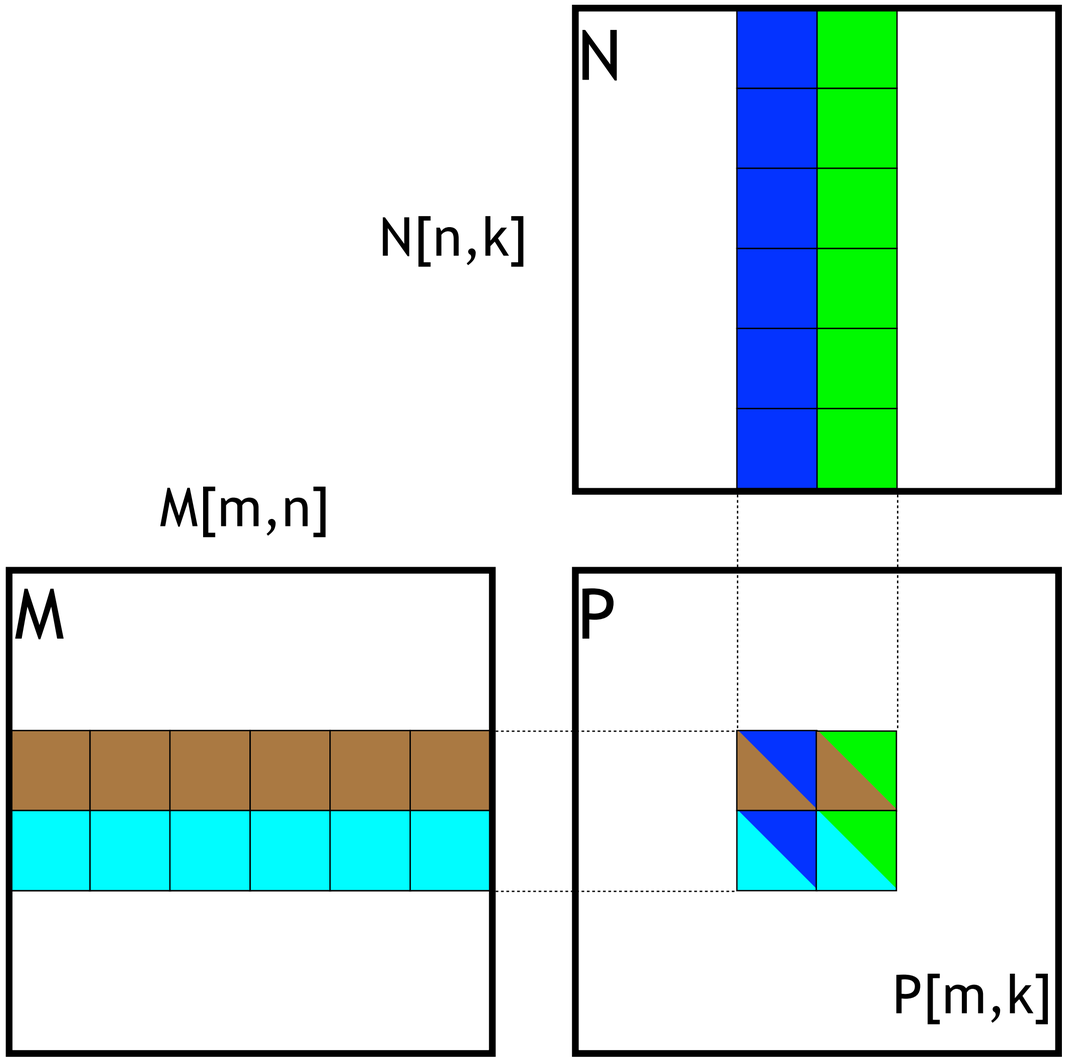
GPU COMPUTING
LECTURE 04 - SHARED MEMORY OPTIMIZATIONS

Kazem Shekofteh
Kazem.shekofteh@ziti.uni-heidelberg.de
Institute of Computer Engineering
Ruprecht-Karls University of Heidelberg
Inspired from lectures by Holger Fröning

