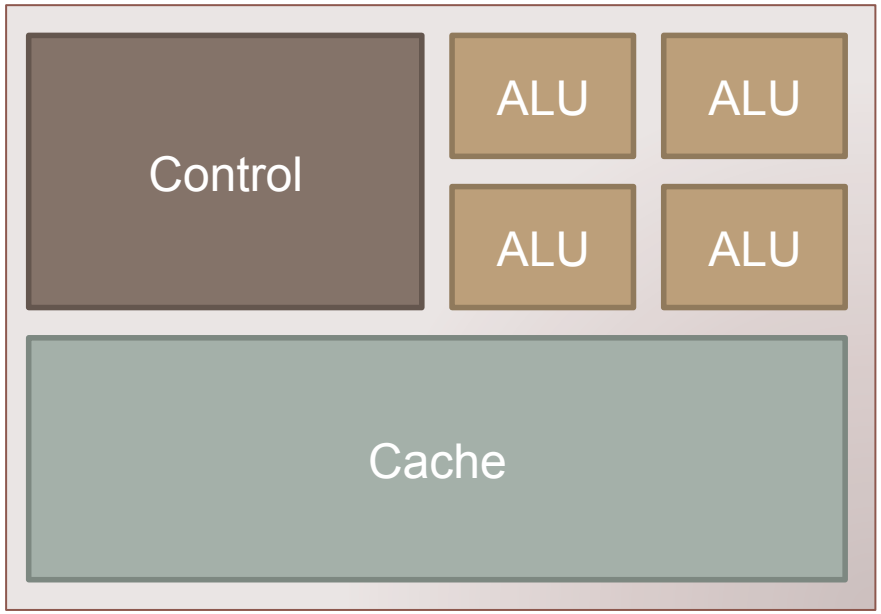
MORE ADVANCED GPU PROGRAMMING

CPU vs. GPU

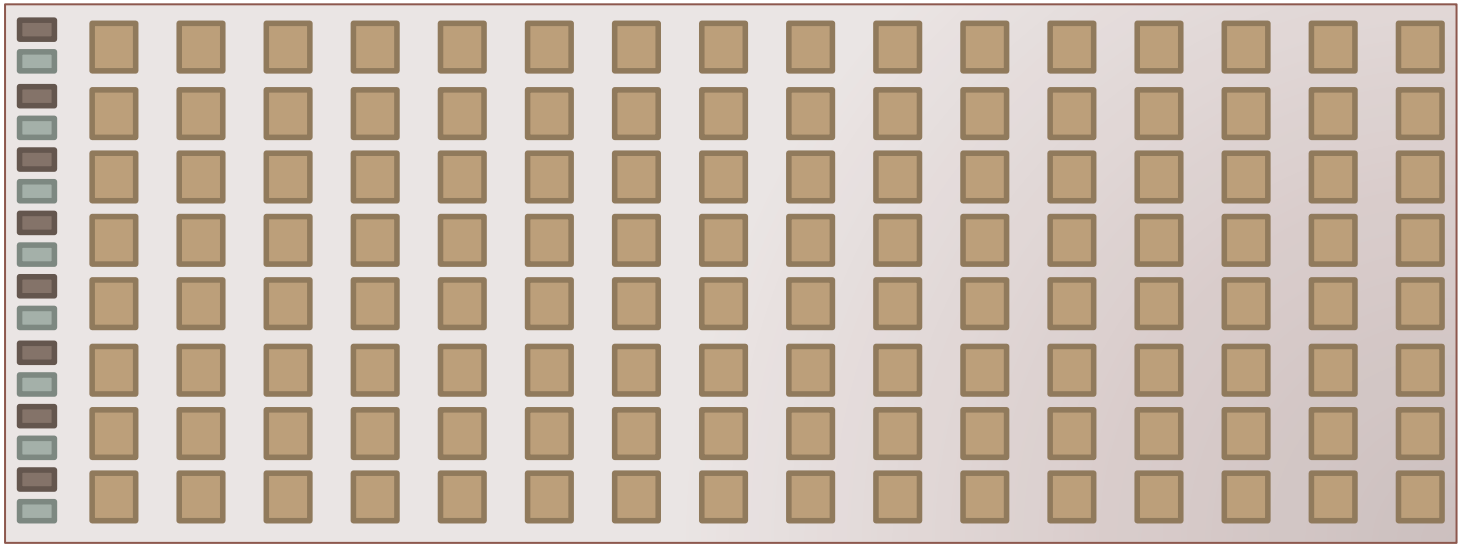


CPU

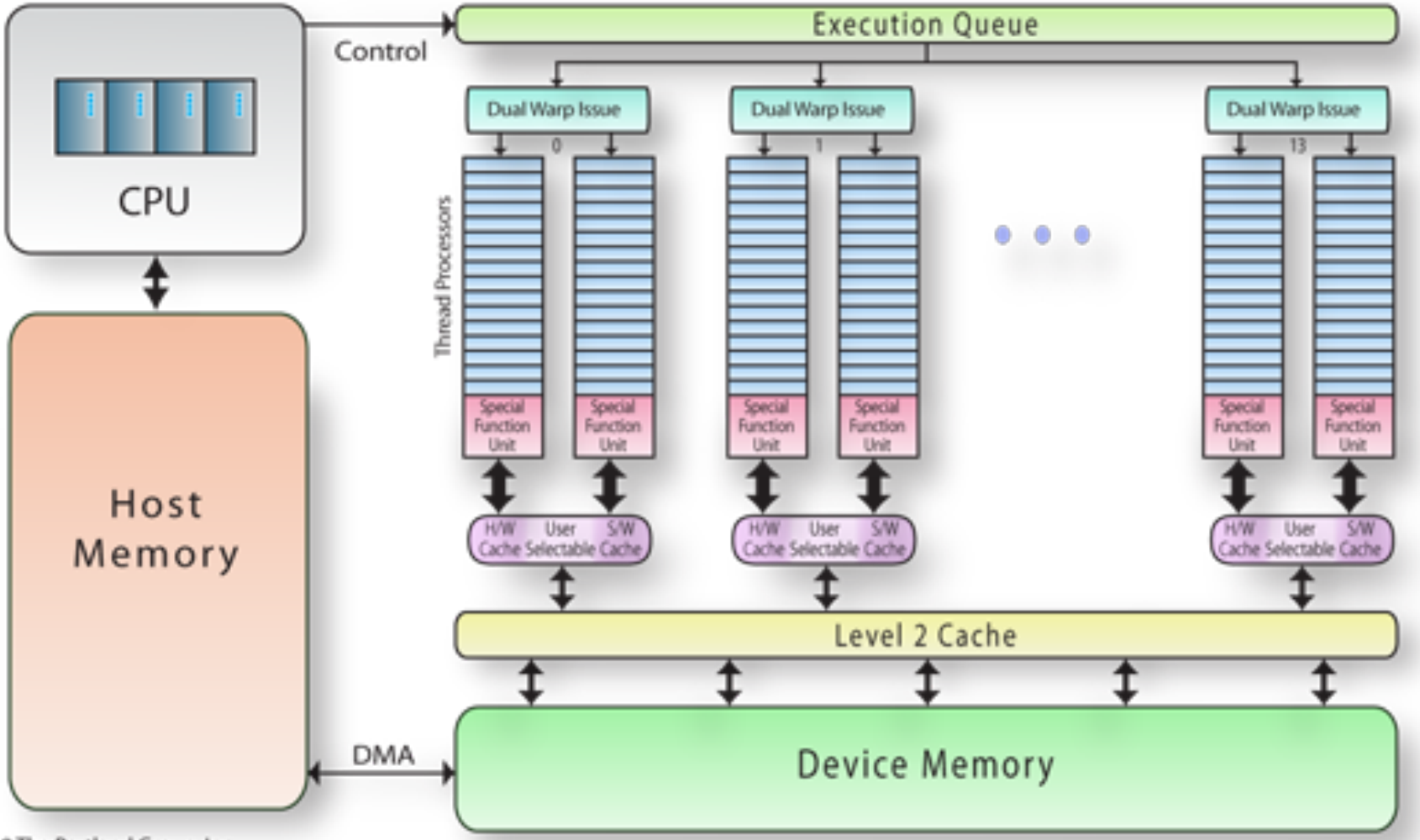
Few complex cores
Lots of on-chip memory
Lots of control logic

GPU

many
simple cores,
little memory,
little control



A GPU Architecture



ADVANCED CUDA CONCEPTS

Thread Scheduling

- Order of threads within a block is undefined!
  - Threads are grouped in warps (32 threads/warp)
    - AMD calls it “a wavefront” (64 threads/wavefront)
- Order in which thread blocks are mapped and scheduled is undefined!
  - Blocks run to completion on one SM without preemption
  - Can run in any order
    - Any possible interleaving of blocks should be valid
  - Can run concurrently OR sequentially

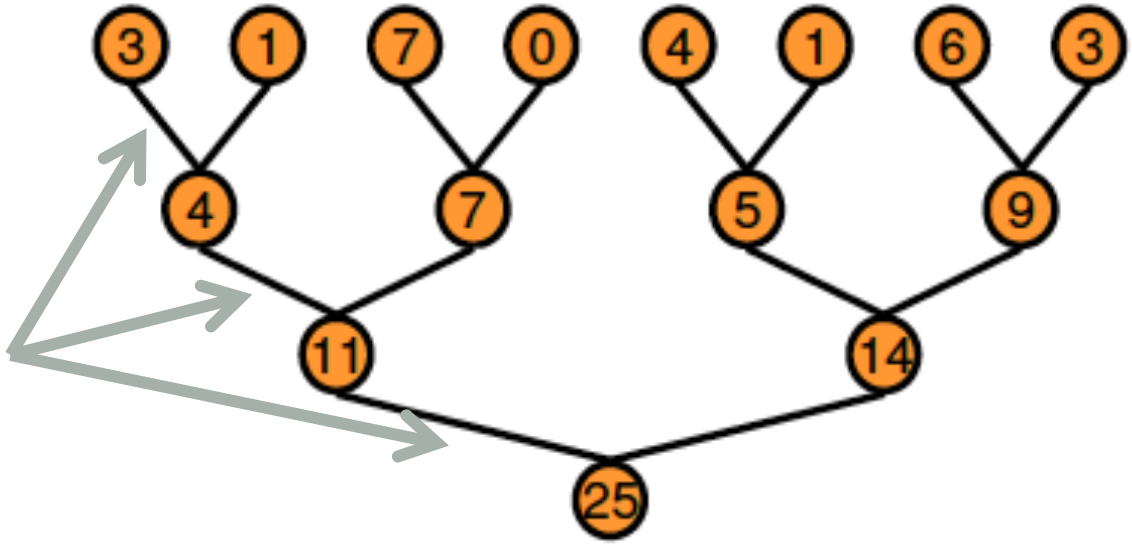
Global synchronization

- We launch many more blocks than physical SM's.
- Each block might/should have more threads than the SM's cores

```
__global__ void my_kernel() {
    step1; // compute some values in a global array
    // wait for *all* threads to finish
    __my_global_barrier();
    step2; // use the array
}

int main() {
  dim3 blockSize(32, 32);
  dim3 gridSize(100, 100, 100);
  my_kernel<<<gridDim, blockDim>>>();
}
```

An example: parallel reduction

- Given an array with data, “reduce” it to a single value
  - The sum of all elements
  - The min/max of all elements
- Sequentially: O(n)
- In parallel?
  - Tree-based algo.
    - O(log n)
  - Requires a barrier after each step

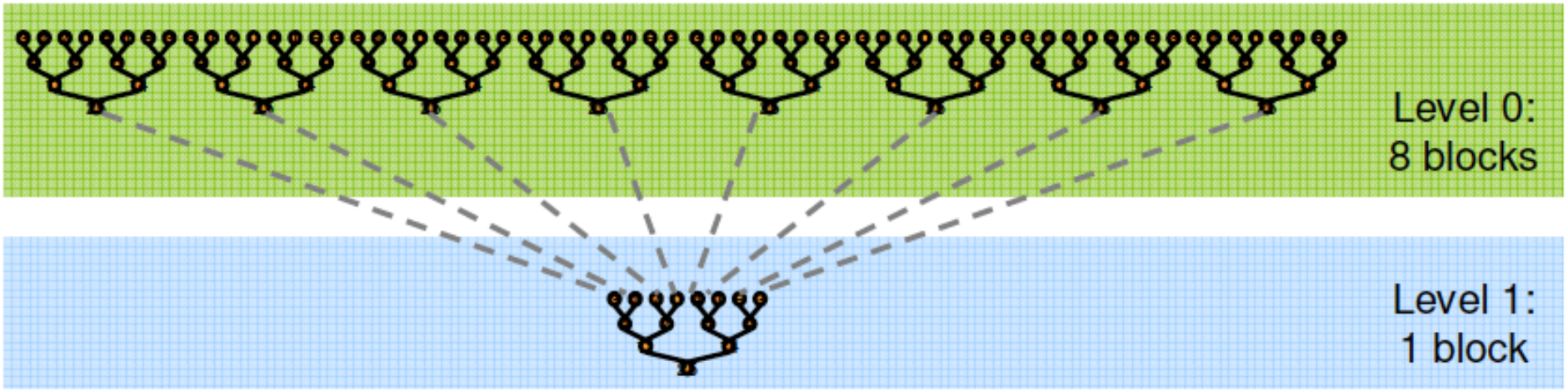


Parallel reduction in CUDA*

- One element per thread
- We need to use multiple blocks
  - Large arrays
  - Good GPU utilization
- We need global synchronization
  - Synchronization inside blocks is possible.
  - Synchronization between blocks is not possible!
- Solution: decompose into multiple kernels
  - Kernel launch serves as a global synchronization point
  - Kernel launch has negligible HW overhead, low SW overhead

