RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

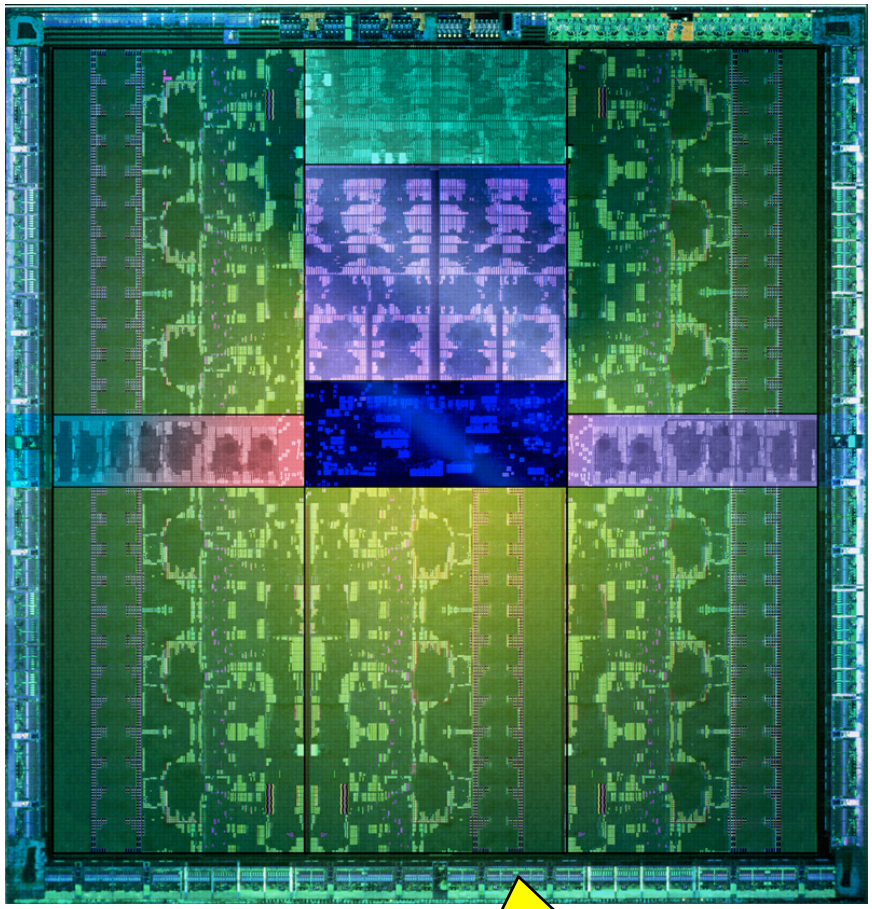
Introduction to High Performance Computing

Lecture 10 – GPU Computing II

Holger Fröning
Institut für Technische Informatik
Universität Heidelberg

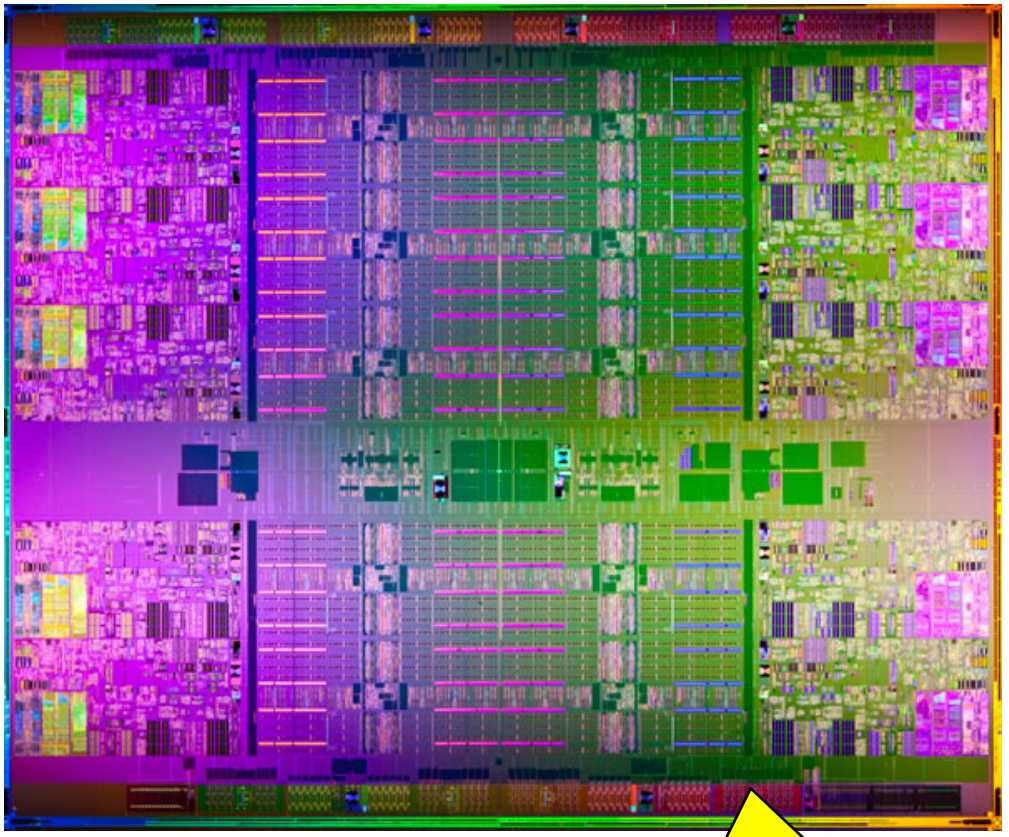
Introduction

GK110



2500 “cores”

XEON-E7



10 “cores”

CUDA Thread Scheduling

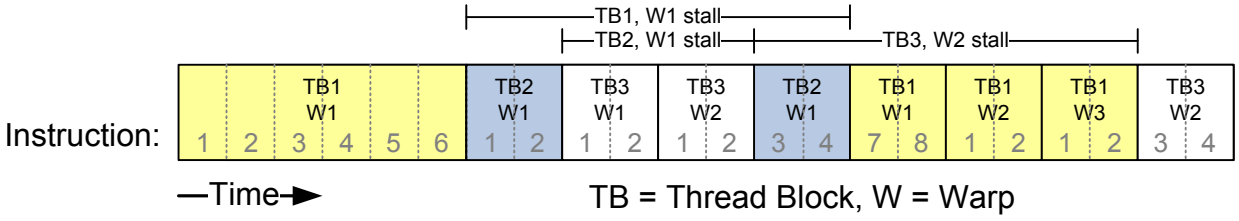
src:flickr.com

Thread Scheduling

- Up to 1k threads per block
  - One block executes on one SM
- Each thread block is divided in 32-thread **warps**
  - Implementation decision, not CUDA
- Warps are the units for the scheduler
- Example:
  - 4 blocks being executed on one SM, each block 1k threads
  - How many warps?
    - Each block here has 32 warps, thus 128 warps in total
- Scheduler: select one out of these 128 warps for instruction fetch and execution

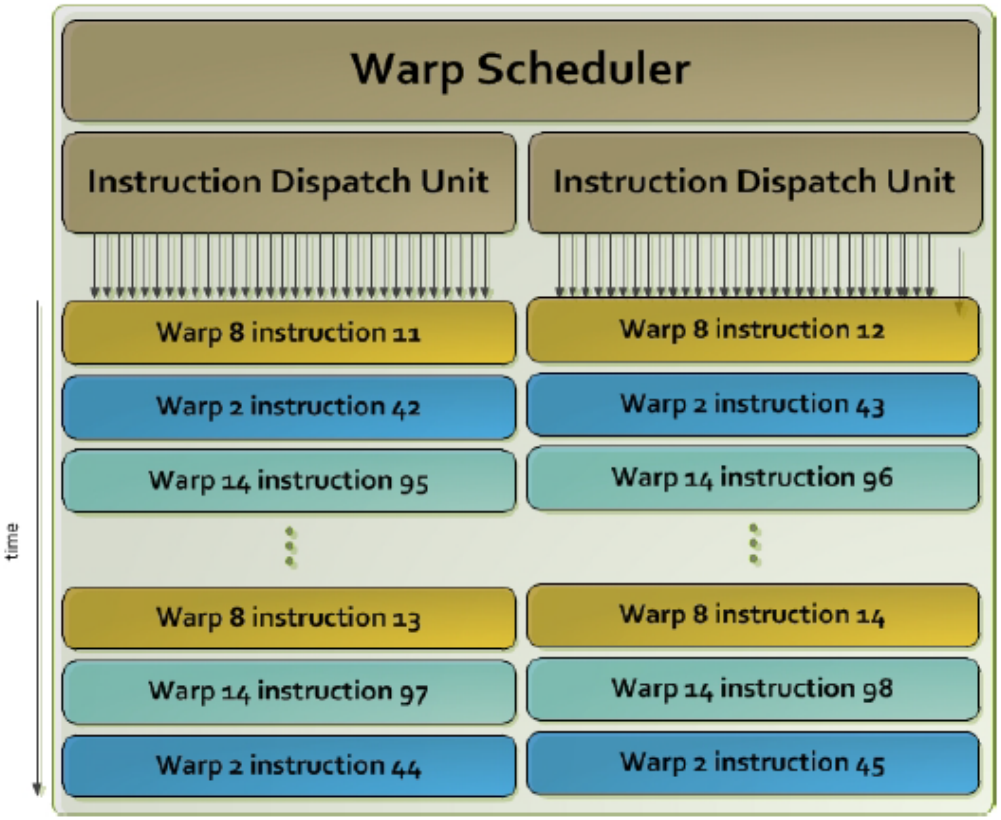
Thread Scheduling

- Fine-grained multi-threading (FGMT)
  - Switch context (i.e., warp) every cycle
  - A warp that has the operands ready for its next instruction is ready for execution
  - All threads in a warp execute the same instruction
- Goal of FGMT: latency hiding
  - Global memory access latency: ~400-600 cycles
  - Sufficient number of warps can keep all functional units busy
- Warp count for maximum utilization depends on computational intensity