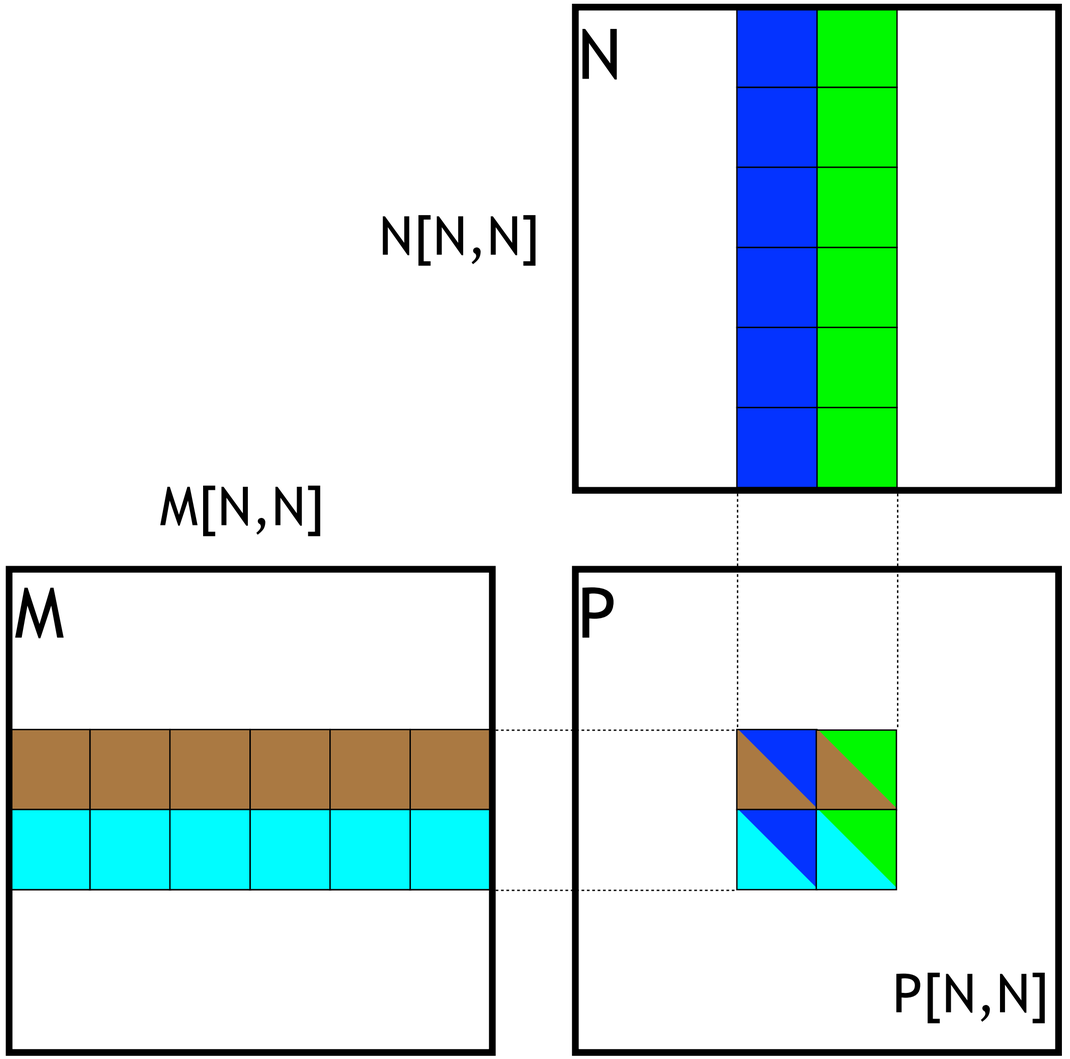
SEQUENTIAL NAIVE VERSION

- Why always Matrix Multiply?
  - Often used
  - Heavily optimized
  - Interesting access patterns
  - Good mixture of sufficient complexity but still simple enough for a comprehensive understanding
  - Finally, it's an important operation!
- Used in many applications as computational kernel
  - In particular for sparse matrix operations
- Here: for dense matrices
  - Experiments and learning
  - High sustained/peak ratio
  - Test system/compiler/OS
- Note on notation
  - M[row,column] = M[row][column]
  - Analogous to C