*http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf

Parallel reduction in CUDA*



- Other optimizations
  - Use shared memory
  - Increase granularity
  - Avoid branching
  - Improve data access patterns

*http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf

Memory consistency

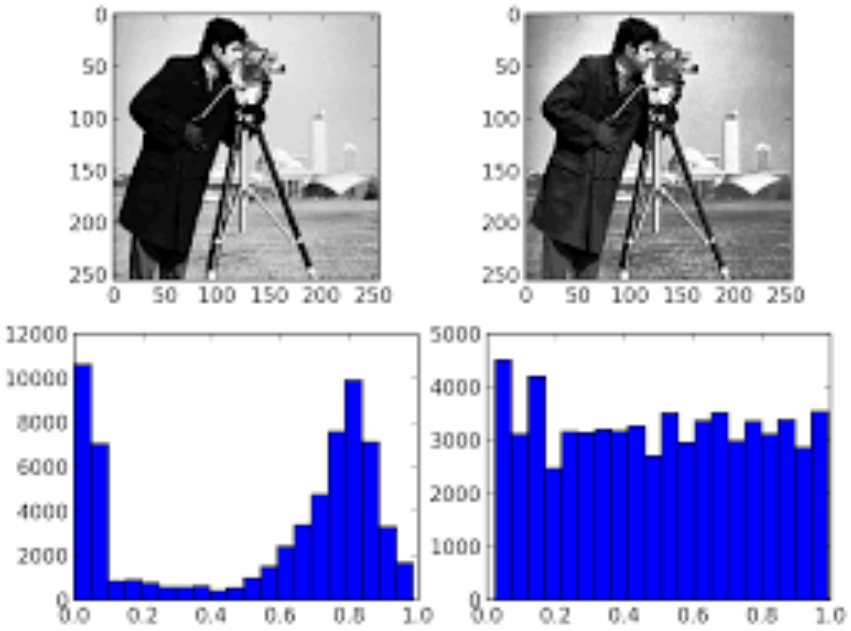
- Device (global) memory is not serially consistent
  - No ordering guarantees in shared/global memory Rd/Wr
- Share data between streaming multiprocessors
  - Potential write hazards!
- Use **atomics** to avoid data races for global (and shared) memory variables!
- Evolution:
  - Fermi has reasonable atomics for both shared and global memory
  - Kepler increases *global memory atomics* performance vs. Fermi
  - Maxwell uses native support for shared memory atomics
    - Much faster than Fermi and Kepler

Atomics

- Guarantee that only a single thread has access to a piece of memory during an operation
  - Ordering is still arbitrary
- Different types of atomic instructions
  - Add, Sub, Exch, Min, Max, Inc, Dec, CAS, And, Or, Xor
- Both for device memory and shared memory
- Much more expensive than load + operation + store

An example: image histogram

- The histogram of an image: the distribution of the pixels in the image.
  - In practice: count the pixels of each color
  - Useful image feature detection for image recognition.



An example: image histogram

```
// Determine frequency of colors in a picture.
// Colors have already been converted into integers
// between 0 and 255.
// Each thread looks at one pixel,
// and increments a counter

__global__ void histogram(int* colors, int* buckets)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int c = colors[i];
    buckets[c] += 1;  // incorrect!
}
```

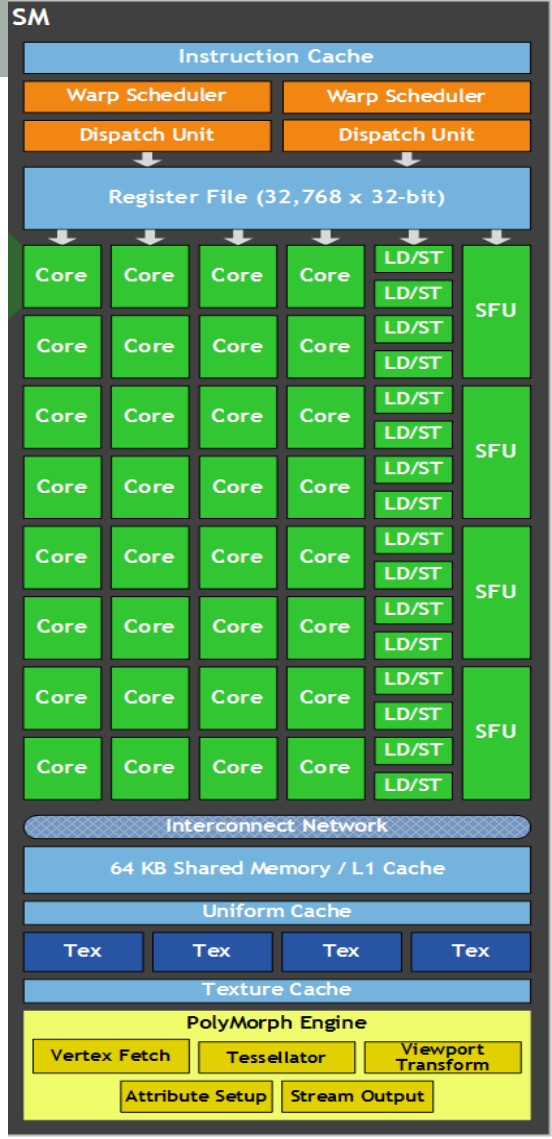
An example: image histogram

```
// Determine frequency of colors in a picture.
// Colors have already been converted into integers
// between 0 and 255.
// Each thread looks at one pixel,
// and increments a counter atomically

__global__ void histogram(int* colors, int* buckets)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int c = colors[i];
    atomicAdd(&buckets[c], 1);
}
```

OCCUPANCY

Warps = 32 threads

- Threads are scheduled in warps
  - AMD calls them “wavefronts”
- One warp => on one SM
  - Same SM till completion
- Scheduling
  - GigaThread Unit : schedules blocks per SM’s
  - Inside SM: warp scheduler(s) + instruction dispatcher
    - Replace warps that are stalled by warps waiting to compute
    - Very fast context switching

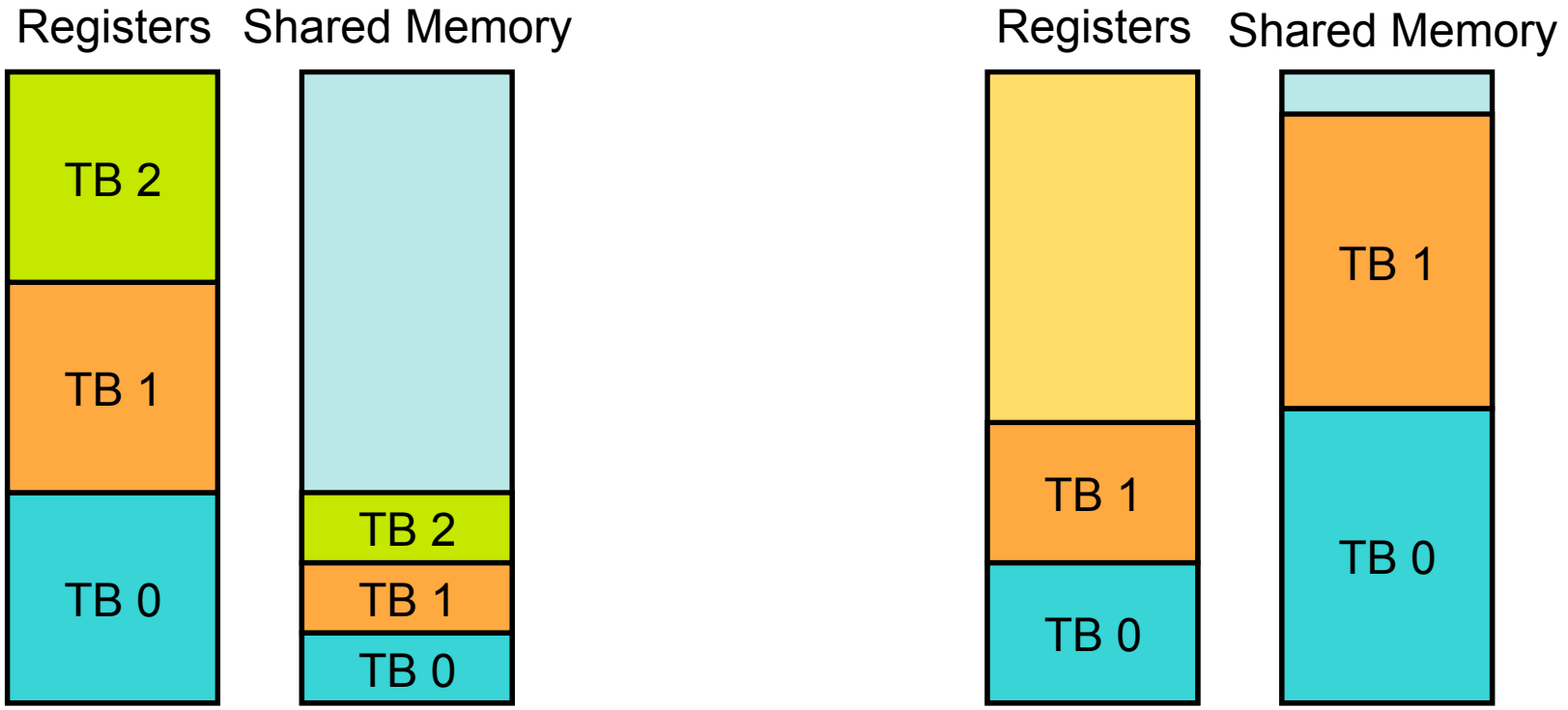


Occupancy

Occupancy = Active Warps / Maximum Active Warps

- Remember: resources are allocated for the entire block!
- Resources are finite
  - Utilizing too many resources per thread may limit the occupancy
- Potential occupancy limiters:
  - Register usage
  - Shared memory usage
  - Block size

Resource Limits



- Pool of registers and shared memory per SM
  - Each thread block grabs registers & shared memory
  - If one or the other is fully utilized => no more thread blocks

How do you know what you're using?

- Use compiler flags to get register and shared memory usage
  - "`nvcc -Xptxas -v`"
- Use the NVIDIA Profiler
- Plug those numbers into CUDA Occupancy Calculator

- Maximize occupancy for improved performance
  - Empirical rule! Don't overuse!

CUDA_Occupancy_calculator.xlsm - Microsoft Excel

Home | Insert | Page Layout | Formulas | Data | Review | View

Security Warning Macros have been disabled. Options...

