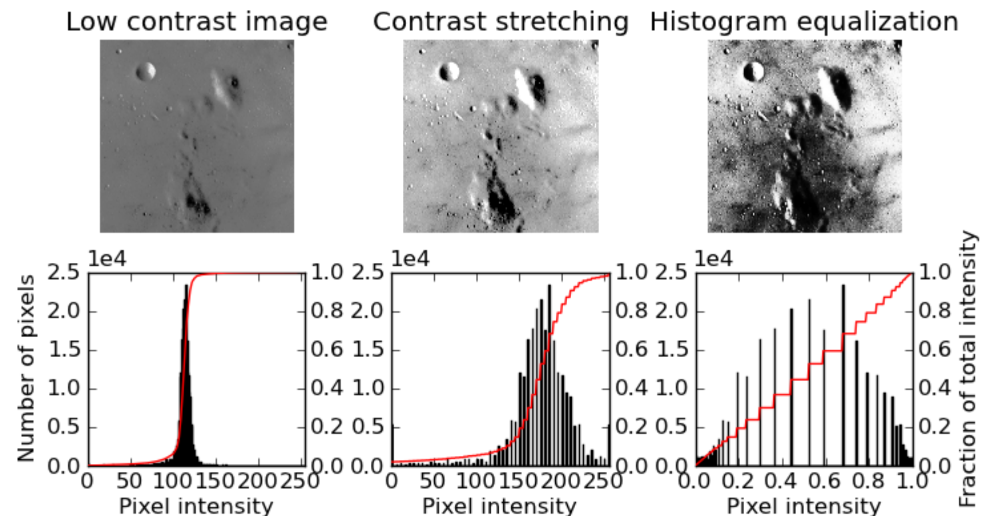
$$h(x) = \#\text{ pixels with level } x$$

$$x \in 0, \ldots, L-1 \qquad L = \#\text{ levels}$$



- Applications: many!
  - Huffman compression (see computer science 2nd semester)
  - Histogram equalization (see Advanced Computer Graphics)





- The sequential algorithm:

```
unsigned char input[MAX_INP_SIZE];// e.g. image
int input_size;                   // # valid chars in input
unsigned int histogram[256];      // here, 256 levels

// clear histogram
for (int i = 0; i < 256; i ++ )
   histogram[i] = 0;
for (int i = 0; i < input_size; i ++ )
   histogram[ input[i] ] ++ ;     // real histogram comput.

// verify histogram
long int total_count = 0;
for (int i = 0; i < 256; i ++ )
   total_count += histogram[i];
if ( total_count != input_size )
   fprintf(stderr, "Error! ..." );
```

- Naïve "massively parallel" algorithm:
  - One thread per bin (e.g., 256)
  - Each thread scans the complete input and counts the number of occurrences of its "own" intensity level in the image
  - At the end, each thread stores its level count in its histogram slot

- Disadvantage: not so massively parallel …

- New approach: "one thread per pixel"
- The setup on the host side:

```
set up device arrays d_input, d_histogram
cudaMemset( d_histogram, 0, 256 * sizeof(int) );
int threadsPerBlock = 256;
int nBlocks = #(multiprocessors on device) * 2;
computeHistogram <<< nBlocks, threadsPerBlock  >>>
                  ( d_input, input_size, d_histogram );
```

- Notes:
  - Letting **`threadsPerBlock`** = 256 makes things much easier in our case
  - Letting **`nBlocks`** = (number of multiprocessors [SMs] on the device) * 2 is a good rule of thumb, YMMV
  - On current hardware (Kepler) → ~ 16384 threads

- The kernel on the device side:

```
__global__ void
computeHistogram( unsigned char * input,
                  long int input_size,
                  unsigned int histogram[256] )
{
   int i = threadIdx.x + blockIdx.x * blockDim.x;
   int stride = blockDim.x * gridDim.x;
   while ( i < input_size )
   {
      histogram[ input[i] ] += 1;
      i += stride;
   }
}
```

- Problem: race condition!!

Solution: Atomic Operations

- The kernel with atomic add:

```
__global__ void
computeHistogram( unsigned char * input,
                  long int input_size,
                  unsigned int histogram[256] )
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;
    while ( i < input_size )
    {
        atomicAdd( & histogram[input[i]], 1 );
        i += stride;
    }
}
```

- Prototype of atomicAdd():

```
T atomicAdd( T * address, T val )
```

where `T` can be **`int`**, **`float`** (and a few other types)

- Semantics: *while atomicAdd performs its operation on* address, *no other thread can access this memory location*! *(neither read, nor write)*
- Problem: this algorithm is much slower than the sequential one!
  - Lesson: always measure performance against CPU!