ANALYSIS

- Assumptions
  - Assume square matrices
  - Assume perfect write-through cache (no issues with conflict or capacity)
- Number of flops: $f = 2*N^3$
  - N^2 elements in C, each N steps, each step: multiply & add
- Number of unique memory accesses: $m_{unique} = 3*N^2$
  - Assuming perfect caching
  - Load from A,B,C, store to C
  - Counting all accesses (RW): $m_{all} = 4*N^3$
- Computational intensity
  - $r = f/m = f/m_{unique} = 2N^3/3N^2 = O(N)$
  - Computationally intensive (if perfect caching)
  - Peak performance expected for cache-based processor



OPTIMIZING MATRIX MULTIPLY FOR A CPU

Or: how to program a cache

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 NAIVE

Performance for single-threaded CPU run

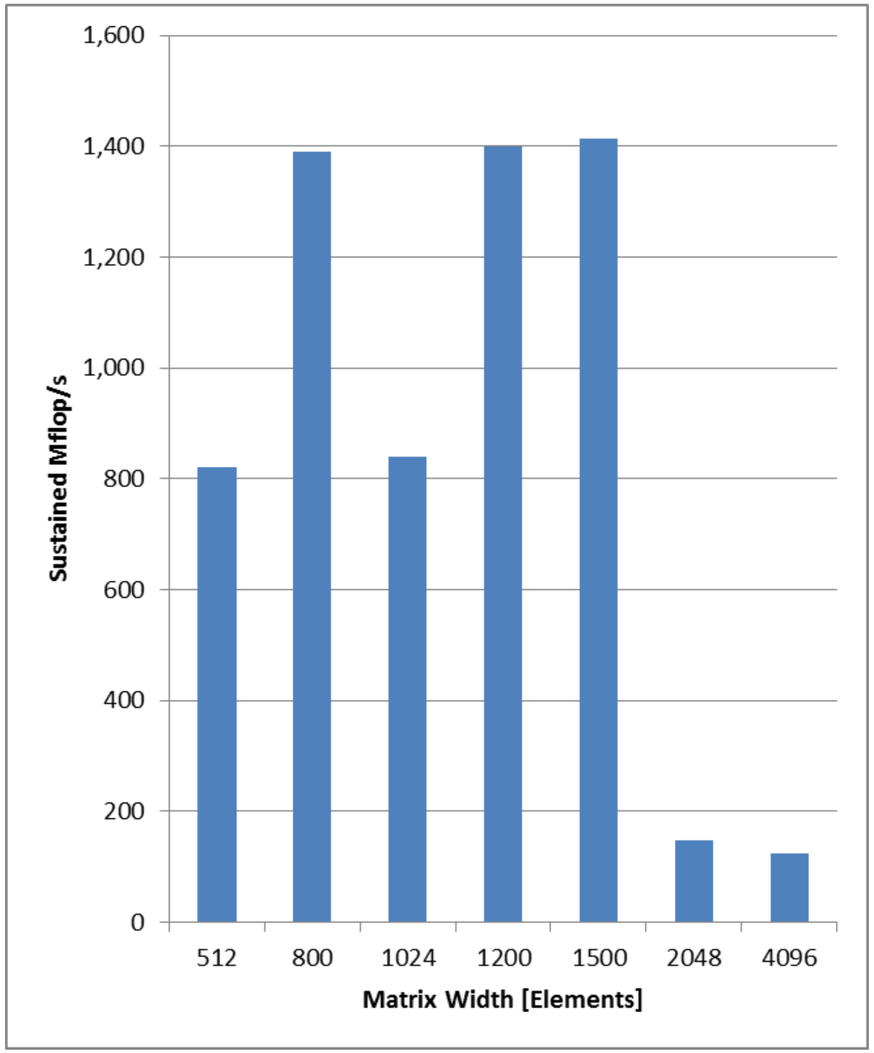
- Single precision (float, SP)
- Xeon E5 Sandy Bridge
- 4 cores @ 2.4GHz (76.8 GFLOP/s peak)

High performance until 1500x1500 elements?

- Fits in cache (10MB) – capacity!
- $(1.5k\ elements)^2$ x 4B (float) = 9MB/matrix
- $(2k\ elements)^2$ x 4B (float) = 16MB/matrix

Reason for drops @ 512 and 1024 though?