MyRegCount | 25

1.) Select Compute Capability (click):	1.3	(Help)
2.) Enter your resource usage:		
Threads Per Block	128	(Help)
Registers Per Thread	25	
Shared Memory Per Block (bytes)	640	
(Don't edit anything below this line)		
3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	512	(Help)
Active Warps per Multiprocessor	16	
Active Thread Blocks per Multiprocessor	4	
Occupancy of each Multiprocessor	50%	

Physical Limits for GPU Compute Capability:	1.3
Threads per Warp	32
Warps per Multiprocessor	32
Threads per Multiprocessor	1024
Thread Blocks per Multiprocessor	8
Total # of 32-bit registers per Multiprocessor	16384
Register allocation unit size	512
Register allocation granularity	block
Shared Memory per Multiprocessor (bytes)	16384
Shared Memory Allocation unit size	512
Warp allocation granularity (for register allocation)	2

Allocation Per Thread Block	
Warps	4
Registers	3584
Shared Memory	1024

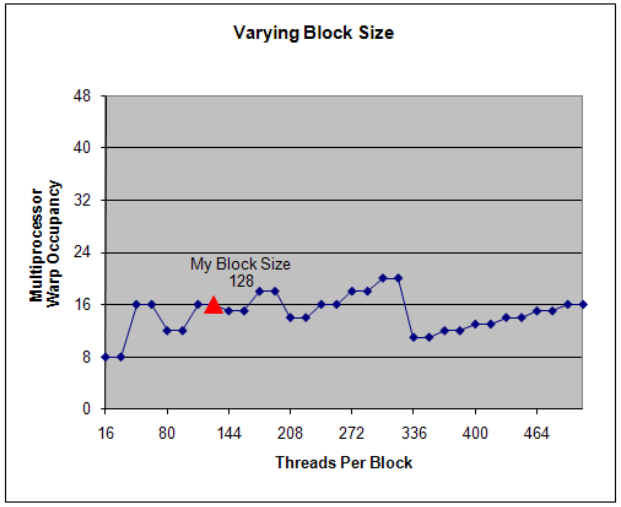
These data are used in computing the occupancy data in blue

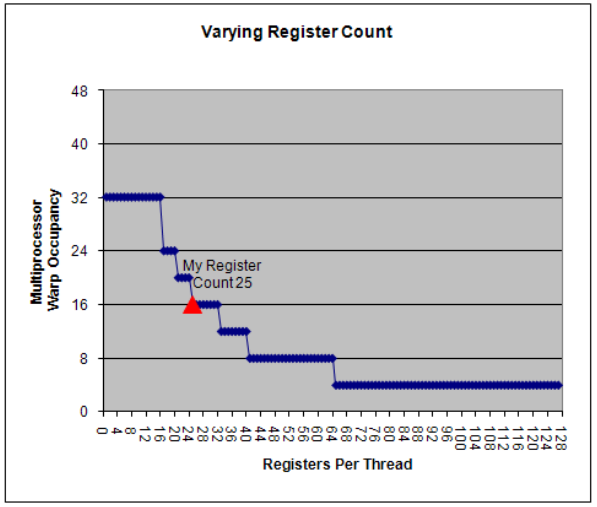
Maximum Thread Blocks Per Multiprocessor	Blocks
Limited by Max Warps / Blocks per Multiprocessor	8
Limited by Registers per Multiprocessor	4
Limited by Shared Memory per Multiprocessor	16

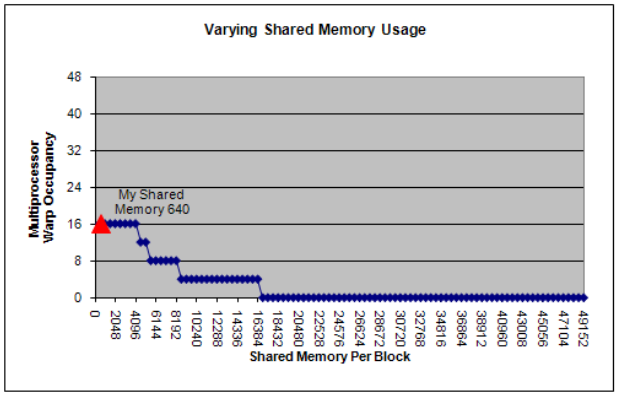
Thread Block Limit Per Multiprocessor highlighted RED

CUDA Occupancy Calculator	
Version:	2.0
Copyright and License	

The other data points represent the range of possible block sizes, register counts, and shared memory allocation.







Calculator | Help | GPU Data | Copyright & License

Ready

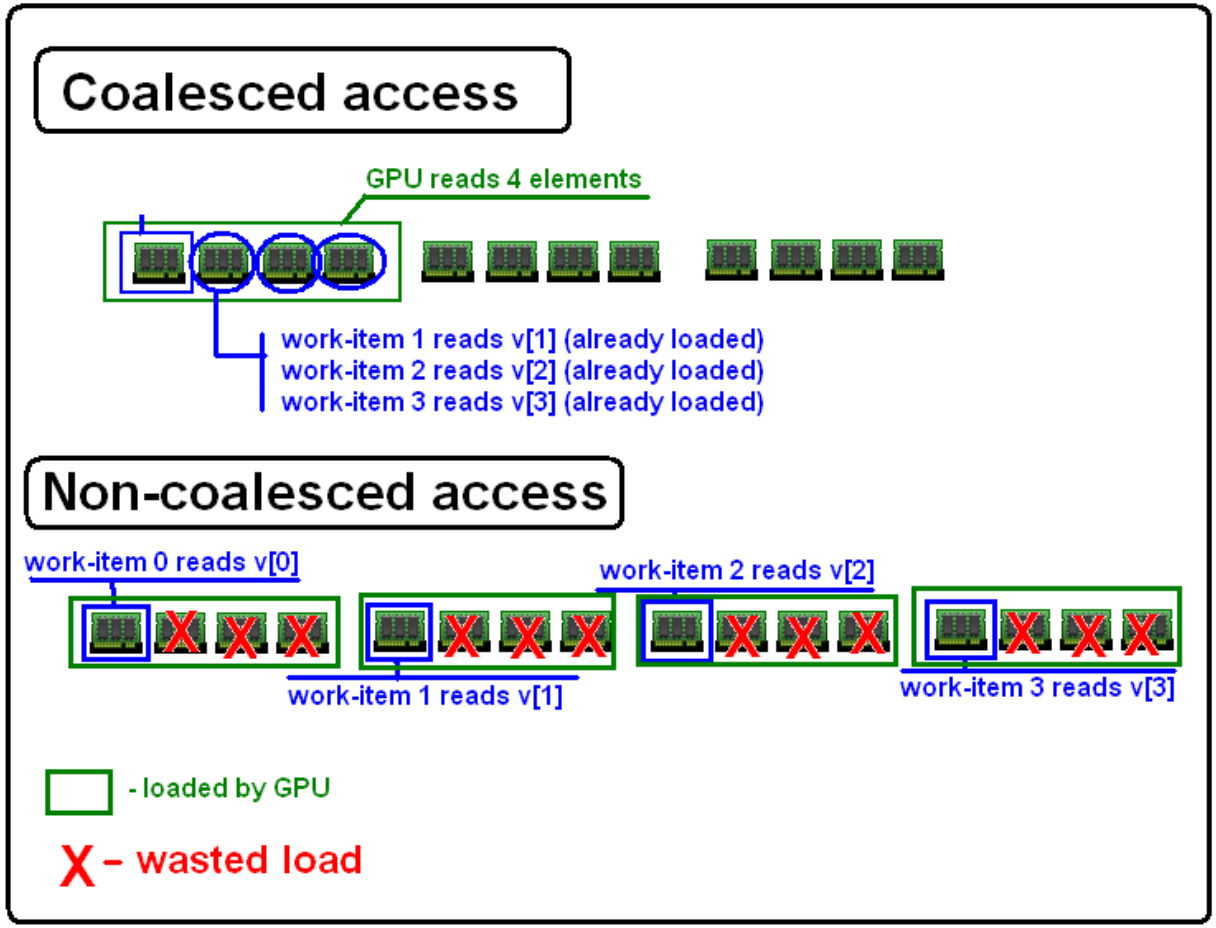
Thread divergence - penalty?

- Depends on the amount of divergence
  - Worst case: 1/32 performance
    - When each thread does something different
- Depends on whether branching is data- or ID- dependent
  - If ID – consider grouping threads differently
  - If data – consider sorting
- Non-diverging warps => NO performance penalty
  - In this case, branches are not expensive …

CUDA: MEMORY COALESCING

Memory Coalescing

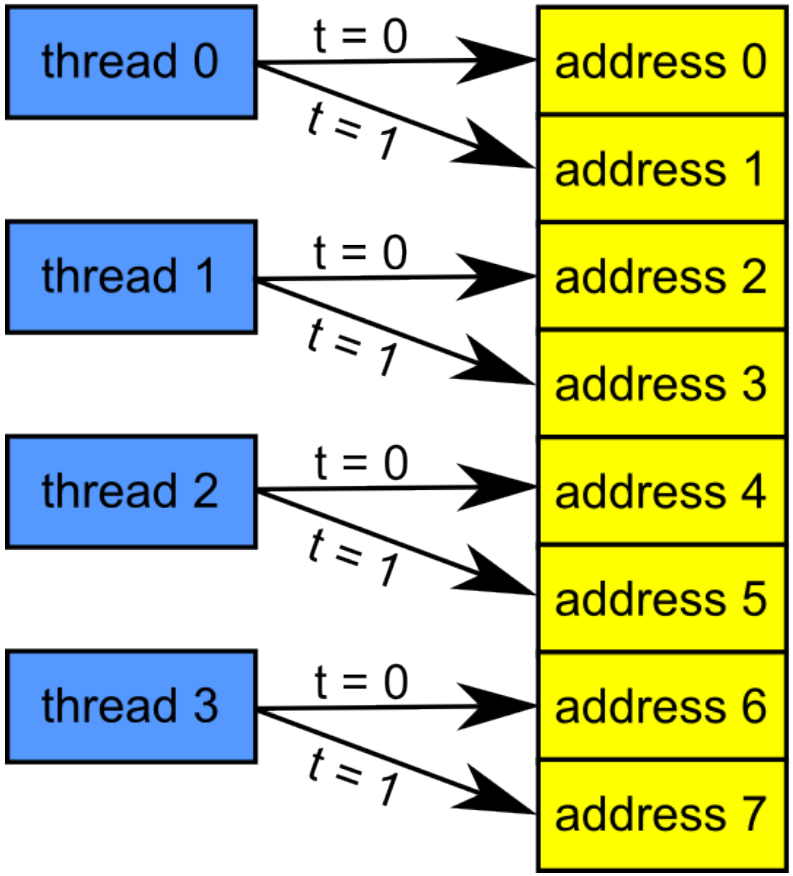
- **Memory coalescing** refers to combining multiple **memory** accesses into a single transaction



Caching vs. Coalescing

traditional multi-core

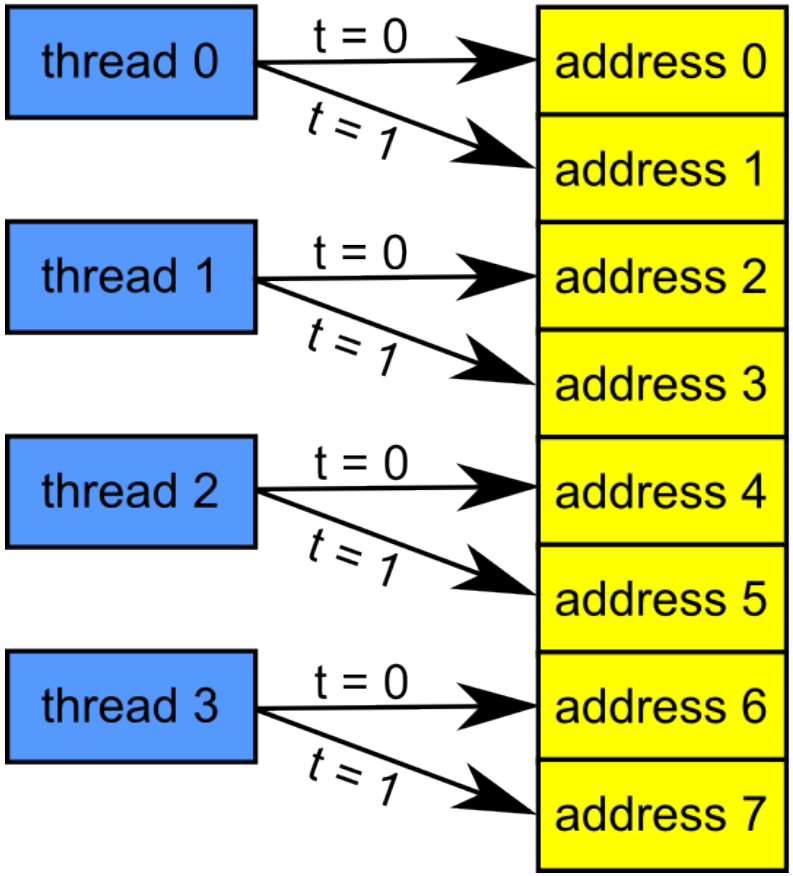
optimal memory access pattern



Caching

Caching vs. Coalescing

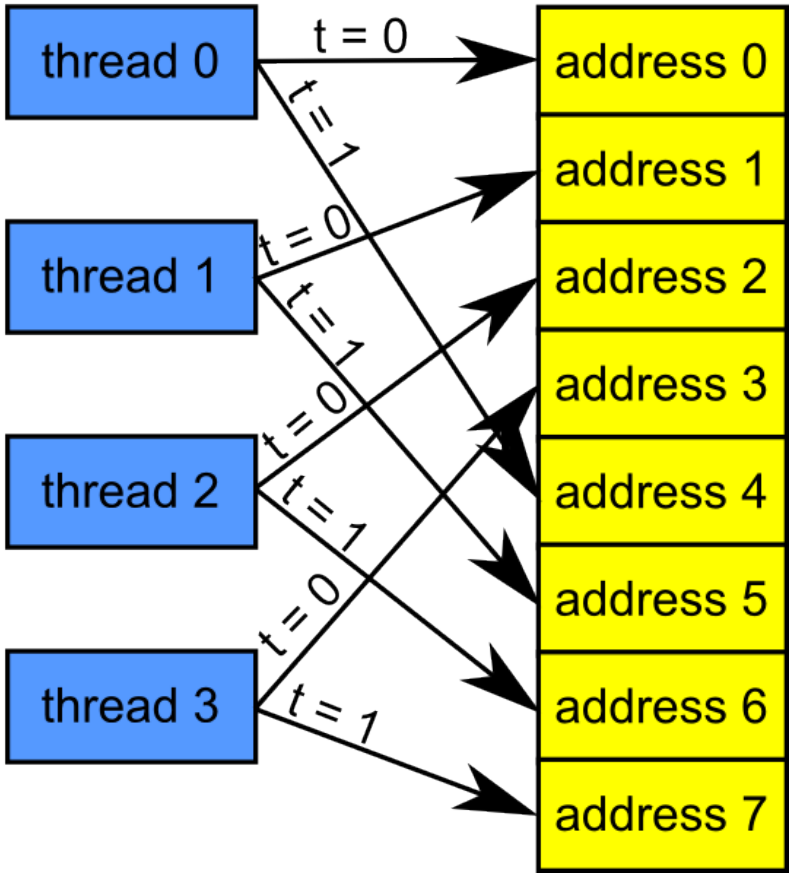
traditional multi-core
optimal memory access pattern