- Cause: congestion
  - *Lots* of threads waiting for a *few* memory locations to become available

- Remedy: partial histograms in shared memory

```
computeHistogram( unsigned char * input,
                  long int input_size,
                  unsigned int histogram[256] )
{
    __shared__ unsigned int partial_histo[256];

    partial_histo[ threadIdx.x ] = 0;
    __syncthreads();

    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;
    while ( i < input_size ) {
        atomicAdd( & partial_histo[input[i]], 1 );
        i += stride;
    }
    __syncthreads();
    atomicAdd( & histogram[threadIdx.x],
               partial_histo[threadIdx.x] );
}
```

- Note: now it's obvious why we chose 256 threads/block

More Atomic Operations

- All programming languages / libraries / environments providing for some kind of parallelism/concurrency have one or more of the following atomic operations:
  - **`int atomicExch( int* address, int val )`**:
    Read old value at address, store **val** in address, return old value
  - Atomic AND: performs the following in one atomic operation

```
int atomicAnd( int* address, int val )
{
    int old = *address;
    *address = old & val;
    return old;
}
```

  - Atomic Minimum operation (just analogous to AND)
  - Atomic compare-and-swap (CAS), and several more ...

- The fundamental atomic operation Compare-And-Swap:
  - In CUDA: **`int atomicCAS( int* address, int compare, int val )`**
  - Performs this little algorithm atomically:

```
atomic_compare_and_swap( address, compare, new_val ):
    old ← value in memory location address
    if compare == old:
        store new_val → memory location address
    return old
```

- Theorem (w/o proof):
  All other atomic operations can be implemented using atomic compare-and-swap.

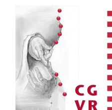
Optional

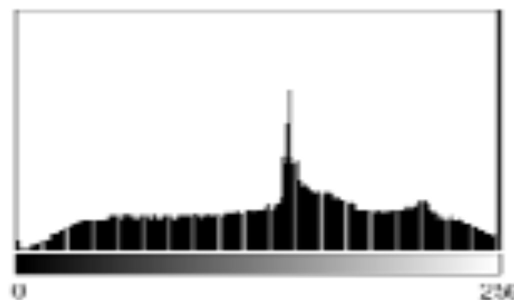
- Example:

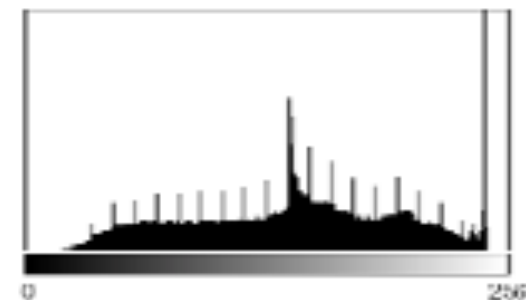
```
atomic_add( address, incr ):
   current_val := value in memory location address
   repeat
      new_val     := current_val + incr
      assumed_val := current_val
      current_val := compare_and_swap( address,
                                       assumed_val,
                                       new_val )
   until assumed_val == current_val
```