- Evictions due to conflicts



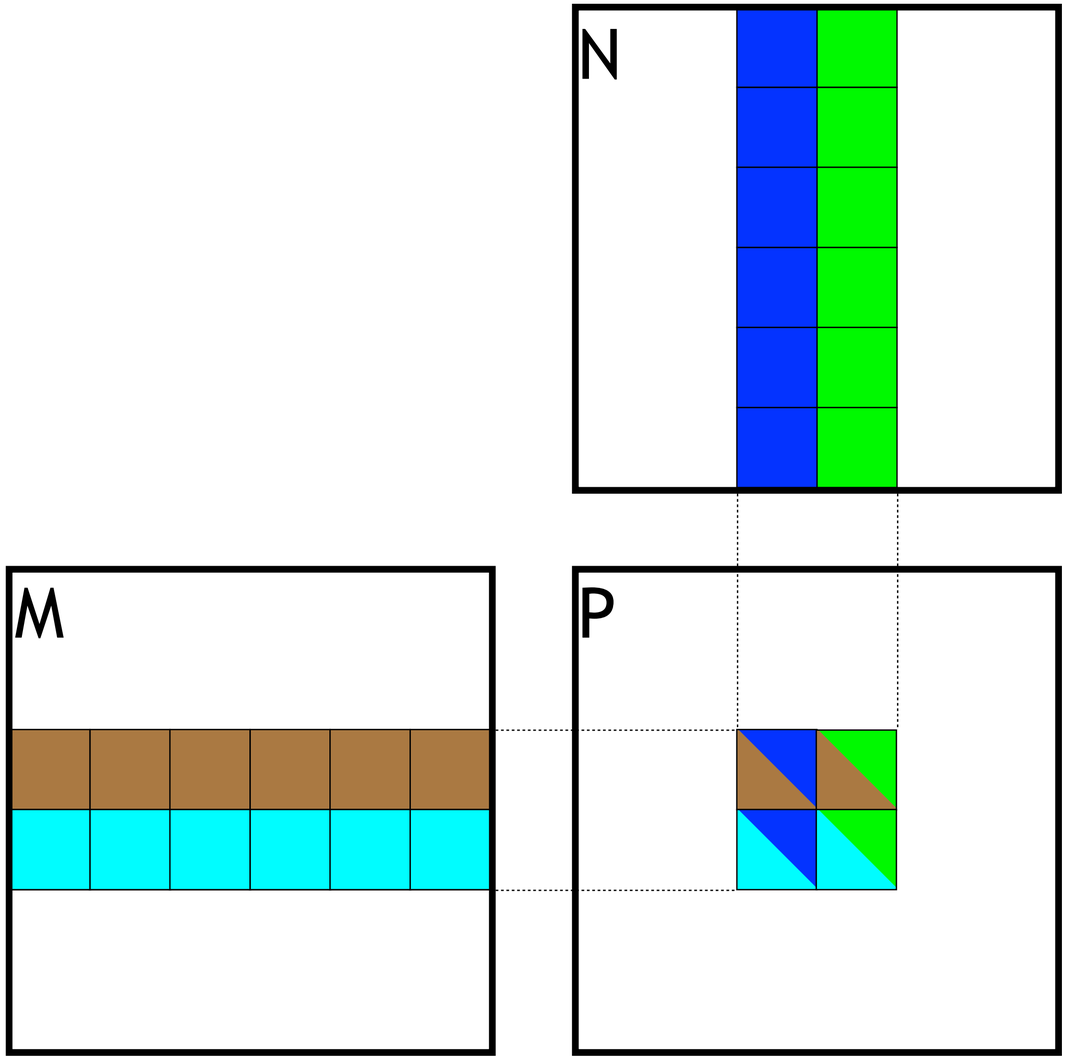
MATRIX MULTIPLY – CPU TILED/BLOCKED

Addition is associative

- a + (b + c) = (a + b) + c
- So feel free to reorder multiply operations
- Goal: increase cache hit rate

Block size is architecture-dependent parameter

- Cache size



MATRIX MULTIPLY – CPU BLOCKED

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

`min` only relevant if `matWidth` is not a full multiple of `blockSize`

MEMORY ACCESS PATTERN

Trace for naive implementation

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

Trace for blocks of two-by-two

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

No locality - RED

Spatial locality - RED

Temporal locality - RED

PERFORMANCE ANALYSIS

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)