Caching

vs.

many-core GPU
optimal memory access pattern



Coalescin

Consider the stride of your accesses

	Input[0]	Input[1]	Input[2]	Input[3]	Input[4]	Input[5]	Input[6]	Input[7]
Stride1	T0	T1	T2	T3	T4	T5	T6	T7
Stride2		T0		T1		T2		T3
“random”			T7		T1			T4

```
__global__ void foo(int* input, float3* input2) {
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  // Stride 1, full bandwidth used!
  int a = input[i];
  // Stride 2, 50% of the bandwidth is wasted
  int b = input[2*i+1];
  // “Random” stride – ?? up to 7/8 bandwidth wasted
  int c = input[f(i)];
}
```

Example: Array of Structures (AoS)

```
Struct AoS{
    int key;
    int value;
    int flag;
};

record *d_AoS_data;
cudaMalloc((void**)&d_AoS_data, ...);

kernel {
 threadID = blockDim.x * blockIdx.x + threadIdx.x;
 // …
 d_AoS_data[threadID].value += i; // wastes bandwidth!
 // …
}
```

Example: Structure of Arrays (SoA)

```
Struct SoA {
    int* keys;
    int* values;
    int* flags;
};

SoA d_SoA_data;
cudaMalloc((void**)&d_SoA_data.keys, ...);
cudaMalloc((void**)&d_SoA_data.values, ...);
cudaMalloc((void**)&d_SoA_data.flags, ...);

kernel {
  threadID = blockDim.x * blockIdx.x + threadIdx.x;
…
  d_SoA_data.values[threadID] += i; // full
 bandwidth!
… }
```

Memory Coalescing*

- Group memory accesses in as few memory transactions as possible.
  - 128-byte or 32-byte long lines
- Stride 1 access patterns are preferred!
  - Other patterns can still get benefits
- Structure of arrays is often better than array of structures
- Unpredictable/ irregular access patterns
  - Case-by-case performance impact
- No coalescing => performance loss ~10x or more !
  - Caching might improve this impact …

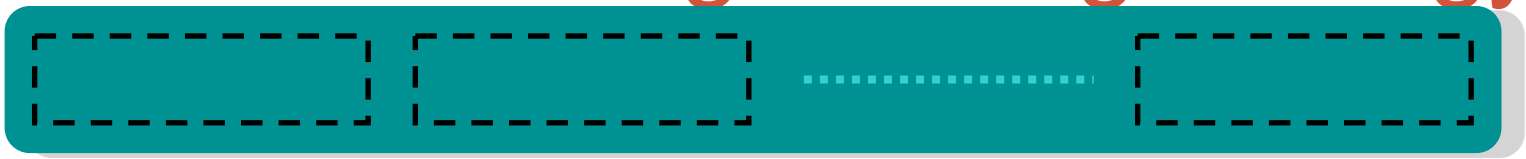
*http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#axzz3nJ0kBsWe

CUDA: USING SHARED MEMORY

Using shared memory

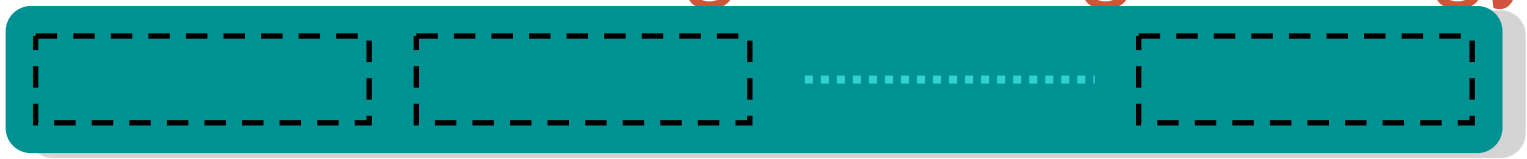
- Equivalent with providing software caching
  - Explicit: Load data to be re-used in shared memory
  - Use it for computation
  - Explicit: Store results back to global memory
- All threads in a block share memory
  - Load/Store: using all threads
  - Barrier: __syncthreads
    - Guard against using uninitialized data – not all threads have finished loading data to shared memory
    - Guard against corrupting live data – not all threads have finished computing

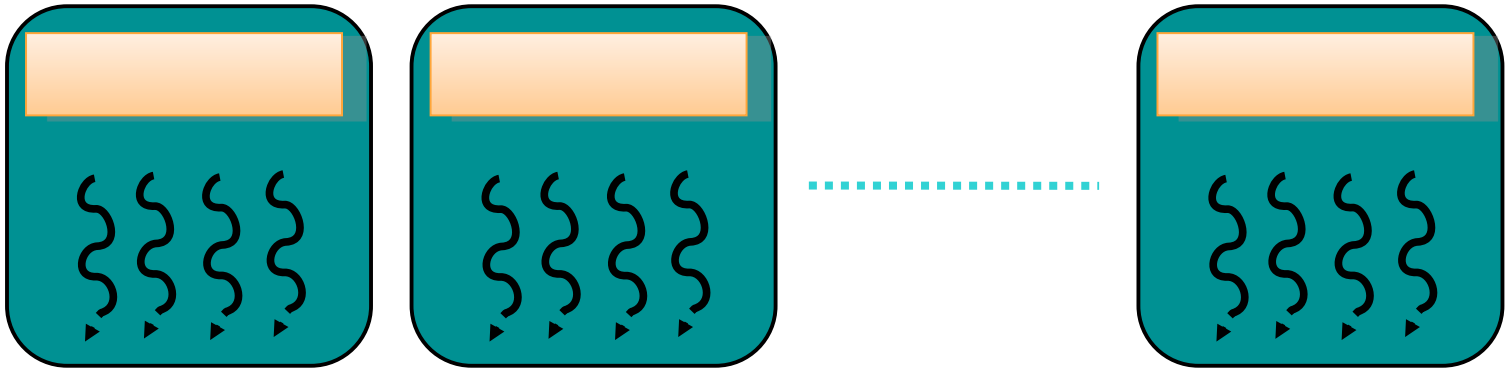
A Common Programming Strategy



- Partition data into subsets that fit into shared memory

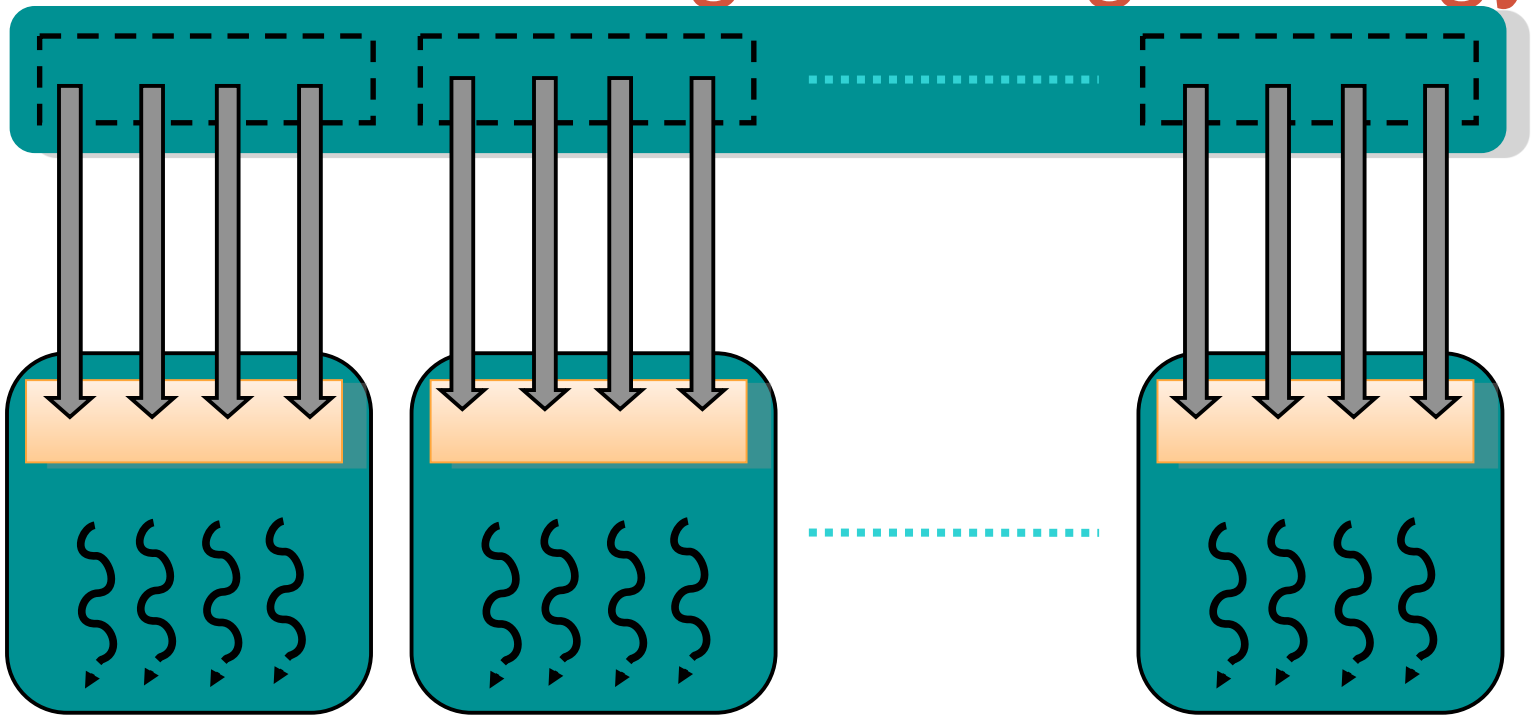
A Common Programming Strategy





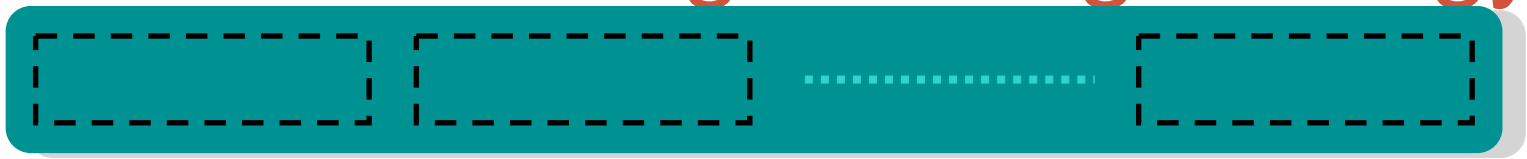
- Handle each data subset with one thread block