Thread Scheduling

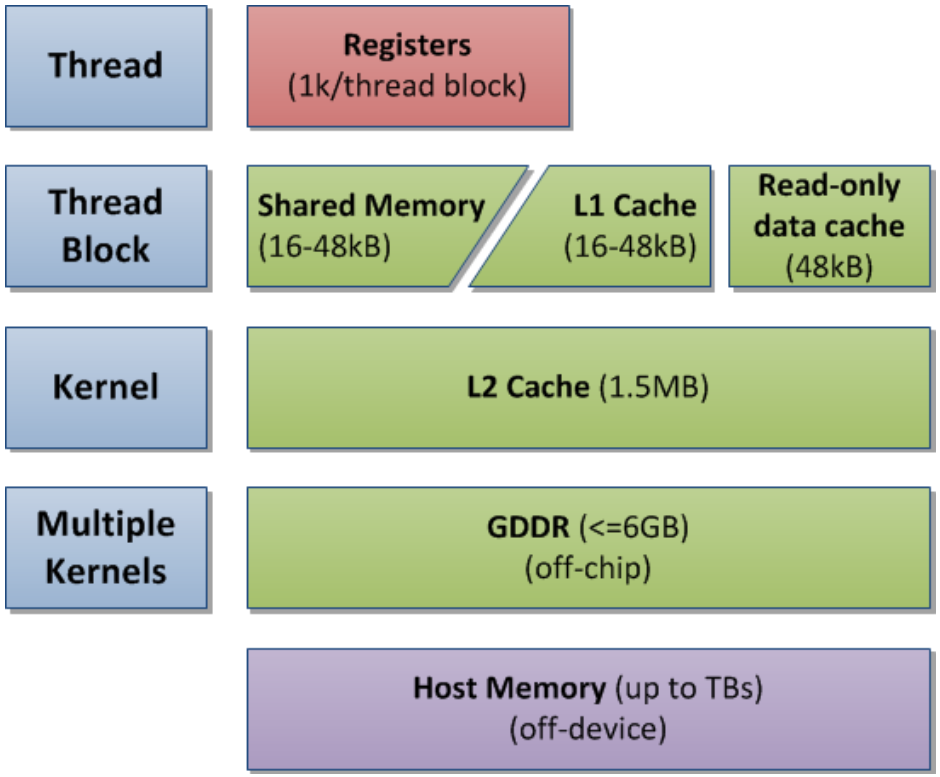
- Fetch one instruction per cycle
  - From I$
- Determine dependencies (operands)
- Scoreboard checks if dependencies are resolved
  - Prevent data hazards
- Issue: select one warp based on prioritized round-robin
  - Priority: warp age
- Scheduler broadcasts the instruction to all 32 threads in a warp
  - Dedicated control paths
  - Branch divergence problem
  - Write-masks



Memory Subsystem

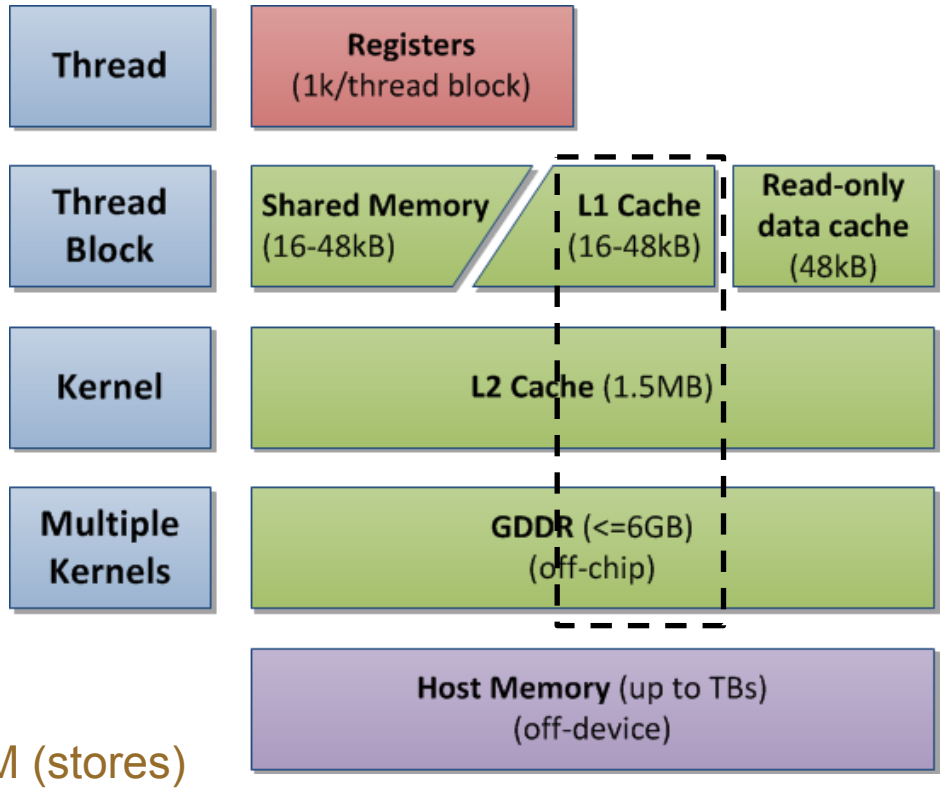
GK110 – Memory Hierarchy

- Registers at thread level
  - Registers/thread depends on run-time configuration
  - Max. 255 registers/thread
- Shared memory / L1$ at block level
  - Variable sizes
  - L1$ can serve for register spilling
  - L1$ not coherent, write-invalidate
  - Compiler controlled RO data$
- L2$ / GDDR at device level
  - GDDR: ~400-600 cycles access latency
  - L2$ as victim cache for all upper units, write-back
  - Purpose: reducing contention



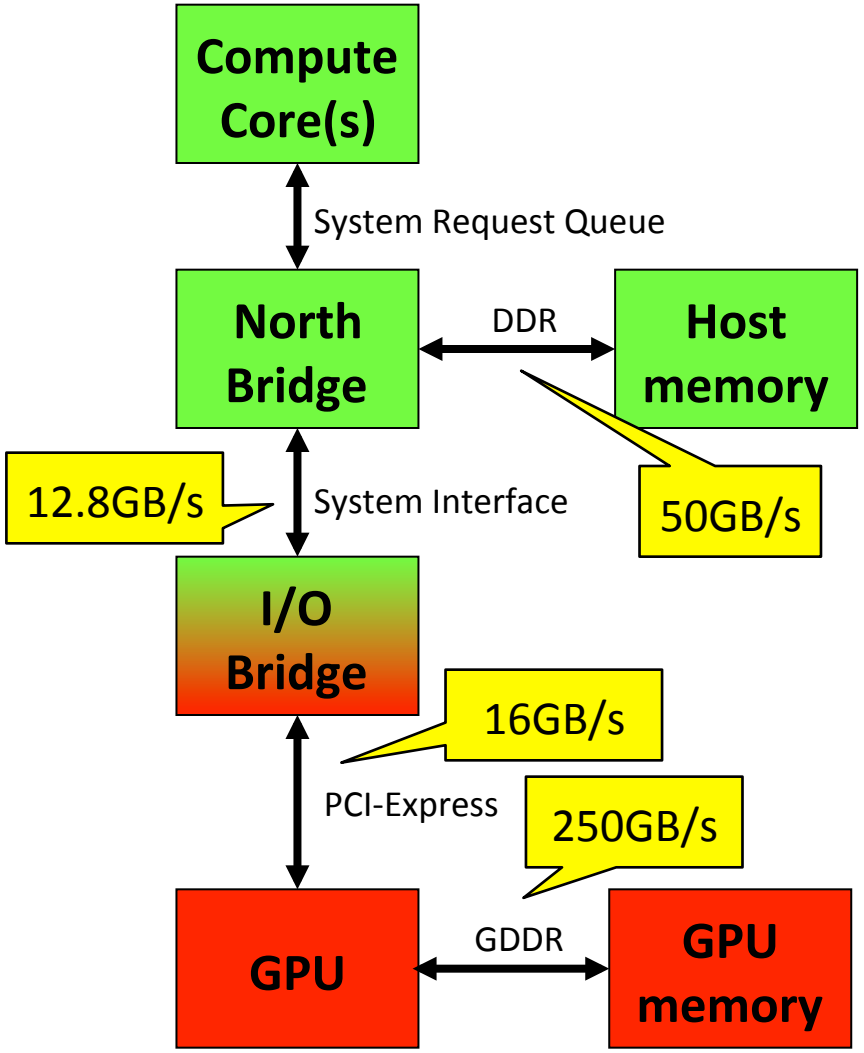
Local Memory

- Local memory: part of global memory, but thread-local
  - Register spilling: when SM runs out of resources
  - Limited register count per thread
  - Limited total number of registers
  - LM is used if the source code exceeds these limits
  - Local because each thread has its private area
- Differences from global memory
  - Stores are cached in L1$
  - Addressing is resolved by compiler
- Store always happens before load
  - Per thread: move data from GM to LM (stores)
  - Subsequent load accesses



Host Memory

- Pinned/unpinned host memory
  - Unpinned host memory: probability of demand paging
  - Pinned host memory: autonomous device access possible
- cudaMemcpy
  - GPU DMA engine(s)
- Zero copy
  - GPU threads (CC >= 2.0)
  - For initial shared memory fills etc.