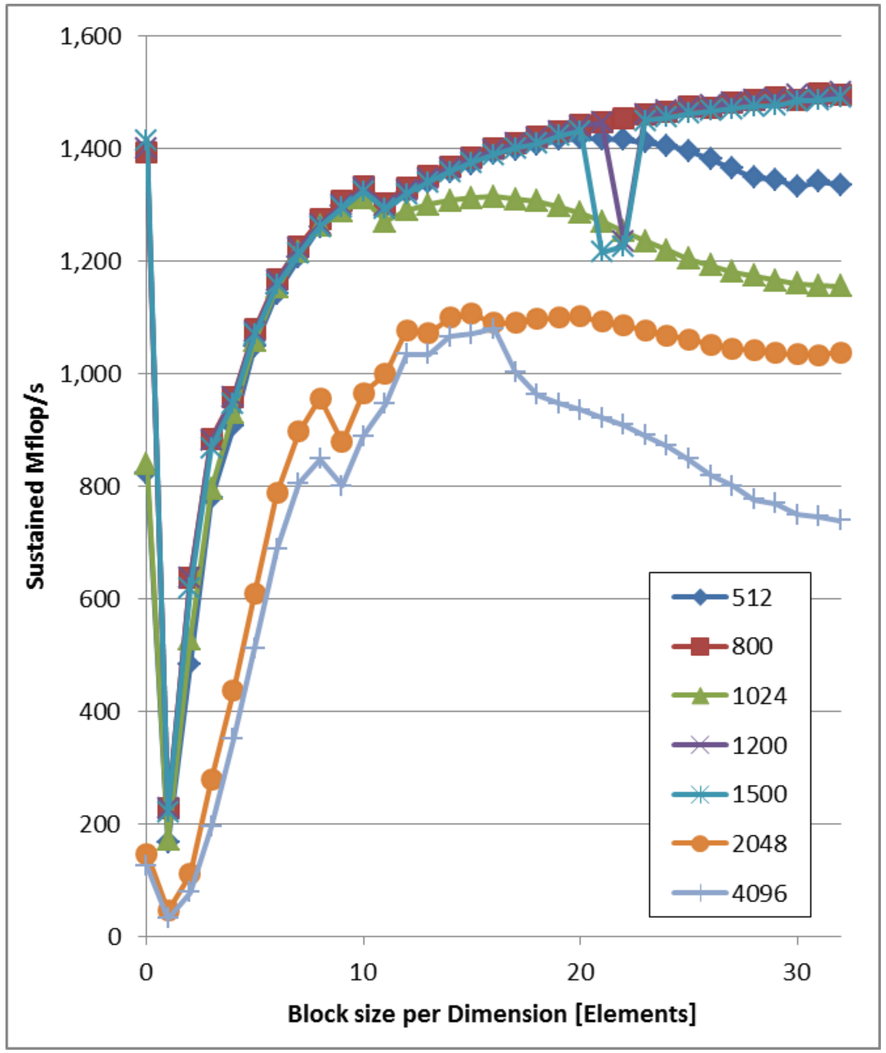
Huge drop for block size of 1?

- Control flow overhead

Non-blocked better than blocked?

- Cache size!

Factor of 5-10x for blocked vs. non-blocked typical



MATRIX MULTIPLY FOR A GPU

INITIAL GPU VERSION

GPU version

- Kernel only, data movement & control is missing
- Notice the “d”-suffix!

Two outer loops are missing

- Handled instead by a 2D thread array

Per loop

- 2 FLOPS
- 4 memory accesses

```
// Matrix multiplication kernel – thread code
__global__ void MatrixMulKernel ( float* Md,
                                  float* Nd,
                                  float* Pd,
                                  int Width )
{
  float Pvalue = 0; // intermediate result
  float Melement, Nelement;

  for ( int k = 0; k < Width; ++k ) {
    Melement = Md[threadIdx.y * Width + k];
    Nelement = Nd[k * Width + threadIdx.x];
    Pvalue += Melement * Nelement;
  }
  Pd[threadIdx.y * Width + threadIdx.x] = Pvalue;
}
```