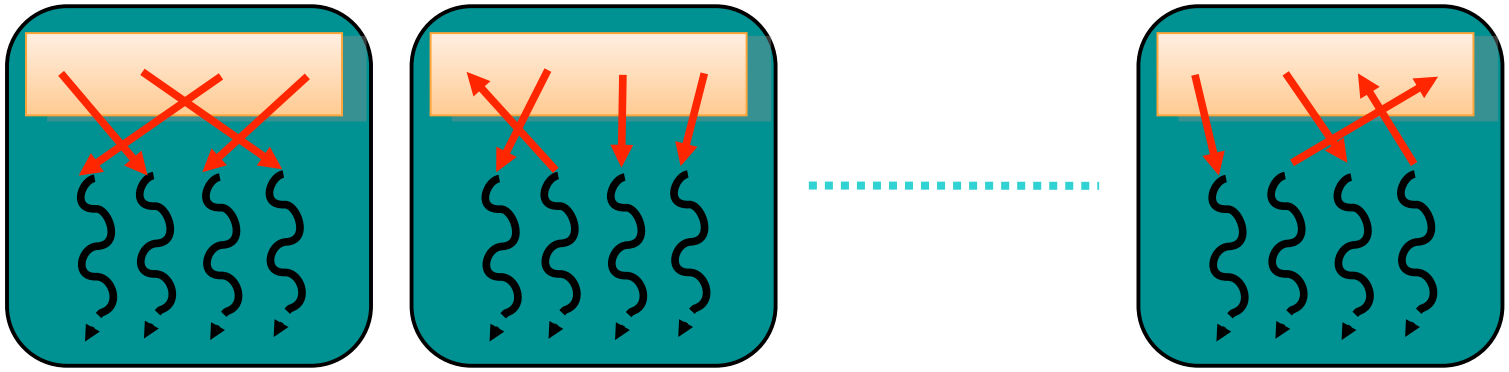
A Common Programming Strategy



- Load the subset from device memory to shared memory, using multiple threads to exploit memory-level parallelism

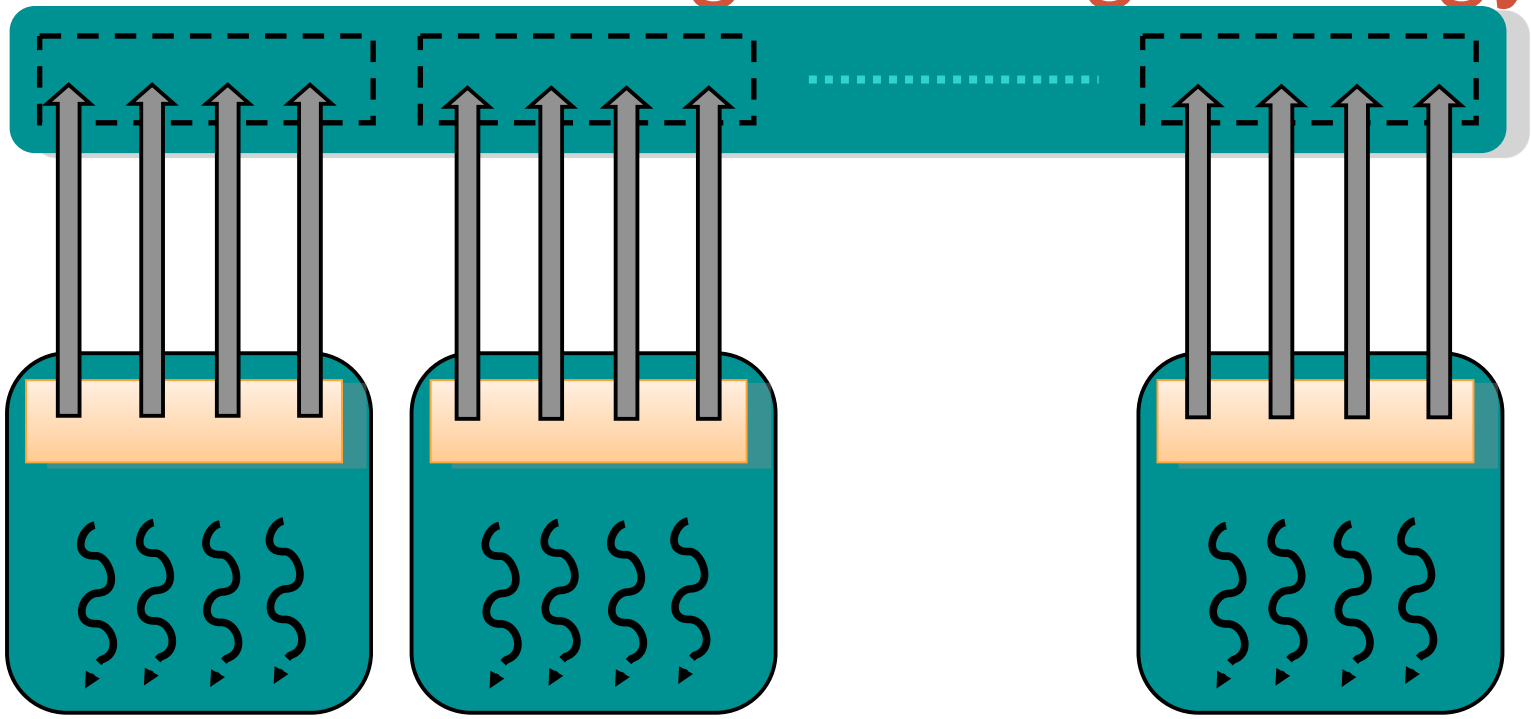
A Common Programming Strategy





- Perform the computation on the subset from shared memory

A Common Programming Strategy



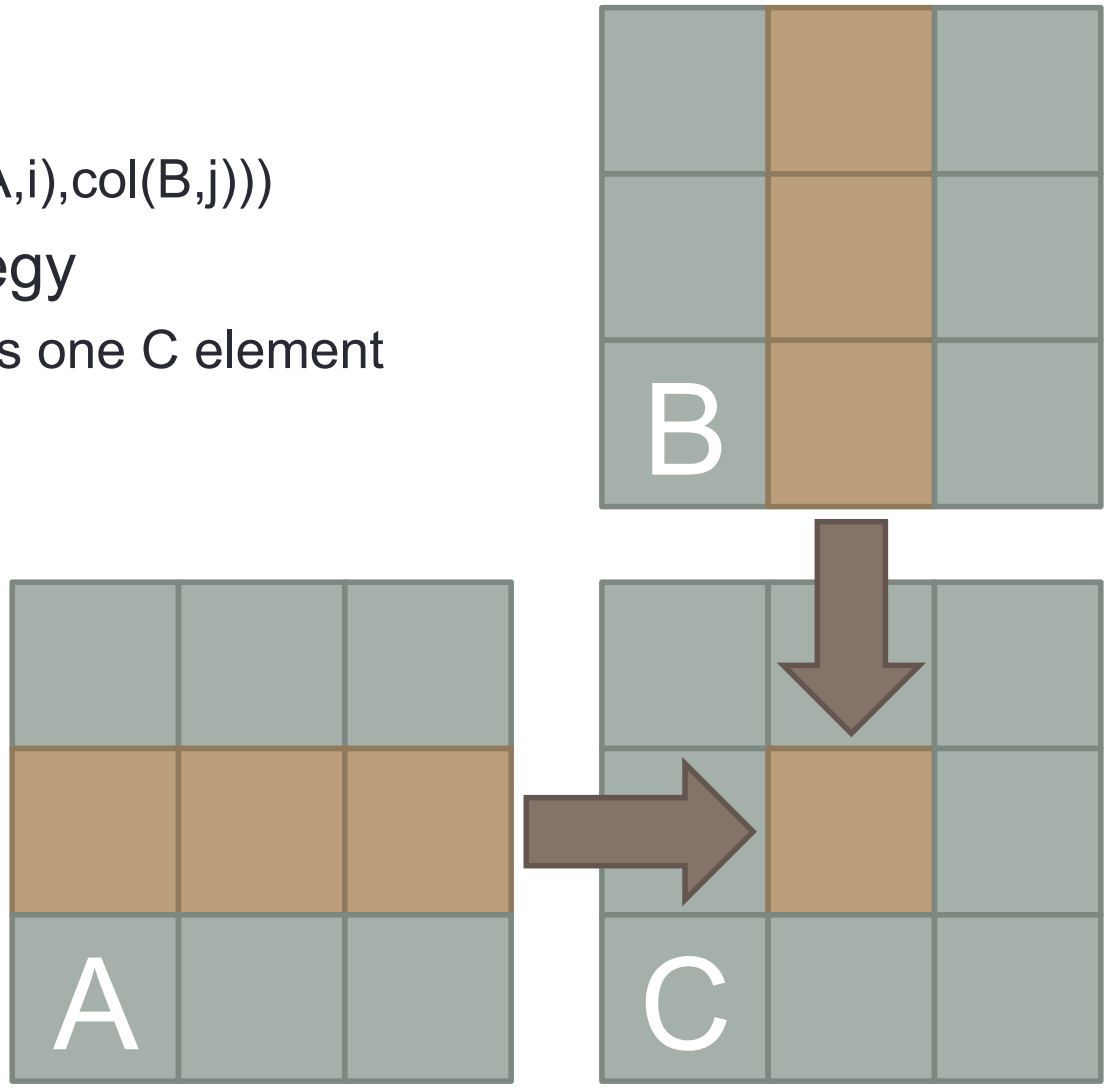
- Copy the result from shared memory back to device memory

Caches vs. Shared Memory

- Since Fermi, NVIDIA GPUs feature BOTH hardware **L1 caches** and **shared memory** per SM
  - They share the same space
    - ¾ Cache + ¼ Shared Memory OR
    - ¼ Cache + ¾ Shared Memory
- L1 Cache
  - Hardware caching enabled
    - The HW decides what goes in or out and when
- Shared memory
  - Software manages what goes in/out
  - Allows more complex access patterns to be cached

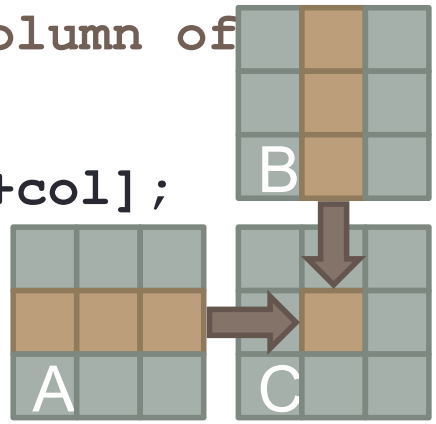
Example: Matrix multiplication

- C = A * B
  - C(i,j) = sum(dot(row(A,i),col(B,j)))
- Parallelization strategy
  - Each thread computes one C element
  - 2D kernel



Matrix multiplication implementation

```
__global__ void mat_mul(float *a, float *b,
                        float *c, int width)
{
  // calc row & column index of output element
  int row = blockIdx.y*blockDim.y + threadIdx.y;
  int col = blockIdx.x*blockDim.x + threadIdx.x;

 float result = 0;

  // do dot product between row of a and column of
  for(int k = 0; k < width; k++) {
      result += a[row*width+k] * b[k*width+col];
 }
  c[row*width+col] = result;
}
```