Global Memory

- High bandwidth, high latency
  - GPU knows how to overcome latency issues
- Coalesced access
  - Avoid access with offset or stride
  - Combine fine-grain accesses by multiple threads into single GDDR operations
  - Coalesced thread access should match a multiple of L1/L2 cache line sizes
    - Cache line sizes: L1: 128B, L2: 32B
- Misaligned accesses: one warp is scheduled, but accesses misaligned addresses
  - GPUs use caches for access coalescing

Global Memory – Access Penalties

- Main problem: thread scheduling does not result in coalesced accesses
- Solution: manually control data movement in memory hierarchy
  - Caches = transparent, implicit hierarchy
  - Scratchpad = opaque, explicit hierarchy
- Collaborative loads from global memory to shared memory
  - Common case: one thread is not moving the data it requires (at least not immediately)
  - Main GPU advantage is memory bandwidth (together with multi-core): **coalescing of upmost importance!**

Matrix Multiply for a CPU (Reference)

Matrix Multiply – CPU naive

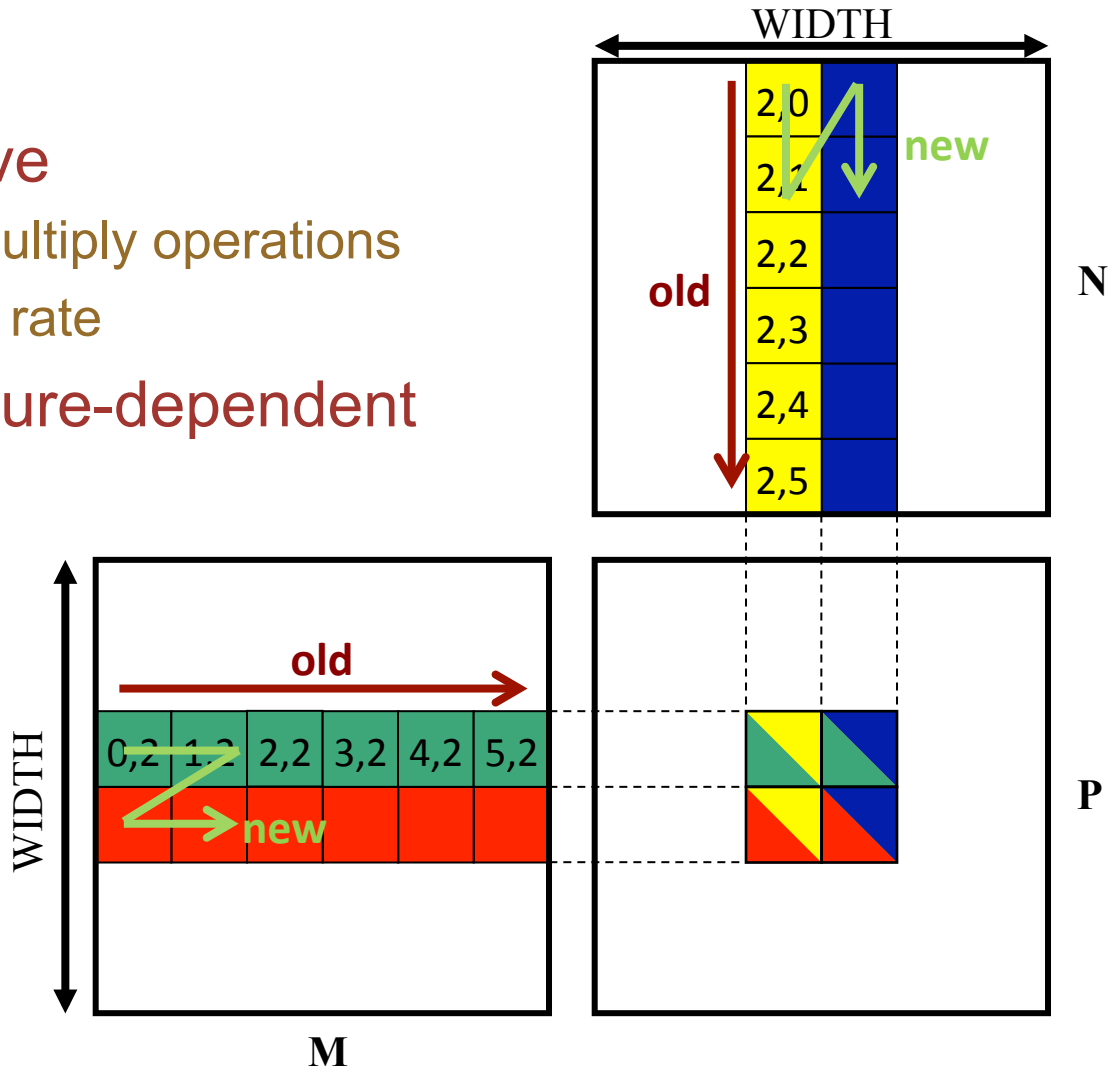
- CPU sequential version
  - No big surprises
  - Can be called directly

```
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
  for (int i = 0; i < Width; ++i) {
    for (int j = 0; j < Width; ++j) {
      float sum = 0;
      for (int k = 0; k < Width; ++k) {
        float a = M[i * width + k];
        float b = N[k * width + j];
        sum += a * b;
      }
      P[i * Width + j] = sum;
    }
  }
}
```

Matrix Multiply – CPU blocked

- Multiply is commutative
  - So feel free to reorder multiply operations
  - Goal: increase cache hit rate
- Block size is architecture-dependent parameter
  - Cache size

$$P_{i,j} = \sum_k M_{i,k} \times N_{k,j}$$



Matrix Multiply – CPU blocked

```
void MatrixMulOnHostBlocked ( float* M, float* N, float* P, long matWidth,
long blockSize )
{
  // Matrix multiply of P = M * N
  // assume P to be initialized to zero
  // assume matrices to be square ones

  float temp;
  for ( long ii = 0; ii < matWidth; ii += blockSize ) {
    for ( long jj = 0; jj < matWidth; jj += blockSize ) {
      for ( long kk = 0; kk < matWidth; kk += blockSize ) {
        for ( long i = ii; i < findMin ( ii+blockSize, matWidth ); i++) {
          for ( long j = jj; j < findMin ( jj+blockSize, matWidth ); j++ ) {
            temp = 0;
            for ( long k = kk; k < findMin ( kk+blockSize, matWidth ); k++ )
              temp += M[i * matWidth + k] * N[k * matWidth + j];
            P[ i * matWidth + j] += temp;
          }
        }
      }
    }
  }
}
```

Matrix Multiply – CPU blocked

Trace for naive implementation

```
..
<snip>
..
P[2][3] += M[3][0] * N[0][2]
P[2][3] += M[3][1] * N[1][2]
P[2][3] += M[3][2] * N[2][2]
P[2][3] += M[3][3] * N[3][2]
..
P[3][3] += M[3][0] * N[0][3]
P[3][3] += M[3][1] * N[1][3]
P[3][3] += M[3][2] * N[2][3]
P[3][3] += M[3][3] * N[3][3]
```

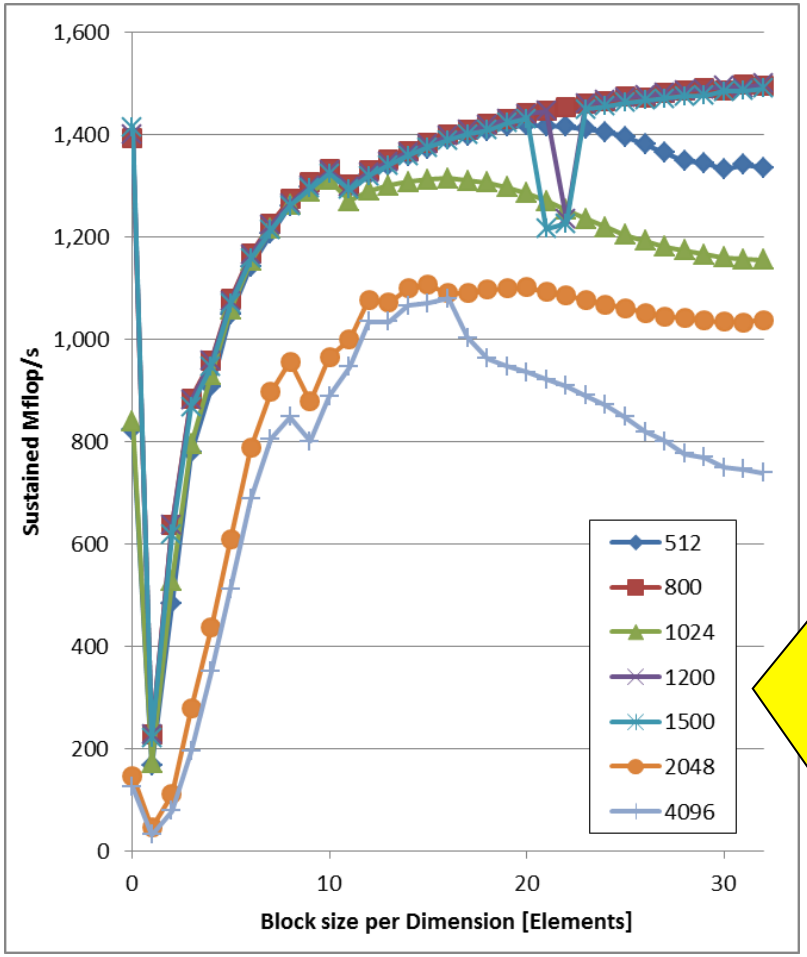
No locality - RED
Where is spatial locality? - GREEN
Where is temporal locality? - BLUE