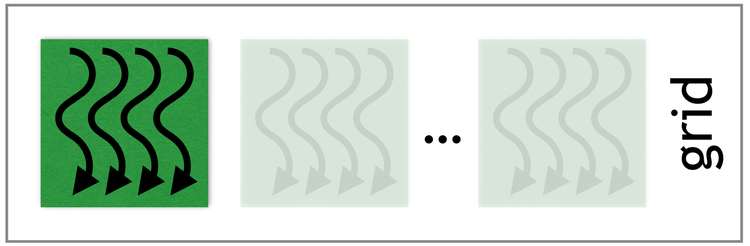
INITIAL GPU VERSION

```
void MatrixMulOnDevice ( float* M, float* N, float* P, int Width )
{
  int size = Width * Width * sizeof(float);
  float* Md, Nd, Pd;
  ...
  // Allocate and Load M, N to device memory
  cudaMalloc ( &Md, size );
  cudaMemcpy ( Md, M, size, cudaMemcpyHostToDevice );
  cudaMalloc ( &Nd, size );
  cudaMemcpy ( Nd, N, size, cudaMemcpyHostToDevice );
  // Allocate P on the device
  cudaMalloc ( &Pd, size );

  // Setup the execution configuration
  dim3 dimGrid ( 1, 1 );
  dim3 dimBlock ( Width, Width );
  MatrixMulKernel <<< dimGrid, dimBlock >>> ( Md, Nd, Pd, Width );

  // Read P from the device
  cudaMemcpy ( P, Pd, size, cudaMemcpyDeviceToHost );
  // Free device matrices
  cudaFree ( Md ); cudaFree ( Nd ); cudaFree ( Pd );
}
```

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 multiplication, one add
- Write Pd



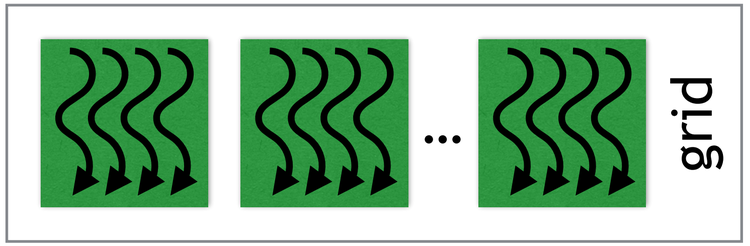
Issue 1: Matrix size limited by threads/block

Issue 2: Compute/Memory ratio

- (= Computational intensity)
- No cache => $m = m_{all} = 4N^3$
- $r = f/m = 2N^3/4N^3 = 1/2$ (very low)
- In FLOPS/Byte even worse: 2 FLOPS vs 16 Bytes = 1/8 (horrible)

MULTIPLE THREAD BLOCKS

Multiple thread blocks, organized in a 2D array



Each block:

- Consists of (`TILE_WIDTH`)^2 threads
- Computes (`TILE_WIDTH`)^2 sub-matrix

Resulting grid

- (`WIDTH/TILE_WIDTH`)^2 blocks

Limited by max grid size

MULTIPLE THREAD BLOCKS

```
// Matrix multiplication kernel – thread code
__global__ void MatrixMulKernel ( float* Md,
                                  float* Nd,
                                  float* Pd,
                                  int Width )
{
  float Pvalue = 0; // intermediate result
  float Melement, Nelement;
  // Calculate the row index of the Pd element
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  // Calculate the column index of the Pd element
  int col = blockIdx.x * blockDim.x + threadIdx.x;

  for ( int k = 0; k < Width; ++k ) {
    Melement = Md[row * Width + k];
    Nelement = Nd[k * Width + col];
    Pvalue += Melement * Nelement;
  }
  Pd[row * Width + col] = Pvalue;
}
```