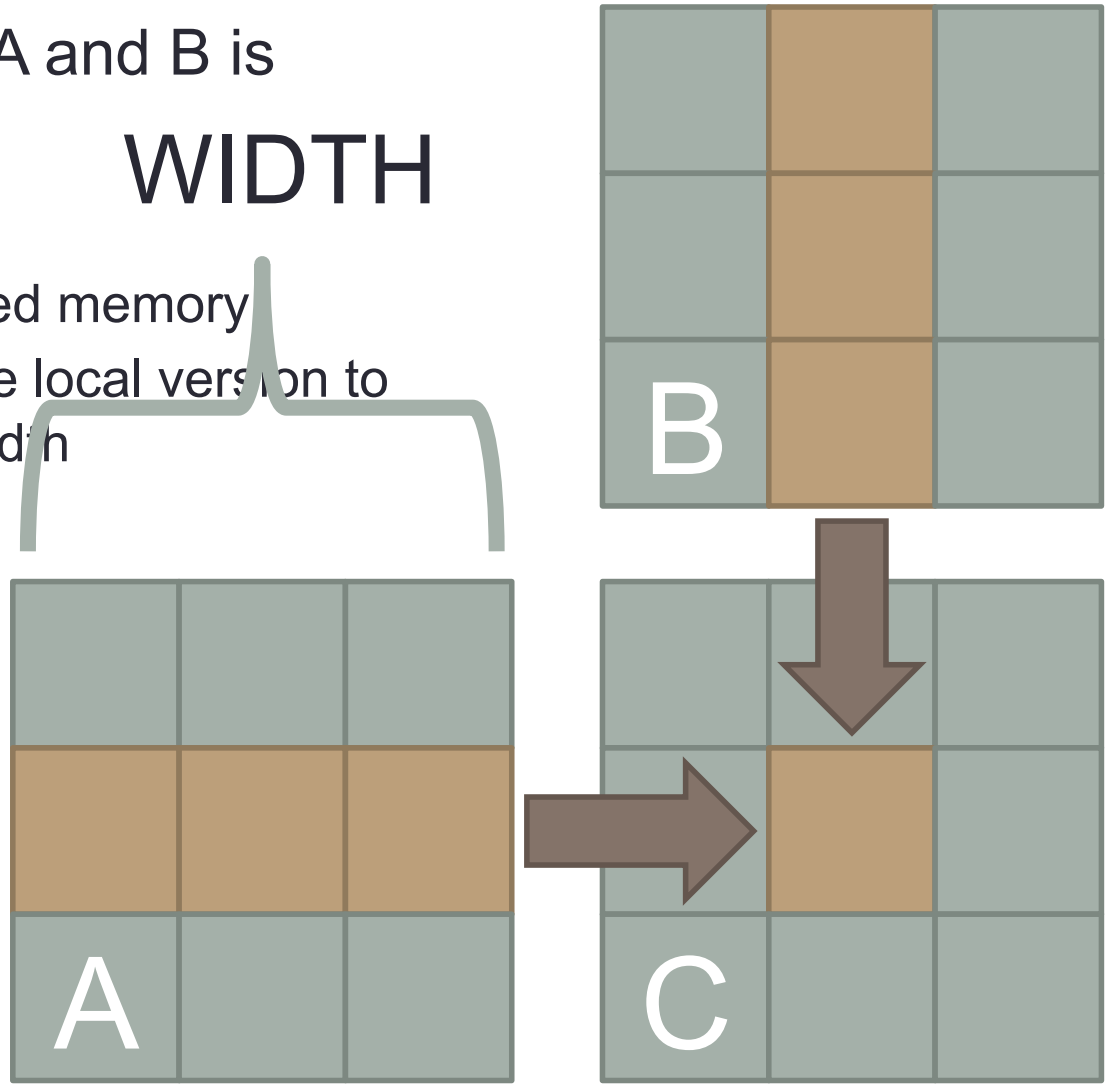


Matrix multiplication performance

Loads per dot product term	2 (a and b) = 8 bytes
FLOPS	2 (multiply and add)
AI	2 / 8 = 0.25
Performance GTX 580	1581 GFLOPs
Memory bandwidth GTX 580	192 GB/s
Attainable performance	192 * 0.25 = 48 GFLOPS
Maximum efficiency	3.0 % of theoretical peak

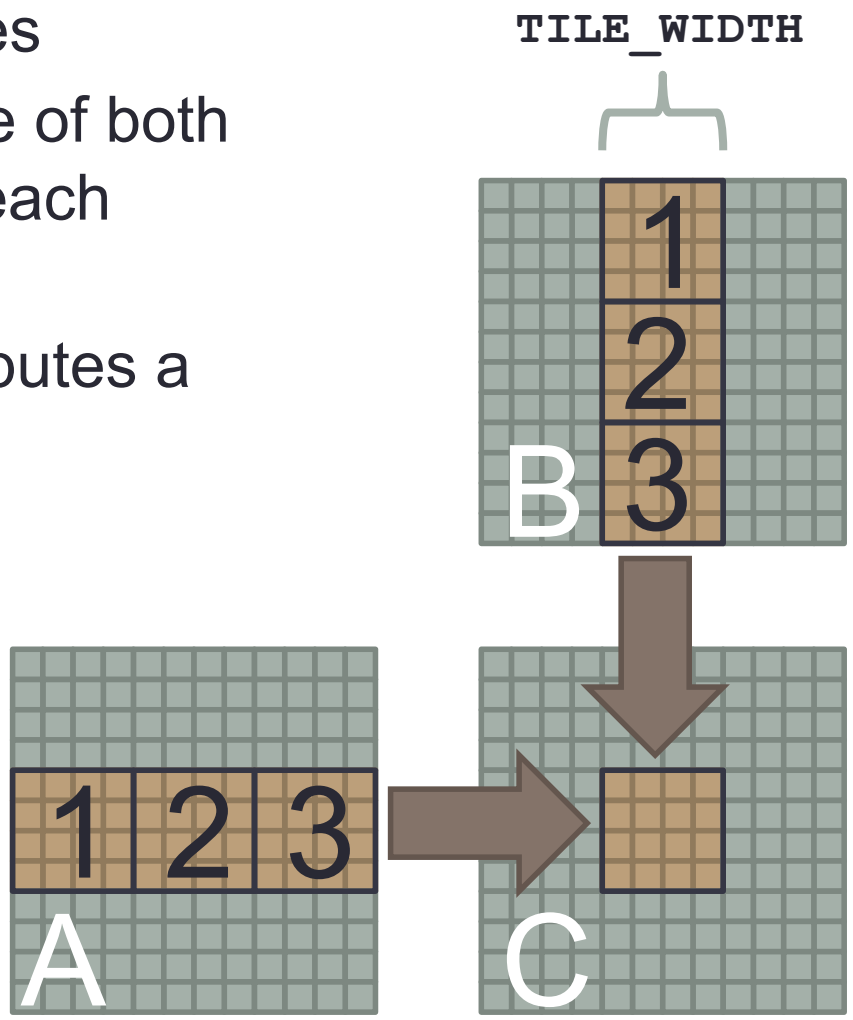
Data reuse

- Each input element in A and B is read WIDTH times
- IDEA:
  - Load elements into shared memory
  - Have several threads use local version to improve memory bandwidth



Using shared memory

- Partition kernel loop into phases
- In each thread block, load a tile of both matrices into shared memory each phase
- Each phase, each thread computes a partial result

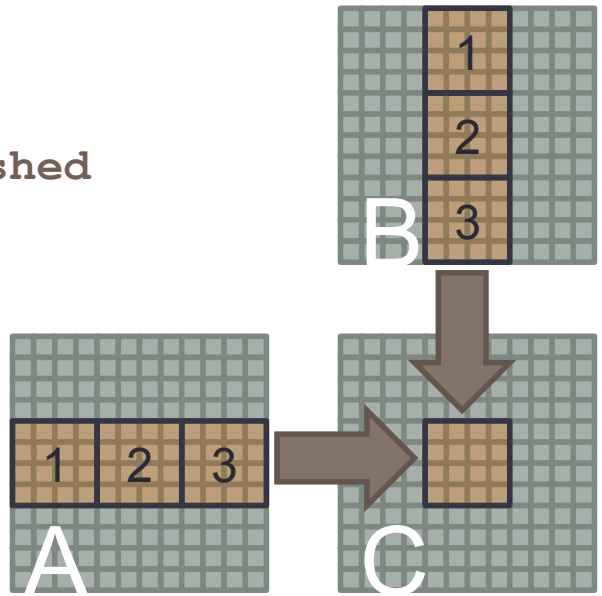


Matrix multiply with shared memory

```
__global__ void mat_mul(float *a, float *b,
                        float *c, int width) {
  // shorthand
  int tx = threadIdx.x, ty = threadIdx.y;
  int bx = blockIdx.x,  by = blockIdx.y;

  // allocate tiles in shared memory
  __shared__ float s_a[TILE_WIDTH][TILE_WIDTH];
  __shared__ float s_b[TILE_WIDTH][TILE_WIDTH];

  // calculate the row & column index from A,B
  int row = by*blockDim.y + ty;
  int col = bx*blockDim.x + tx;

 float result = 0;
```

Matrix multiply with shared memory

```
  // loop over input tiles in phases, p = crt. phase
  for(int p = 0; p < width/TILE_WIDTH; p++) {
    // collaboratively load tiles into shared memory
    s_a[ty][tx] = a[row*width + (p*TILE_WIDTH + tx)];
    s_b[ty][tx] = b[(p*TILE_WIDTH + ty)*width + col];
// barrier: ALL writes to shared memory finished
    __syncthreads();

    // dot product between row of s_a and col of s_b
    for(int k = 0; k < TILE_WIDTH; k++) {
      result += s_a[ty][k] * s_b[k][tx];
    }
// barrier: ALL reads of shared memory finished
    __syncthreads();
  }

  c[row*width+col] = result;
}
```



Use of Barriers in mat_mul

- Two barriers per phase:
  - __syncthreads after all data is loaded into shared memory
  - __syncthreads after all data is read from shared memory
    - Second __syncthreads in phase p guards the load in phase p+1

- Formally, __synchthreads is a barrier for shared memory for a block of threads:

"void __syncthreads();

waits until all threads in the thread block have reached this point **and** all global and shared memory accesses made by these threads prior to __syncthreads() are visible to all threads in the block."

Matrix multiplication performance

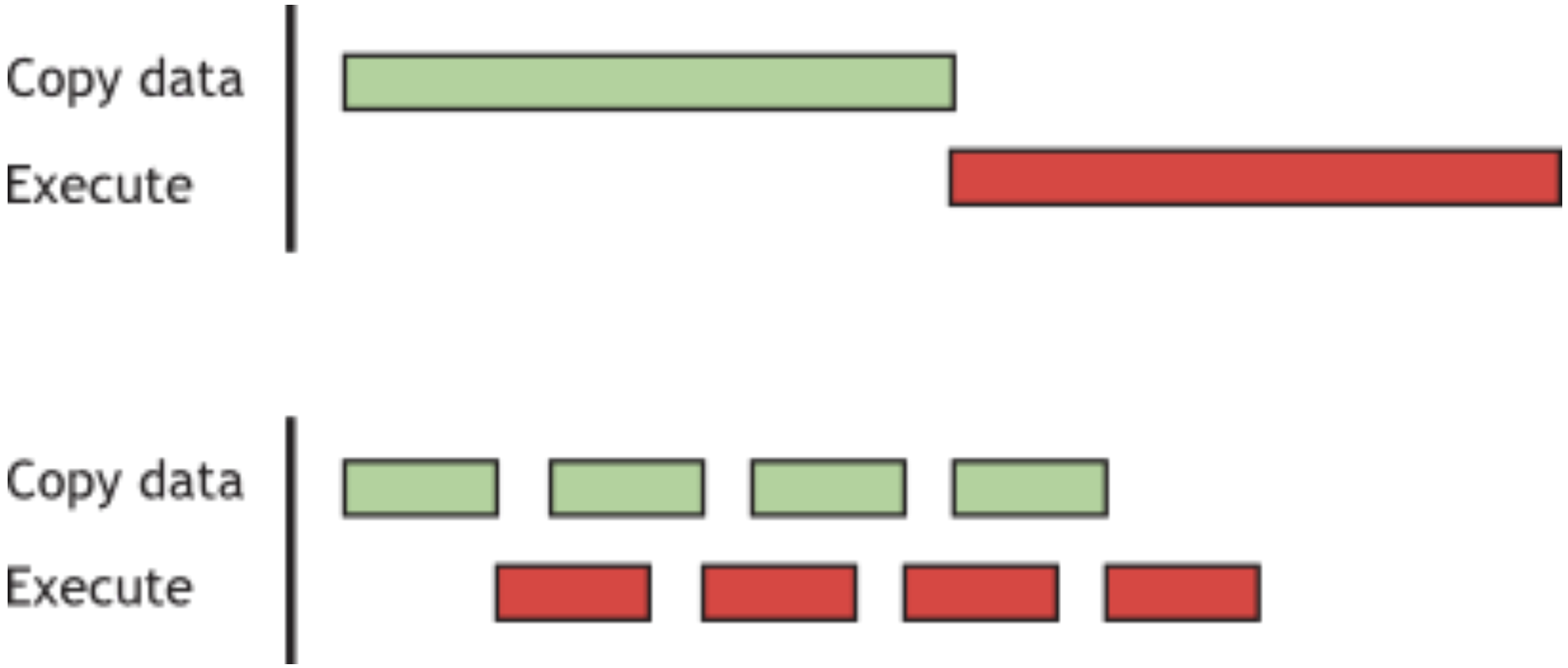
	Original	shared memory
Global loads	$2N^3$ * 4 bytes	($2N^3$ / TILE_WIDTH) * 4 bytes
Total ops	$2N^3$	$2N^3$
AI	0.25	0.25 * TILE_WIDTH

Performance GTX 580	1581 GFLOPs
Memory bandwidth GTX 580	192 GB/s
AI needed for peak	1581 / 192 = **8.23**
TILE_WIDTH required to achieve peak	0.25 * TILE_WIDTH = **8.23**, TILE_WIDTH = 32.9

CUDA: STREAMS

Overlap Computation and Communication

- Main idea: while executing a kernel, bring data in for the next kernel:



What are streams?

- **Stream** = a sequence of operations that execute on the device in the order in which they are issued by the host code.

- Same stream: In-Order execution
- Different streams: Out-of-Order execution

- Default stream = Synchronizing stream
  - No operation in the default stream can begin until all previously issued operations in any stream on the device have completed.
  - An operation in the default stream must complete before any other operation in any stream on the device can begin.