Trace for blocks of two-by-two

```
..
<snip>
..
P[2][3] += M[3][0] * N[0][2]
P[2][3] += M[3][1] * N[1][2]
P[3][3] += M[3][0] * N[0][3]
P[3][3] += M[3][1] * N[1][3]
..
<snip>
..
P[2][3] += M[3][2] * N[2][2]
P[2][3] += M[3][3] * N[3][2]
P[3][3] += M[3][2] * N[2][3]
P[3][3] += M[3][3] * N[3][3]
```

Matrix Multiply – CPU blocked

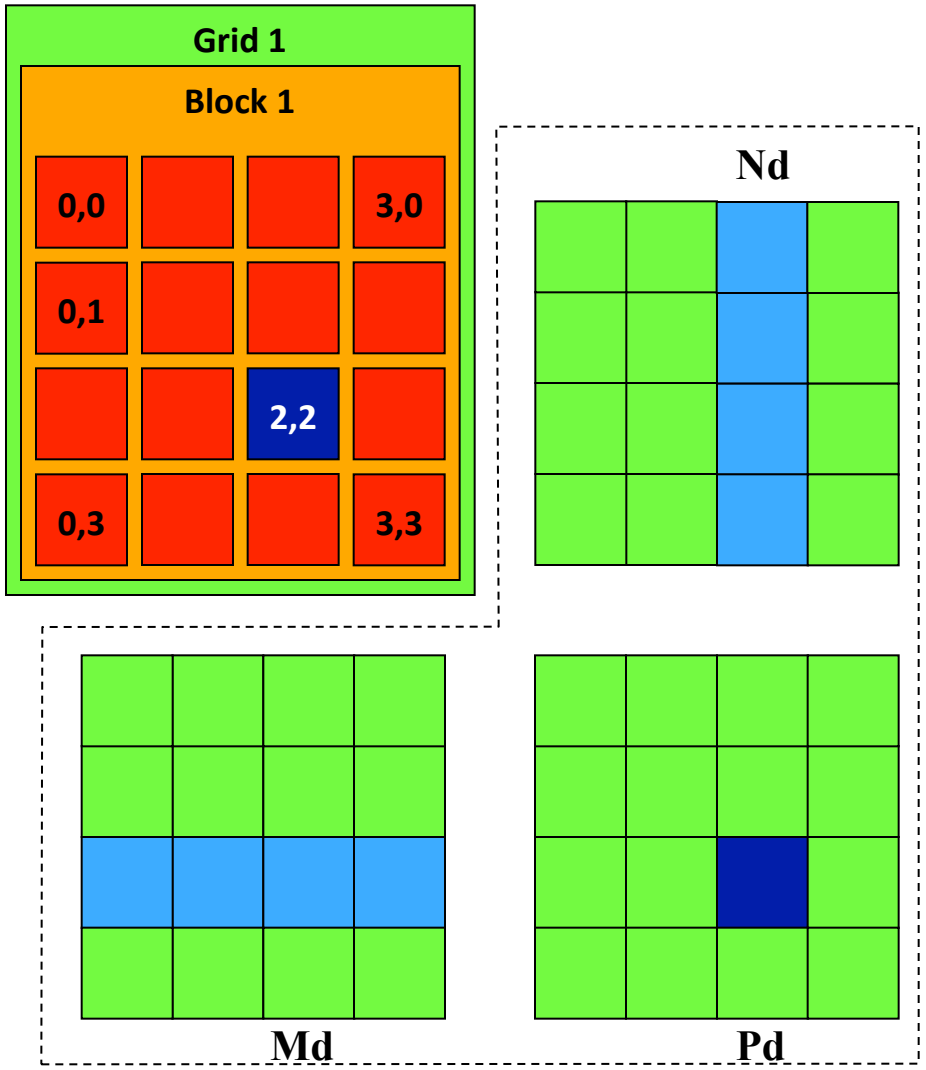
- Performance for single-threaded CPU run
  - Xeon E5 Sandy Bridge
  - 4 cores @ 2.4GHz
- Single precision
- Varying matrix sizes [elements per dimension]
- Block size 0 = non-blocked (reference)
- Huge drop for block size of 1?
  - Control flow overhead
- Non-blocked better than blocked?
  - Cache size!
- Factor of 10x for blocked vs. non-blocked typical



Optimizing Matrix Multiply for a GPU

Matrix Multiply – Quick analysis

- A single thread block computes Pd
  - Each thread computes a single element of Pd
    - Load a row of Md
    - Load a column of Nd
    - Per element: one mult., one add
    - Write Pd
- **Issue 1:** Matrix size limited by threads/block
- **Issue 2:** Compute:Memory ratio
  - Computational intensity
  - ~ 2*WIDTH:2*WIDTH or 1:1 (very low)
    - Or 1 FLOP/4B



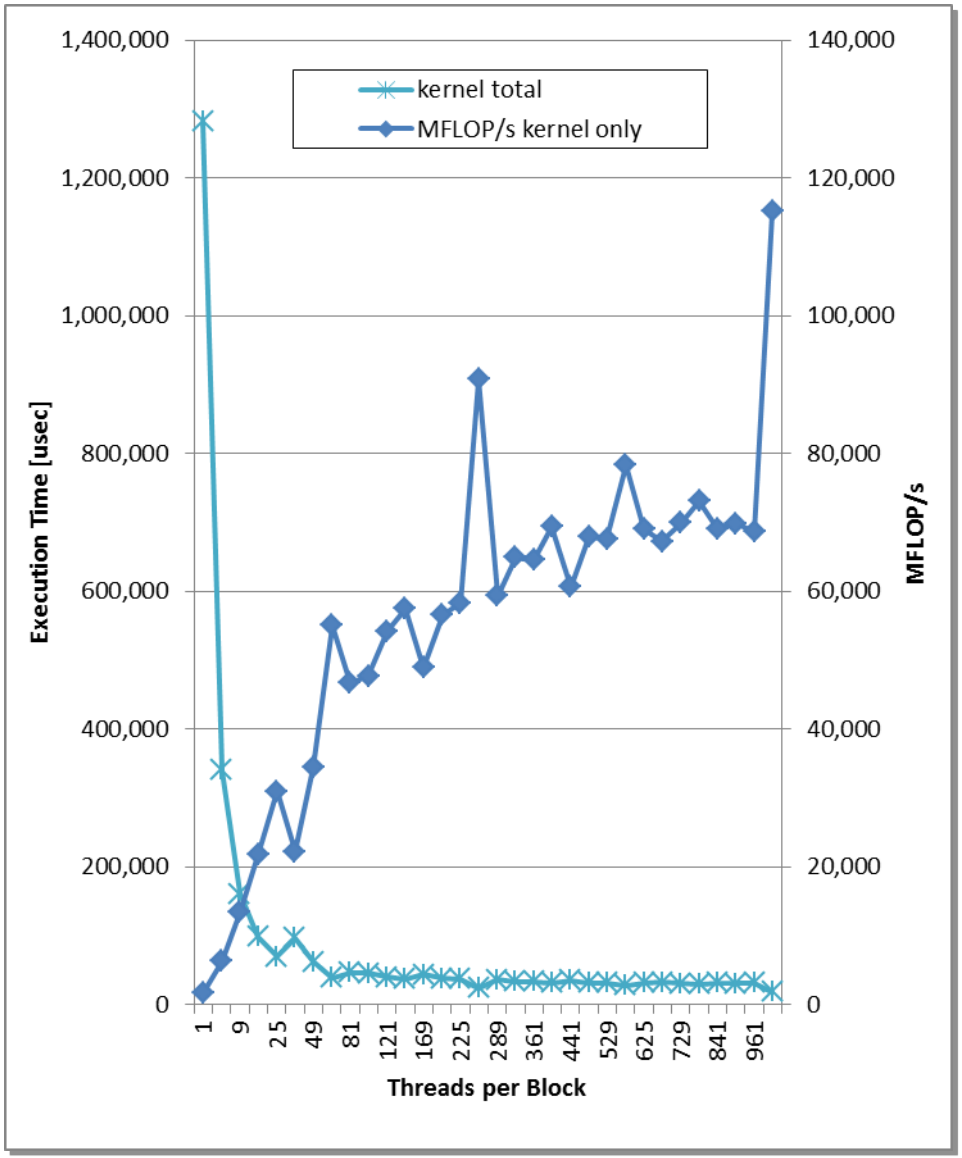
Matrix Multiply – Multiple Thread Blocks