Bremen

Image Restoration Using Histograms

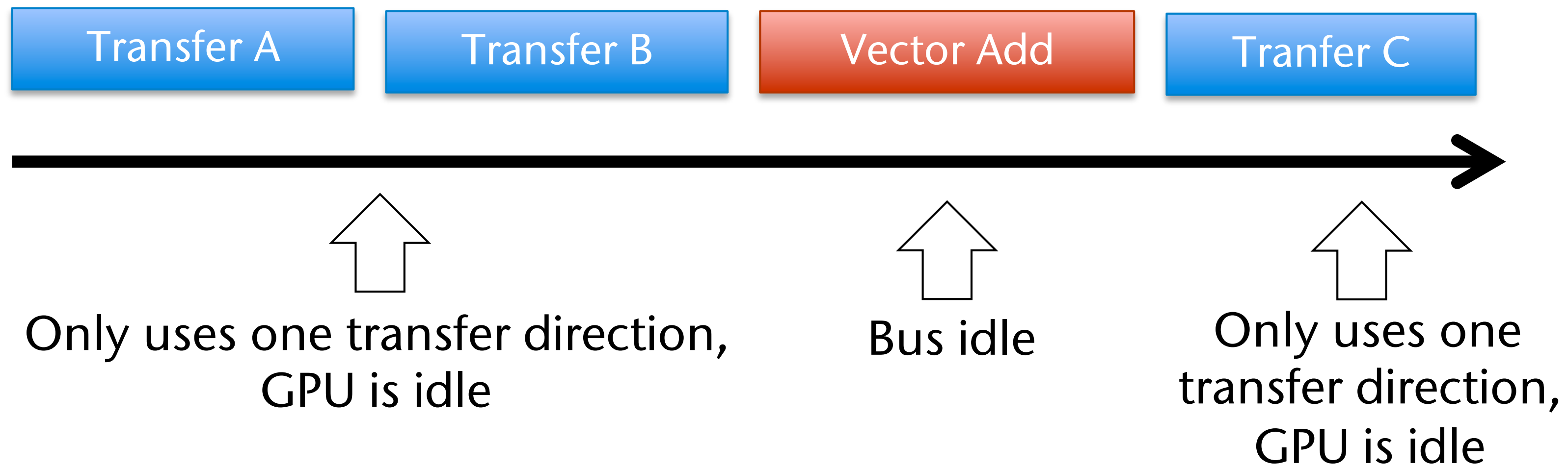




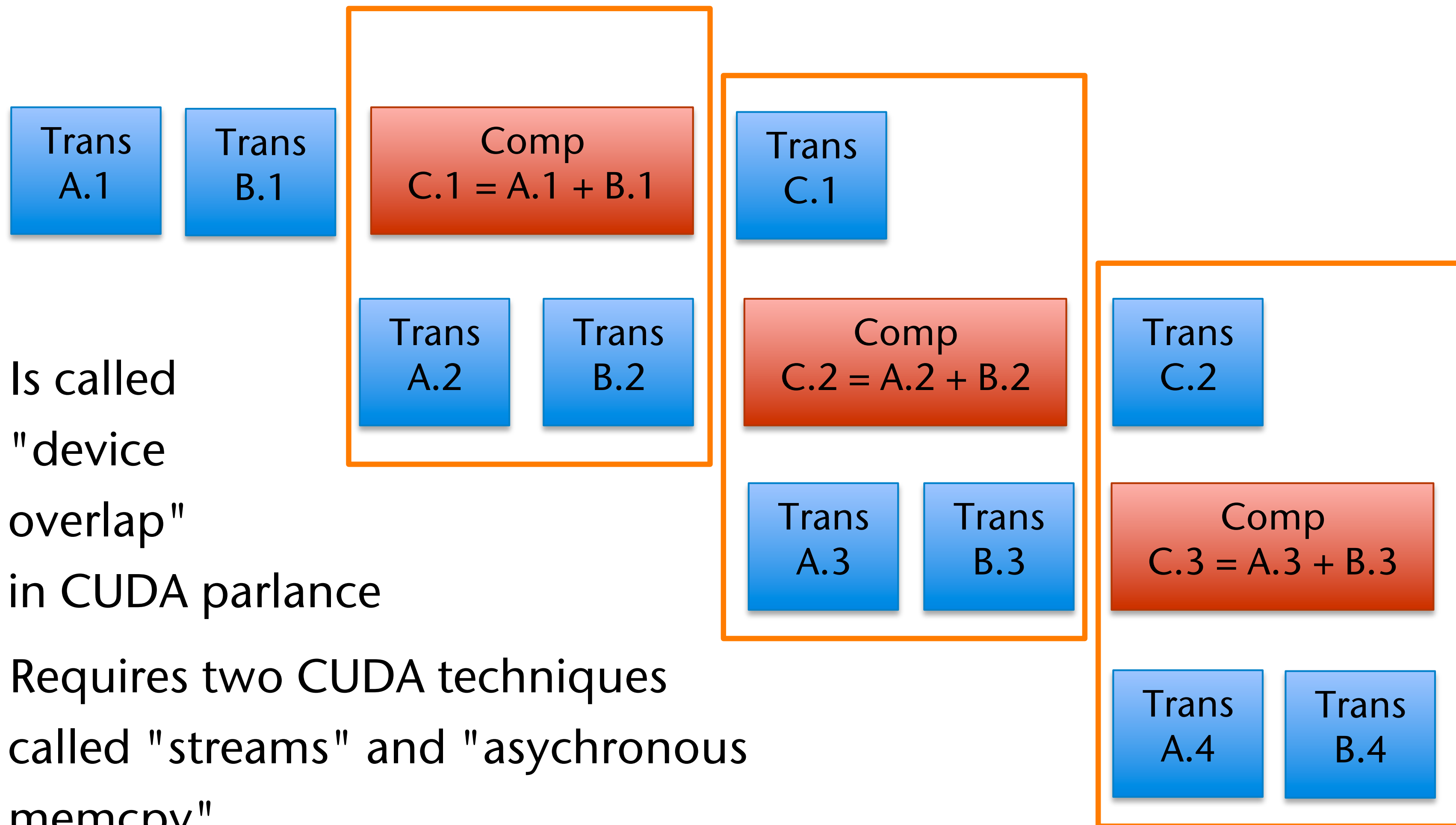


Advanced GPU & Bus Utilization

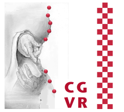
- Problem with performance, if lots of transfer between GPU↔CPU:



- Solution: pipelining (the "other" *parallelism paradigm*)



- Is called "device overlap" in CUDA parlance
- Requires two CUDA techniques called "streams" and "asychronous memcpy"

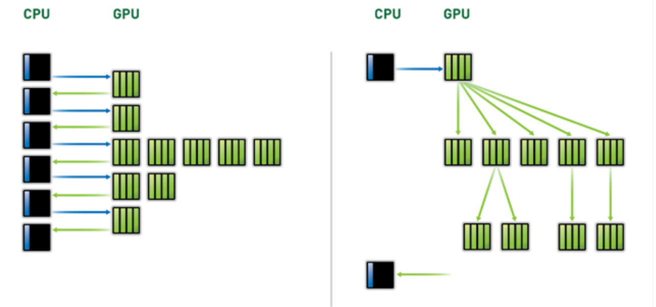


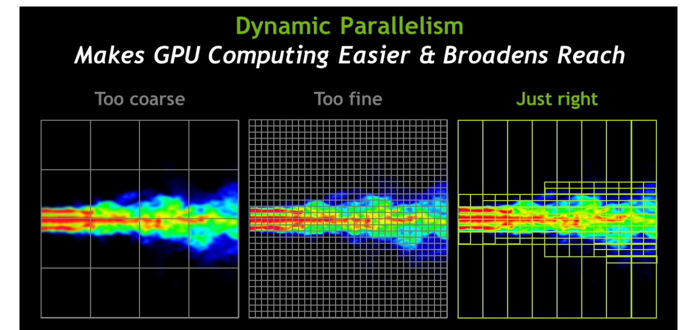
For More Information on CUDA …

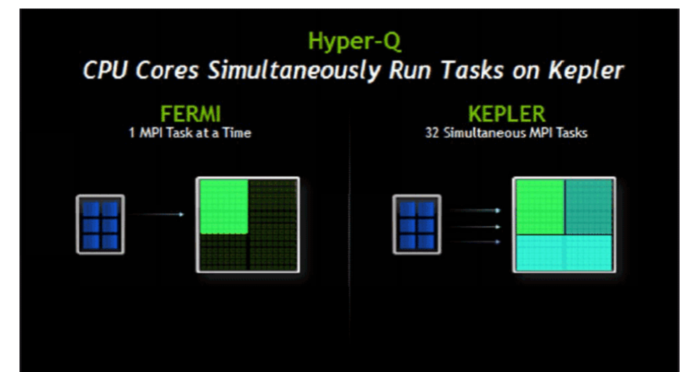
- *CUDA C Programming Guide* (zur Programmiersprache)
- *CUDA C Best Practices Guide* (zur Performance-Steigerung)
- */Developer/NVIDIA/CUDA-5.0/doc/html/index.html*
  (zum Runtime API)

Concepts we Have Not Covered Here

- Dynamic parallelism (threads can launch new threads)
  - Good for irregular data parallelism (e.g., tree traversal, multi-grids)
- Running several tasks at the same time on a GPU (via MPI; they call it "Hyper-Q")







- See:
  - "Introduction to CUDA 5.0" on the course web page
  - "CUDA C Programming Guide" at docs.nvidia.com/cuda/index.html

- Graphics Interoperability:
  - Transfer images directly from CUDA memory to OpenGL's framebuffer
- Dynamic shared memory
- Asynchronous memory copies between host ↔ device
- Dynamic memory allocation in the kernel
  - Can have serious performance issues
- Pinned CPU memory (
- CUDA Streams
- Multi-GPU programming, GPU-to-GPU memory transfer
- Zero-copy data transfer
- Libraries: CUBLAS, Thrust, …
- Voting functions (__all(), __any())

- With Graphics Interoperability, you can render results from CUDA directly in a 3D scene, e.g. by using them as textures