ANALYSIS

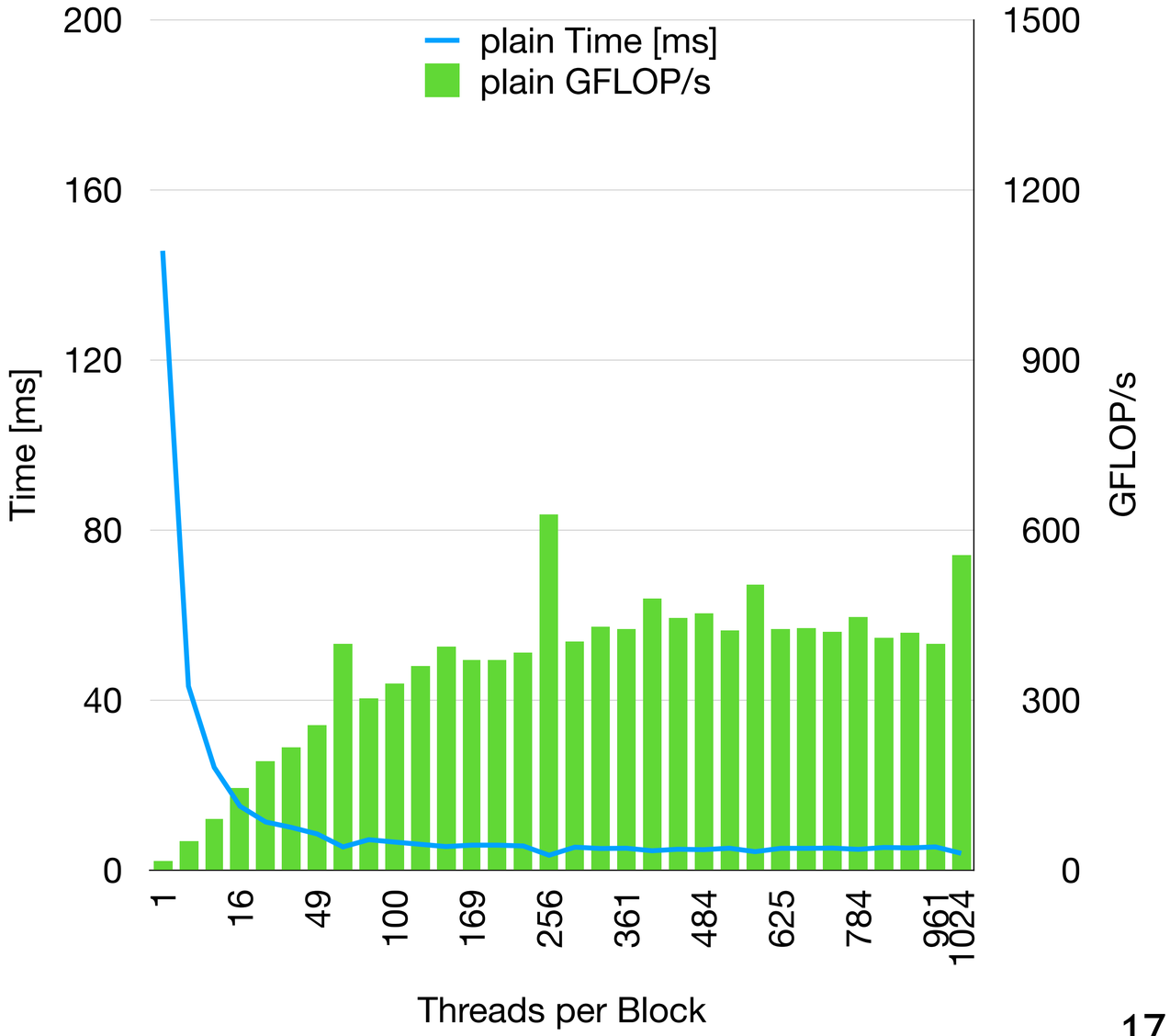
RTX 2080 GPU, Turing-class

Scheduling: varying the number of threads per block

- 1k x 1k matrix size
- Match block count
- In general: “more threads are better”, but not always like this -> e.g. register pressure

Calculating FLOP/s

- N^2 elements, each 2N FLOPS, $2N^3$ FLOPs total
- Here: 2.14 GFLOPS total
- Without data movement



ANALYSIS - UPPER BOUND?

Model	CC Revi-sion	Total global memory [bytes]	Multi-proces sors	Cores	Total constant memory [bytes]	Shared memory per block [bytes]	Regis-ters per block	Warp size	Threads per block	Max dimen-sion of a block	Max. dimen-sion of a grid	Max. memory pitch [bytes]	Clock rate [GHz]	Concurrent copy and execution
GeForce GTX 480	2,0	1.5G	15	480	64k	48k	32k	32	1k	1k x 1k x 64	65535 x 65535 x 65535	2G	1,4	Y 1
Tesla K20c	3,5	5G	13	2496	64k	48k	64k	32	1k	1k x 1k x 64	2G x 65535 x 65535	2G	0,7	Y 2
RTX 2080Ti	7,5	11G	68	4352	64k	48k	64k	32	1k	1k x 1k x 64	2G x 65535 x 65535	2G	1,54	Y 3

For single precision: 4352 * 1.54 * 2 = 13,404.16 GFLOP/s (clock boost)

ANALYSIS - UPPER BOUND?