Default stream: example

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

- All operations happen in the same stream
- Device (GPU)
  - Synchronous execution
    - all operations execute (in order), one after the previous has finished
  - Unaware of `CpuFunction()`
- Host (CPU)
  - Launches `increment` and regains control
  - *May* execute `CpuFunction` *before* increment has finished
  - Final copy starts *after* both `increment` and `CpuFunction()` have finished

Non-default streams

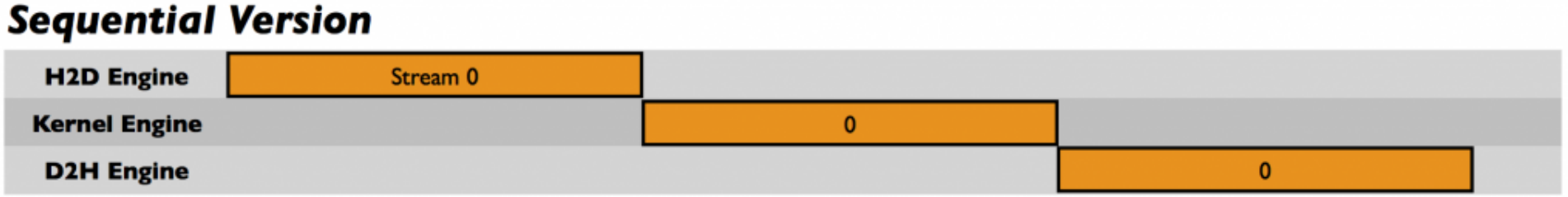
- Enable asynchronous execution and overlaps
  - Require special creation/deletion of streams
    - `cudaStreamCreate(&stream1)`
    - `cudaStreamDestroy(stream1)`
  - Special memory operations
    - `cudaMemcpyAsync(deviceMem, hostMem, size, cudaMemcpyHostToDevice, stream1)`
  - Special kernel parameter (the 4th one)
    - `increment<<<1, N, 0, stream1>>>(d_a)`
- Synchronization
  - All streams
    - `cudaDeviceSynchronize()`
  - Specific stream:
    - cudaStreamSyncrhonize(stream1)

Computation vs. communication

```
//Single stream, numBytes = 16M, numElements = 4M
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
kernel<<blocks,threads>>(d_a, firstElement);
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

C1060 (pre-Fermi): 12.9ms

C2050 (Fermi): 9.9ms

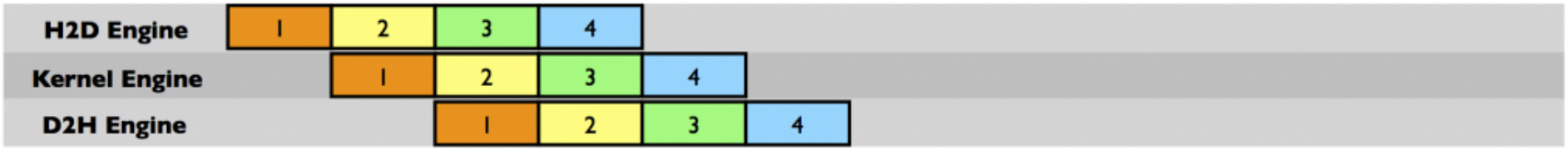


Computation-communication overlap[1]*

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

C1060 (pre-Fermi): 13.63 ms (worse than sequential)

C2050 (Fermi): 5.73 ms (better than sequential)



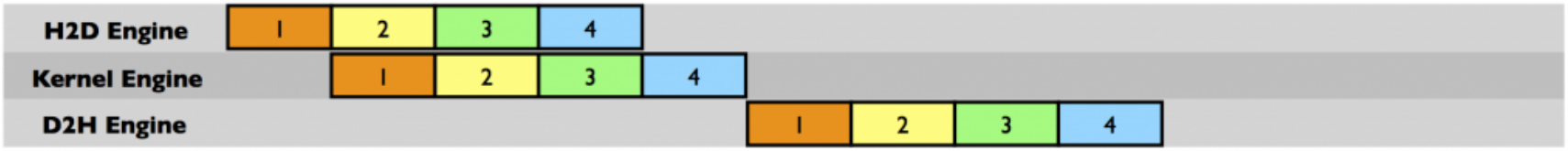
https://github.com/parallel-forall/code-samples/blob/master/series/cuda-cpp/overlap-data-transfers/async

Computation-communication overlap[2]*

```
for (int i = 0; i < nStreams; ++i) offset[i]=i * streamSize;
for (int i = 0; i < nStreams; ++i)
   cudaMemcpyAsync(&d_a[offset[i]], &a[offset[i]], streamBytes,
      cudaMemcpyHostToDevice, stream[i]);

for (int i = 0; i < nStreams; ++i)
   kernel<<blocks,threads,0,stream[i]>>(d_a, offset);