- Kernel using multiple blocks

```
__global__ void MatrixMulKernel ( float* Md, float* Nd, float* Pd, int Width )
{
  // Calculate the row index of the Pd element and M
  int Row = blockIdx.y * blockDim.y + threadIdx.y;
  // Calculate the column index of Pd and N
  int Col = blockIdx.x * blockDim.x + threadIdx.x;

  float Pvalue = 0;
  // each thread computes one element of the block sub-matrix
  for ( int k = 0; k < Width; ++k )
    Pvalue += Md [ Row * Width + k ] * Nd [ k * Width + Col ];

  Pd [ Row * Width + Col ] = Pvalue;
}
```

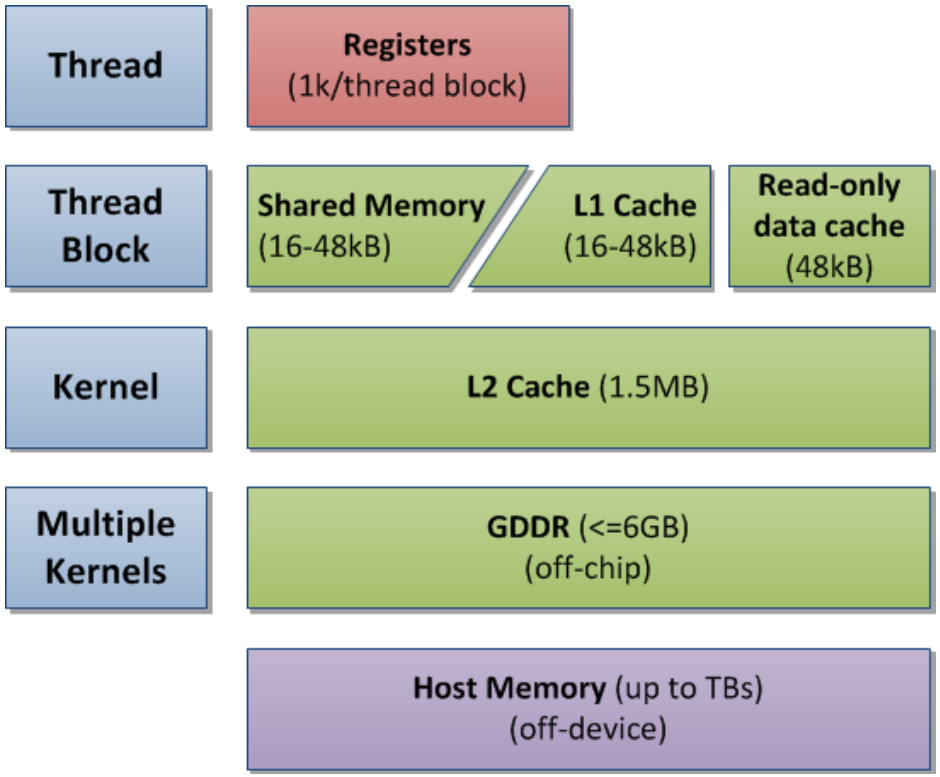
Matrix Multiply – Single Precision Results

- Tesla K20c GPU, Kepler-class
- Scheduling: varying the number of threads per block
  - 1k x 1k matrix size
  - Match block count
  - Nice example for performance increase of large thread counts
    - Not always optimal
- Resulting GFLOP/s
  - N^2=1M elements, each 2N FLOP = 2.14 GFLOP
  - Without data movement



GK110 – Memory Hierarchy

- Shared memory is (yet) not used
  - Unlike caches for CPUs, it is manually controlled
  - I.e., the programmer is responsible to move data from/to shared memory
- If it‘s used, same access costs as any cache

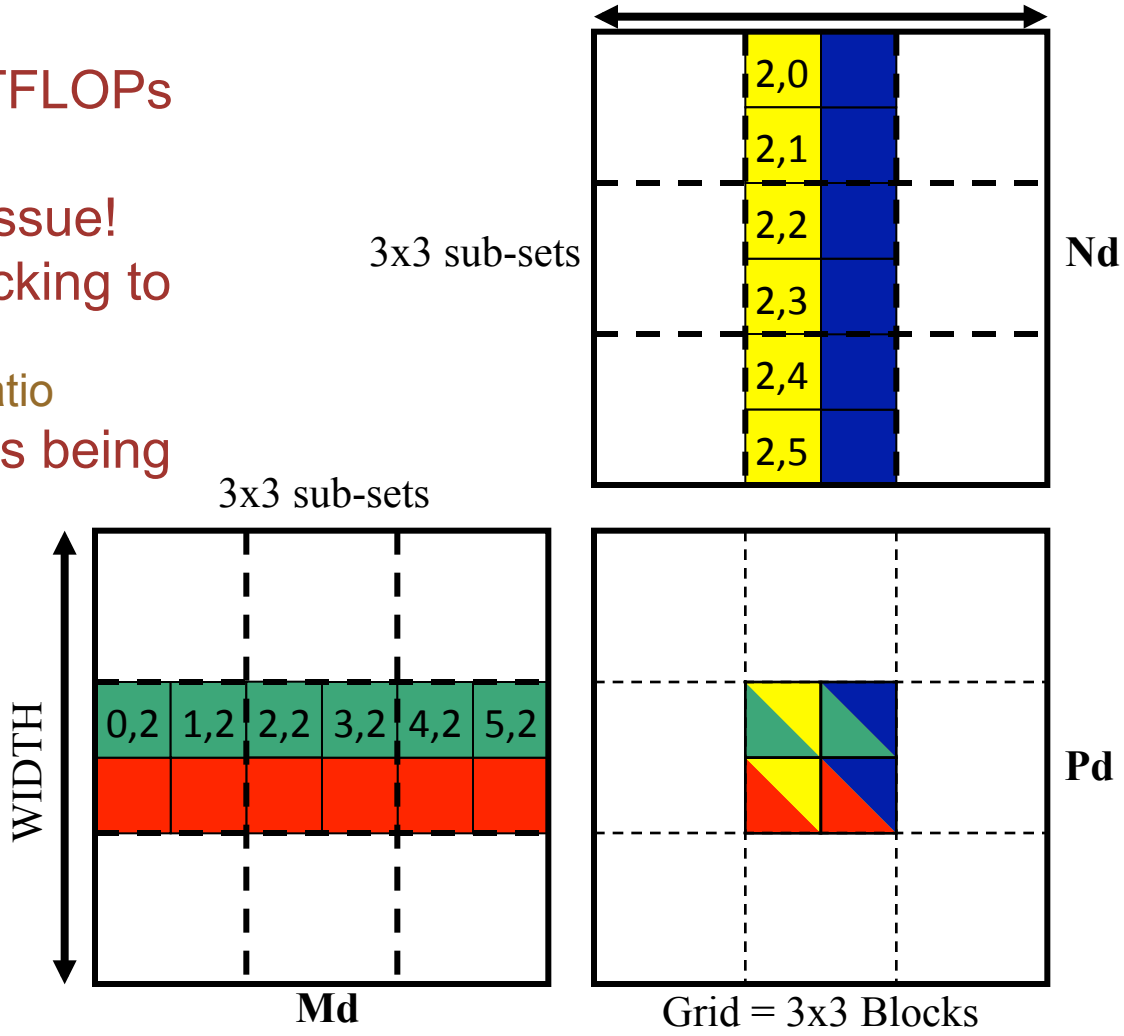


Matrix Multiply – Shared Memory

- ~140GFLOPs out of 3.5TFLOPs

➔Optimizations required!

- Compute intensity is an issue!
- Similar to CPUs, use blocking to increase data reuse
  - And thus the Flop-to-byte ratio
- Old: each input element is being read by WIDTH threads
- New: is read by one thread, but used by multiple threads
- Separate kernel execution into phases
  - Size of a sub-set should match a tile size



Matrix Multiply – Shared Memory

Time →

	Phase 1			Phase 2		
$T_{0,0}$	$\mathbf{Md_{0,0}}$ ↓ $Mds_{0,0}$	$\mathbf{Nd_{0,0}}$ ↓ $Nds_{0,0}$	$PdValue_{0,0}$ += $Mds_{0,0}*Nds_{0,0}$ + $Mds_{1,0}*Nds_{0,1}$	$\mathbf{Md_{2,0}}$ ↓ $Mds_{0,0}$	$\mathbf{Nd_{0,2}}$ ↓ $Nds_{0,0}$	$PdValue_{0,0}$ += $Mds_{0,0}*Nds_{0,0}$ + $Mds_{1,0}*Nds_{0,1}$
$T_{1,0}$	$\mathbf{Md_{1,0}}$ ↓ $Mds_{1,0}$	$\mathbf{Nd_{1,0}}$ ↓ $Nds_{1,0}$	$PdValue_{1,0}$ += $Mds_{0,0}*Nds_{1,0}$ + $Mds_{1,0}*Nds_{1,1}$	$\mathbf{Md_{3,0}}$ ↓ $Mds_{1,0}$	$\mathbf{Nd_{1,2}}$ ↓ $Nds_{1,0}$	$PdValue_{1,0}$ += $Mds_{0,0}*Nds_{1,0}$ + $Mds_{1,0}*Nds_{1,1}$
$T_{0,1}$	$\mathbf{Md_{0,1}}$ ↓ $Mds_{0,1}$	$\mathbf{Nd_{0,1}}$ ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}*Nds_{0,0}$ + $Mds_{1,1}*Nds_{0,1}$	$\mathbf{Md_{2,1}}$ ↓ $Mds_{0,1}$	$\mathbf{Nd_{0,3}}$ ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}*Nds_{0,0}$ + $Mds_{1,1}*Nds_{0,1}$
$T_{1,1}$	$\mathbf{Md_{1,1}}$ ↓ $Mds_{1,1}$	$\mathbf{Nd_{1,1}}$ ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}*Nds_{1,0}$ + $Mds_{1,1}*Nds_{1,1}$	$\mathbf{Md_{3,1}}$ ↓ $Mds_{1,1}$	$\mathbf{Nd_{1,3}}$ ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}*Nds_{1,0}$ + $Mds_{1,1}*Nds_{1,1}$

Per thread: TILE_WIDTH MADDs

Phase change with offset = TILE_WIDTH

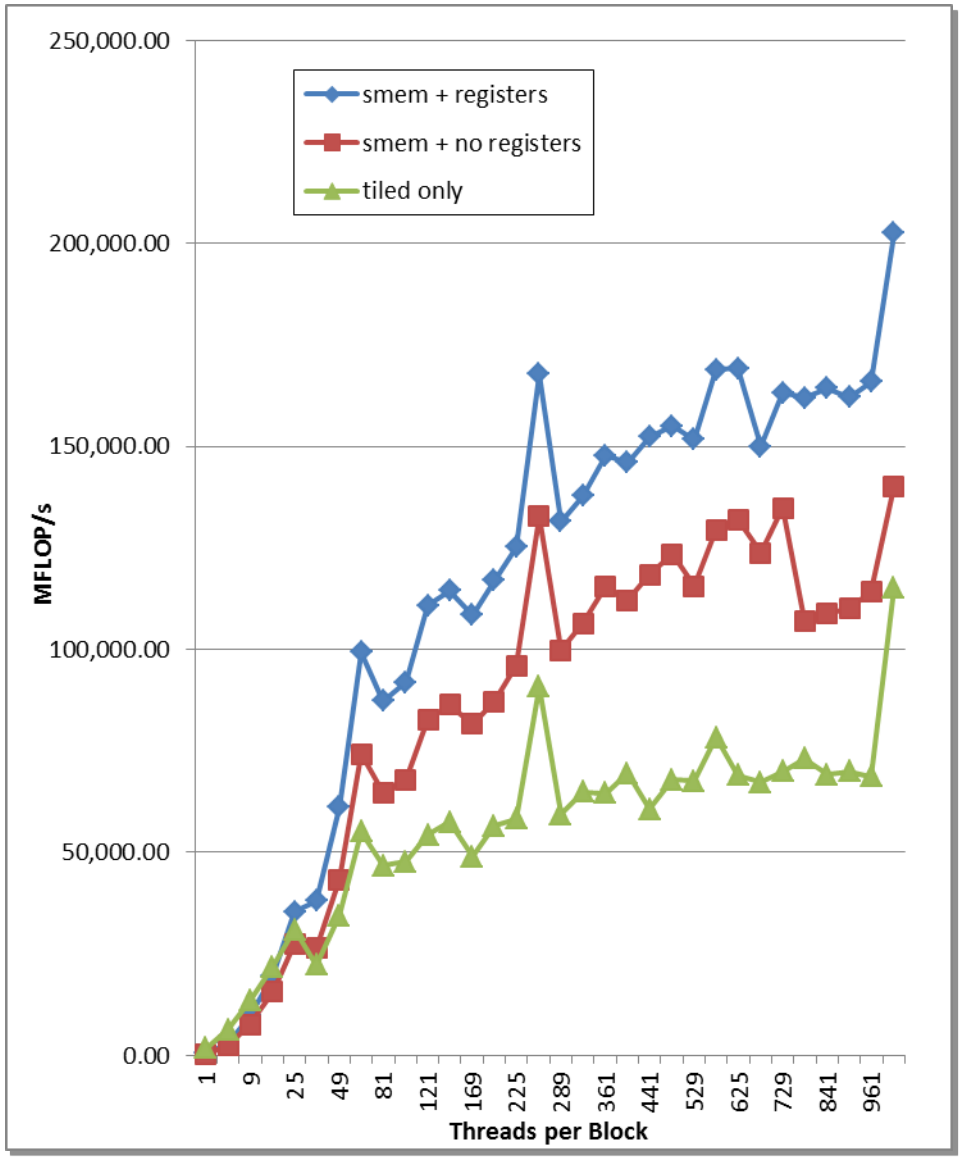
Matrix Multiply – Shared Memory

```
__global__ void MM_SM ( float* Md, float* Nd, float* Pd, int Width )
{
  __shared__ float Mds [TILEWIDTH] [TILEWIDTH];
  __shared__ float Nds [TILEWIDTH] [TILEWIDTH];
  int bx = blockIdx.x;  int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;

  // Identify the row and column of the Pd element to work on
  int Row = by * TILEWIDTH + ty;
  int Col = bx * TILEWIDTH + tx;
  float Pvalue = 0;

  if !(Row > Width || Col > Width) {
  // Loop over the Md and Nd tiles required to compute the Pd element
    for ( int m = 0; m < Width / TILEWIDTH; ++m ) {
      // Collaborative loading of Md and Nd tiles into shared memory
      Mds [ty] [tx] = Md [ Row * Width + ( m * TILEWIDTH + tx ) ];
      Nds [ty] [tx] = Nd [ Col + ( m * TILEWIDTH + ty ) * Width ];
      __syncthreads();

      for ( int k = 0; k < TILEWIDTH; ++k )
        Pvalue += Mds[ty][k] * Nds[k][tx];
      __syncthreads ();
    }
    Pd[Row * Width + Col] = Pvalue;
  }
}
```

Synchronization needed due to dependencies

Matrix Multiply – Shared Memory Results

- Performance comparison for Tesla K20c:
  - Tiled only
  - Use of shared memory, but no register for intermediate value
  - Use of both shared memory and register
- Block size <= 1k
  - Both code optimizations and configuration matters!



Matrix Multiply – Shared Memory Dependencies

```
<snip>
    for ( int m = 0; m < Width / TILEWIDTH; ++m ) {
      // Collaborative loading of Md and Nd tiles into shared memory
      Mds [ty] [tx] = Md [ Row * Width + ( m * TILEWIDTH + tx ) ];
      Nds [ty] [tx] = Nd [ Col + ( m * TILEWIDTH + ty ) * Width ];
1     __syncthreads();

      for ( int k = 0; k < TILEWIDTH; ++k )
        Pvalue += Mds[ty][k] * Nds[k][tx];
2     __syncthreads ();
    }
<snip>
```

- Three types of dependencies
  - RAW: true or data dependency
    - True dependency, so not solvable, see (1)
  - WAR: anti dependency
    - Is a name dependency, so can be solved using renaming, see (2)
  - WAW: output dependency
    - Is a name dependency, so can be solved using renaming, n.a. here

Matrix Multiply – Shared Memory Allocations

```
__global__ void MM_SM ( float* Md, float* Nd, float* Pd, int Width )
{
  __shared__ float Mds [TILEWIDTH] [TILEWIDTH];
  __shared__ float Nds [TILEWIDTH] [TILEWIDTH];
  …
}
```

```
__global__ void MM_SM ( float* Md, float* Nd, float* Pd, int Width )
{
  extern __shared__ float mem_ds [];
  float *Mds = & ( mem_ds [0] );
  float *Nds = & ( mem_ds [size_of_Mds] );

  …
}

int main ()
{
  …
  MM_SM <<< dimGrid, dimBlock, sharedSize >>> ( Md, Nd, Pd, matWidth );
  …
}
```