for (int i = 0; i < nStreams; ++i)
   cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes,
      cudaMemcpyDeviceToHost, stream[i]);
```

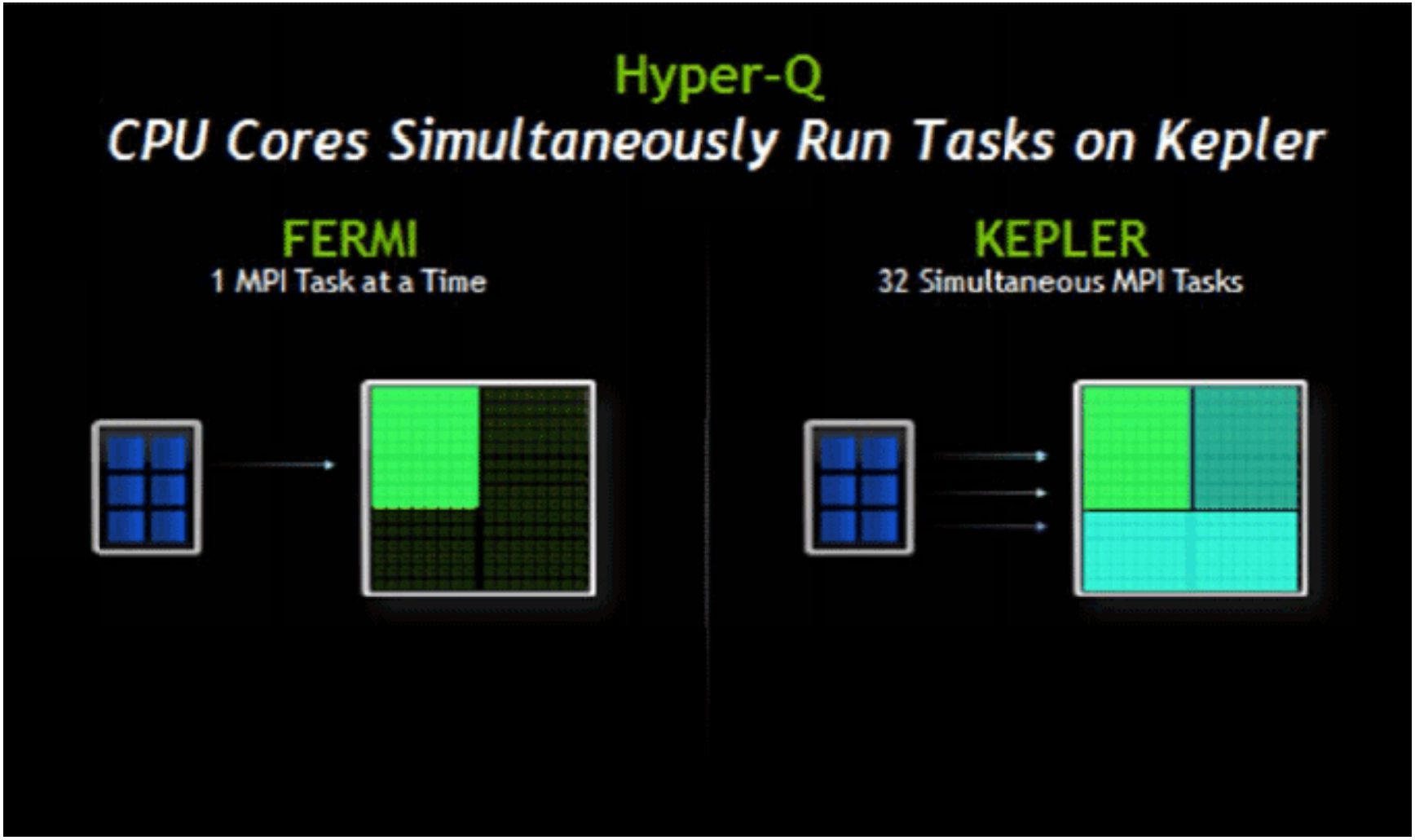
C1060 (pre-Fermi): 8.84 ms (better than sequential)

C2050 (Fermi): 7.59 ms (better than sequential, worse than v1)

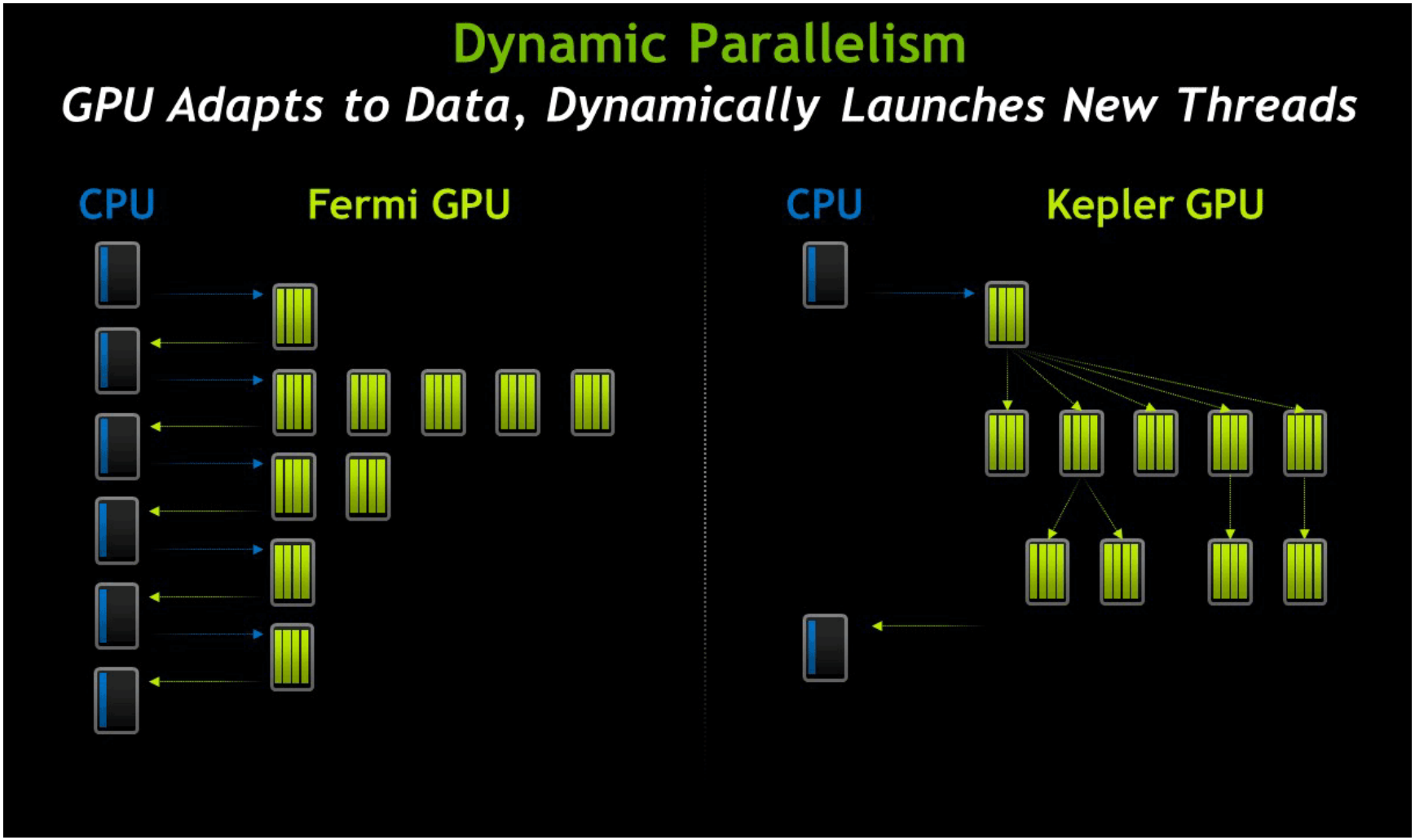


CUDA: MANY OTHER FEATURES

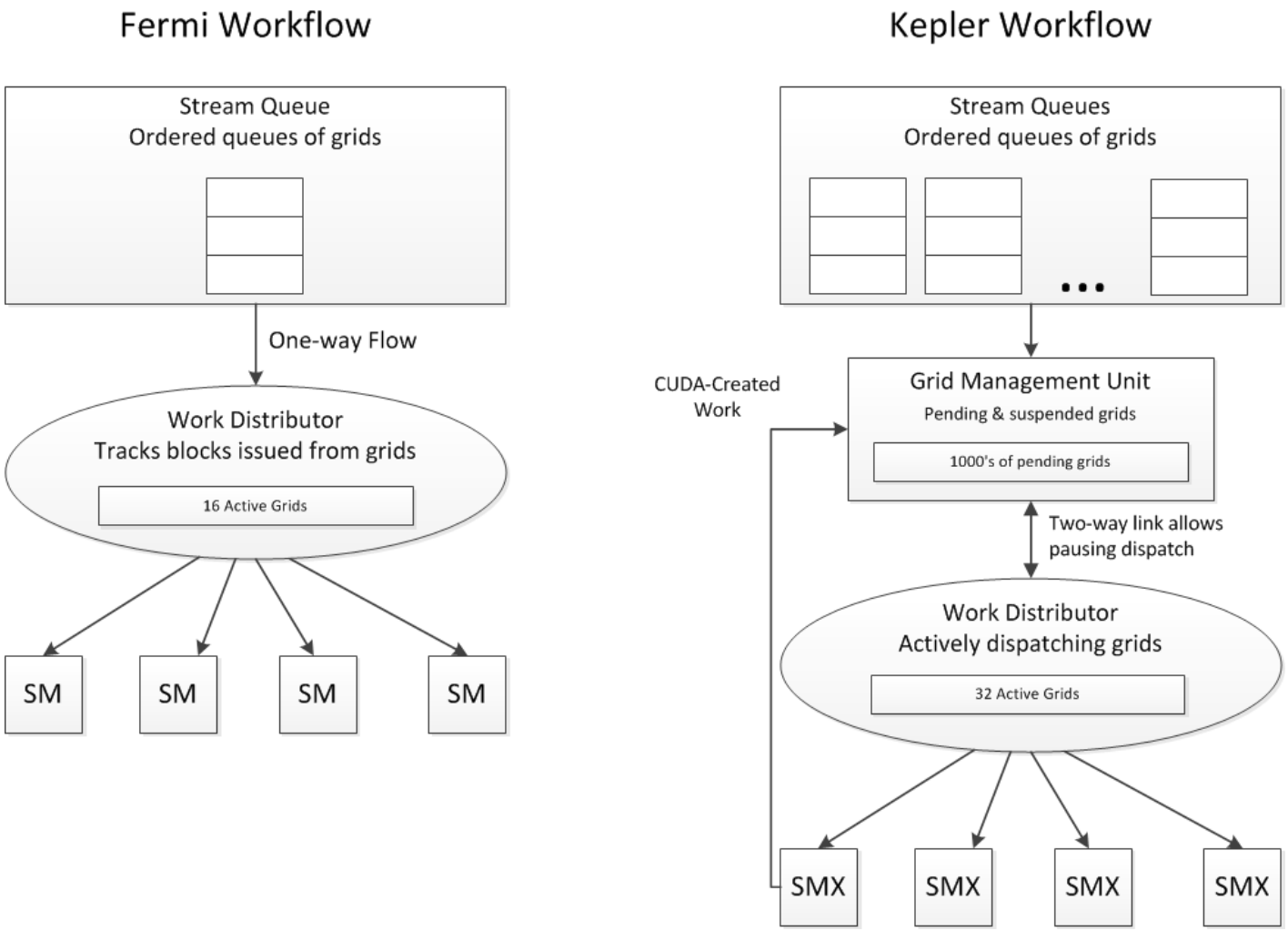
HyperQ


Hyper-Q
CPU Cores Simultaneously Run Tasks on Kepler
FERMI
1 MPI Task at a Time
KEPLER
32 Simultaneous MPI Tasks

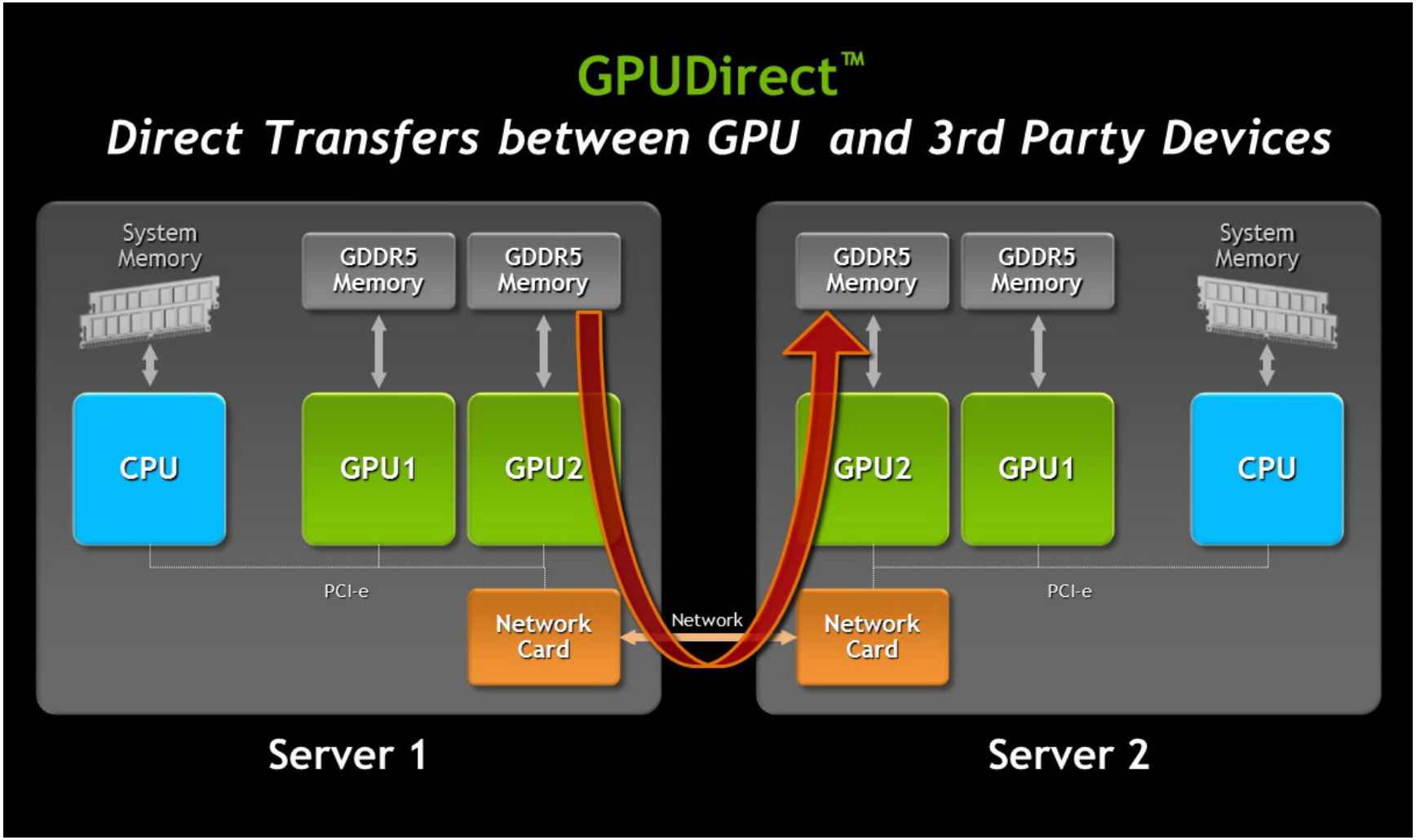
Dynamic parallelism



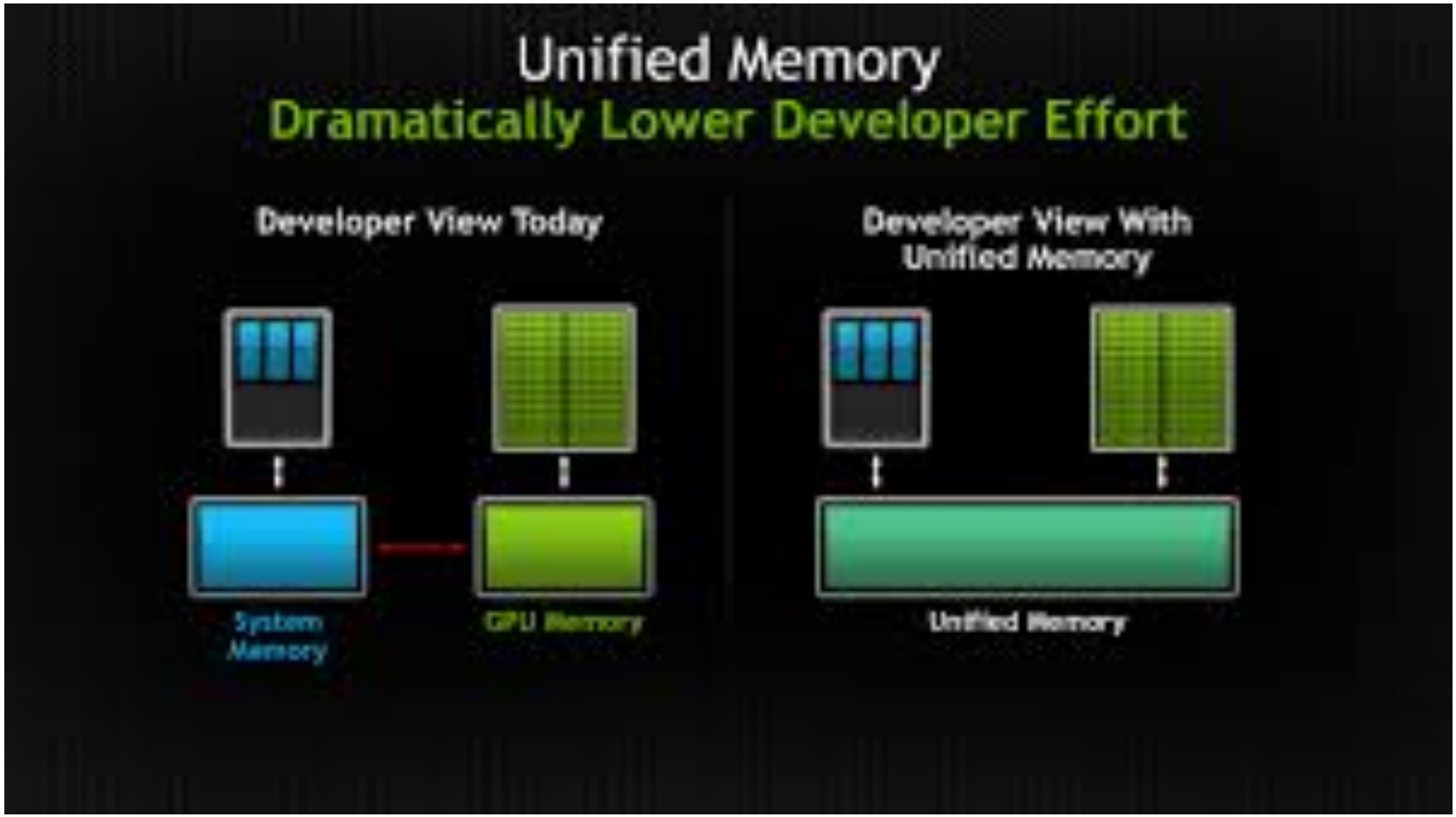
Dynamic parallelism



GPUDirect


GPUDirect™
Direct Transfers between GPU and 3rd Party Devices
System Memory
GDDR5 Memory
GDDR5 Memory
GDDR5 Memory
GDDR5 Memory
System Memory
CPU
GPU1
GPU2
GPU2
GPU1
CPU
PCI-e
PCI-e
Network Card
Network
Network Card
Server 1
Server 2

Unified memory



SUMMARY

Take home message

- GPUs are massively parallel architectures with limited flexibility, but very high throughput
- Pro's:
  - Much higher compute capabilities
  - Higher bandwidth
- Con's
  - Limited on-card memory
  - Low-bandwidth communication with host
- Debate-able
  - Programmability & productivity

Open research questions

- Shall we port all applications on GPUs?
  - If yes – can we automate the process?
  - If not – can we decide how to select?
- Shall we use GPUs in large-scale systems?
- Shall we use heterogeneous CPU+GPU systems?
- Can we improve the GPU design …
  - For HPC?
  - For other application domains?

Questions? Comments? Suggestions?

- A.L.Varbanescu@uva.nl

… also if you want to work on GPU-related projects OR in a team that works on heterogeneous computing.

… All you have to do is ask ☺