Manual memory management

Declare total shared memory

Summary

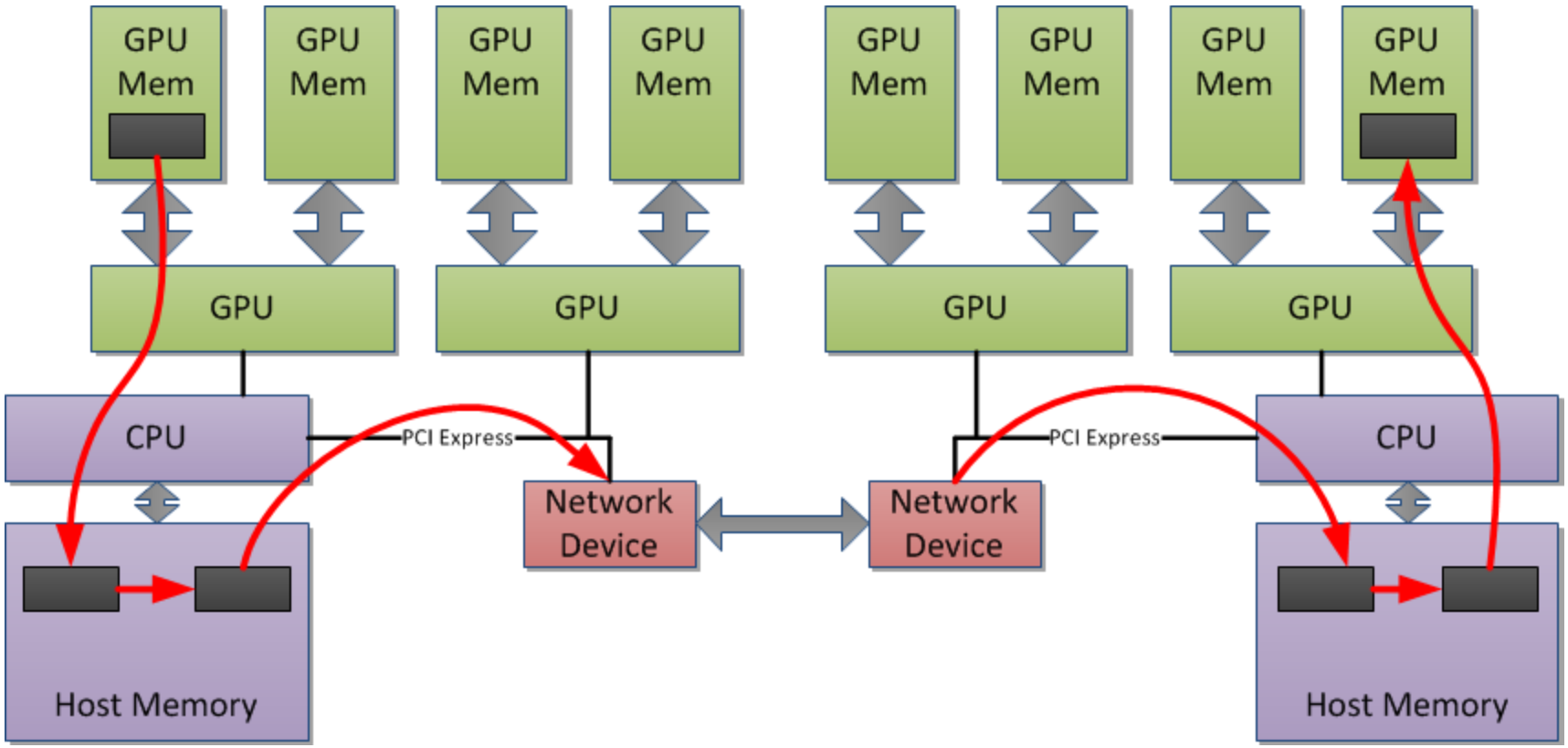
- Matrix multiply as a good example to leverage locality using the shared memory (scratch pad)
- Mind the synchronization within the thread block
  - Dependencies = race conditions
  - Threads are scheduled in warps, threads per warp might not match the scratchpad use model
- Scratch pad about 10x faster than global memory in terms of bandwidth
  - Leverage that for data reuse!
  - Collective memory access, so mind dependencies!
    - Usually one thread will fetch data for other threads to maximize coalescing
- Further candidates for matrix multiply optimizations
  - Improving coalescing, reducing possible bank conflicts

GPUs in MPI Environments

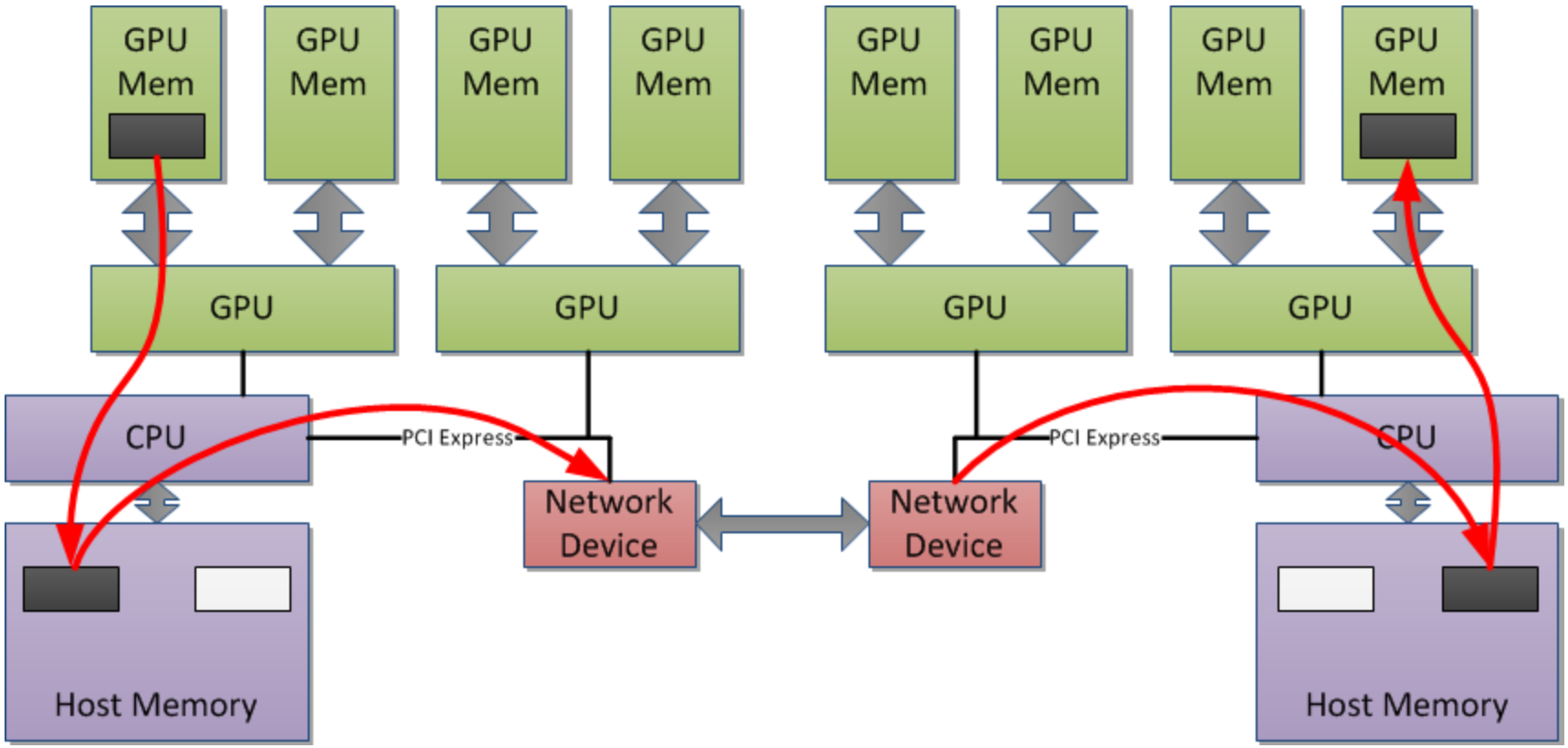
Communication Among Distributed GPUs

- GPUs are the „working horse“ of accelerated HPC clusters
  - Only operate on special on-device memory (global, shared, ...)
  - Question is how to move data efficiently between distributed special memory
  - Control flow for communication is a separate issue

First Approach: 4 copies

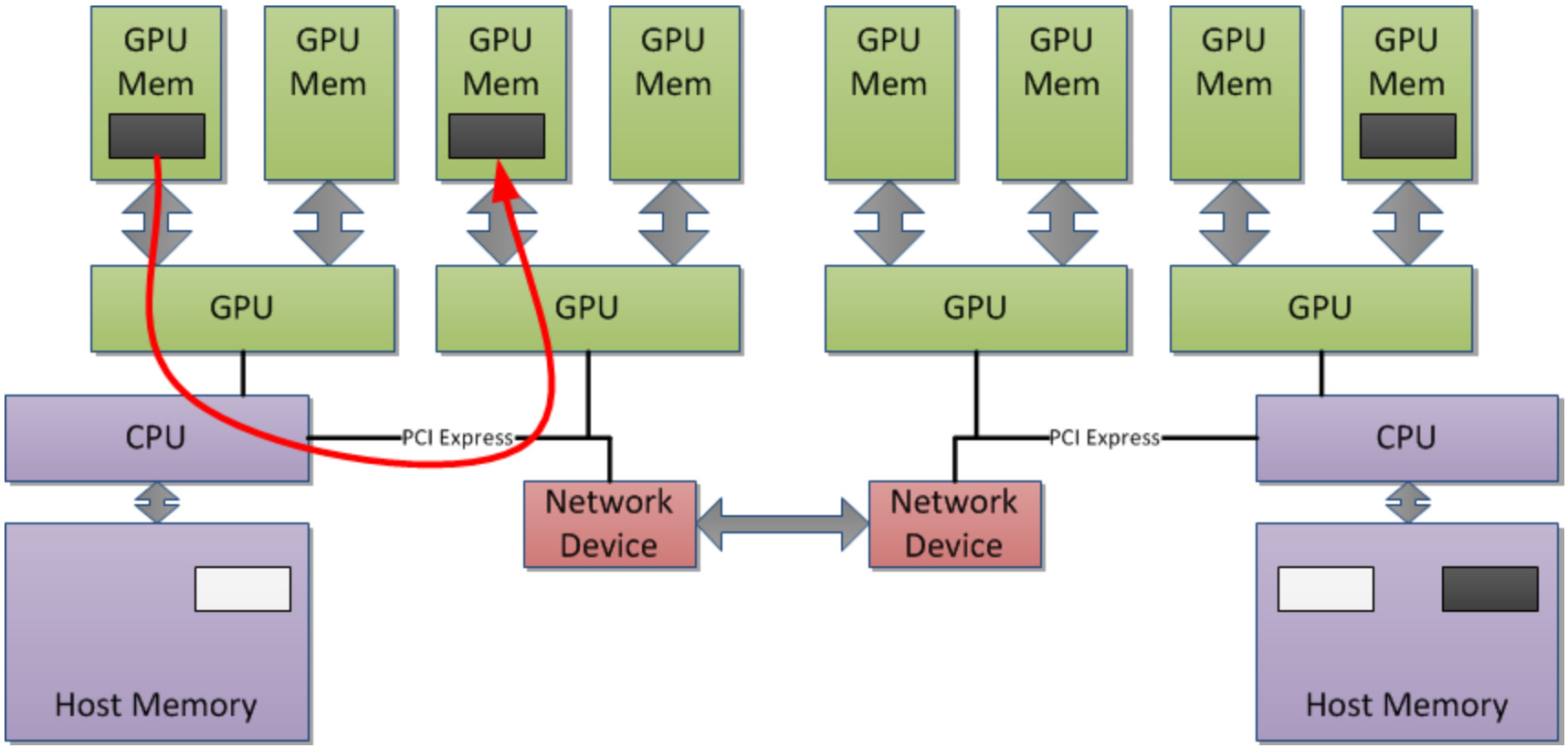


Second Approach: GPUDirect 1.0



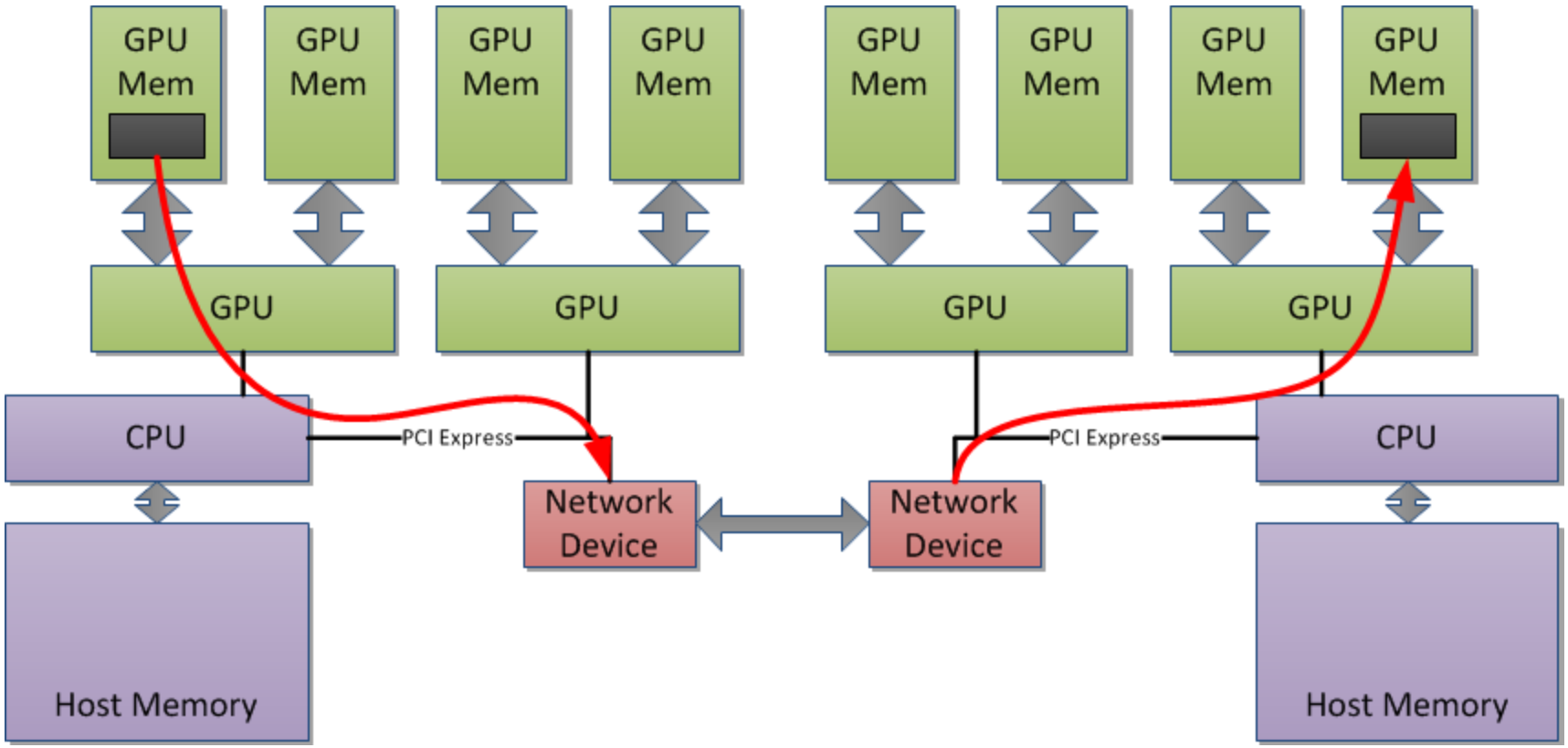
- GPU and network device can read/write the same pinned host memory regions
  - Introduced by NVidia in 2010, 2 copies: Up to 33% speed-ups reported
    - G. Shainer et al., "The development of Mellanox NVIDIA GPUDirect over Infiniband - a new model for GPU to GPU communications", *Comput. Sci. Res. Dev. (2011) 26*:267-273.

Note on GPUDirect 2.0



- Inter-GPU communication, but intra-node (peer-to-peer)
- Zero-copy scheme

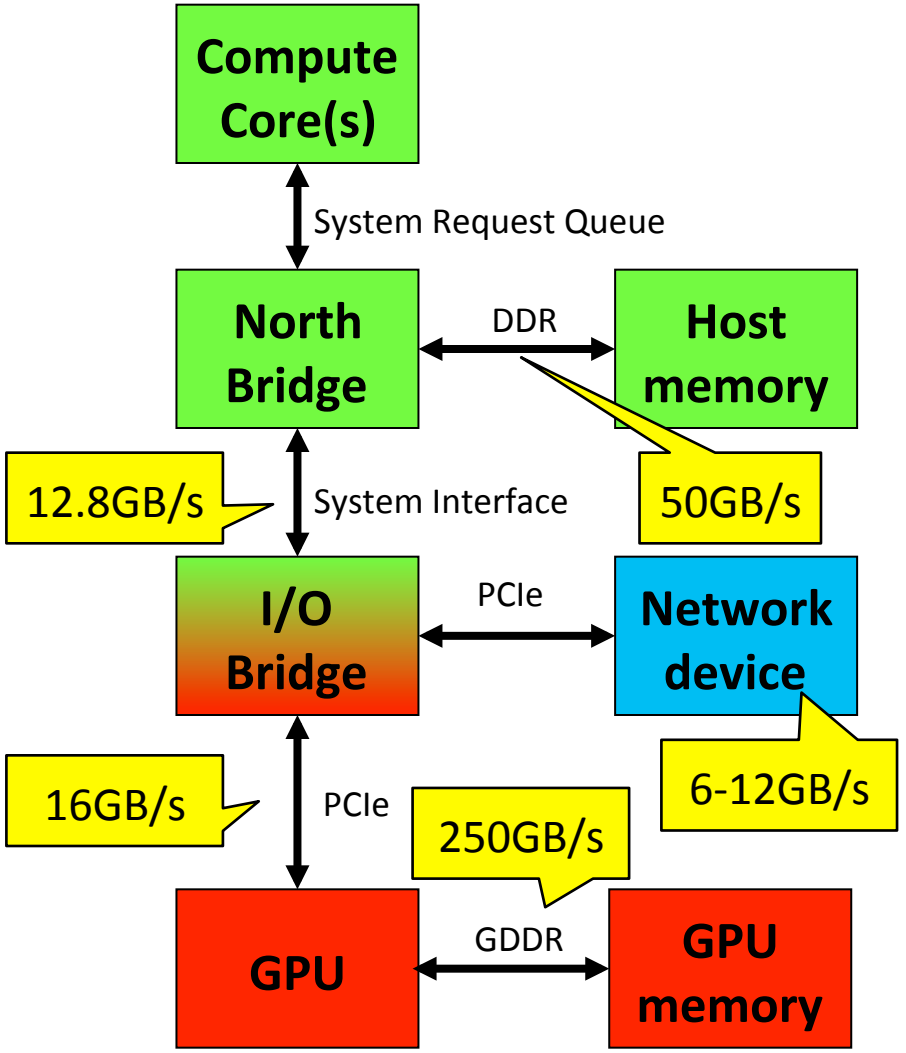
Third Approach: GPUDirect RDMA



- Zero-copy scheme for inter-node GPU communication
- Figure only depicts data movement, not control paths
- Most network devices require host CPU supervision

Outlook

- CPU as offload engine for communication purposes
- Requires to return control flow to the CPUs
  - Associated overhead acceptable for bulk transfers, but not for fine grain communication
- Another possibility:
  - GPU Global Address Spaces (GGAS) & GPU-initiated Put/Get commands



Summary

- GPUs have manually-controlled memory hierarchies (the GPU world is flat)
  - Caches in GPUs not used to reduce latency, but to reduce memory contention and to coalesce accesses
  - Matrix Multiply Example: optimization of similar complexity, but manual data movement for shared memory
- Scheduling
  - Hierarchy: thread block & thread warps
  - Instruction stream == thread warp, != single thread (as for CPUs)
- Latency hiding
  - Hide GM latencies using TLP
  - Also ILP!
    - Less threads ➔ more registers per thread!
- Inter-GPU communication
  - Still some way to go