Each thread works on global memory

- 2 32bit accesses per SP Multiply-Add
- 4B per FLOP

=> 13 TFLOPs require 52 TB/s memory bandwidth

- RTX 2080Ti: 352bit * 1750MHz (14 Gbps effective) (GDDR6) = 616 GB/s

Memory bandwidth limits performance to ~150 GFLOP/s

- GPU caches?

-> Increase flop/memory ratio!

-> Similar to blocking, but this time we have to define reuse manually

SHARED MEMORY OPTIMIZATIONS

SHARED MEMORY

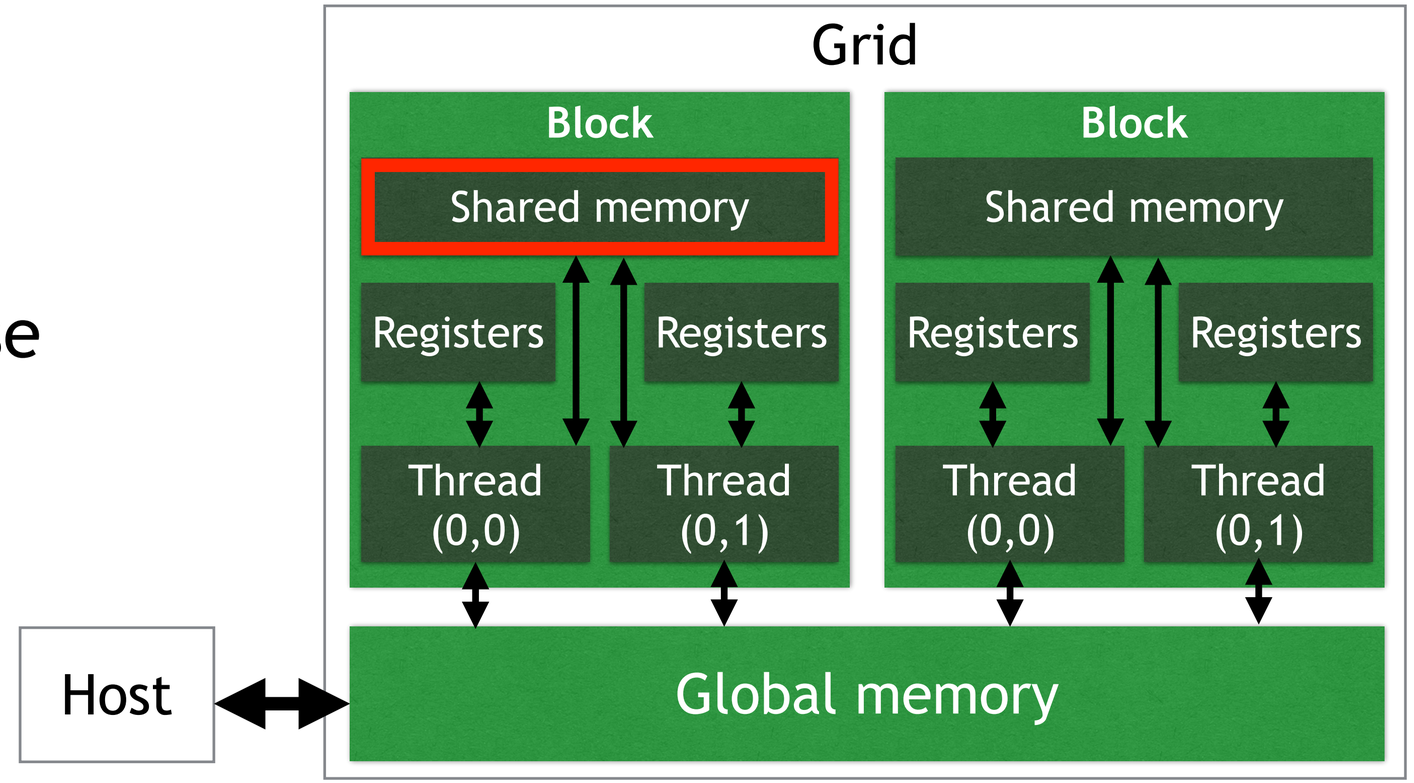
On-chip memory

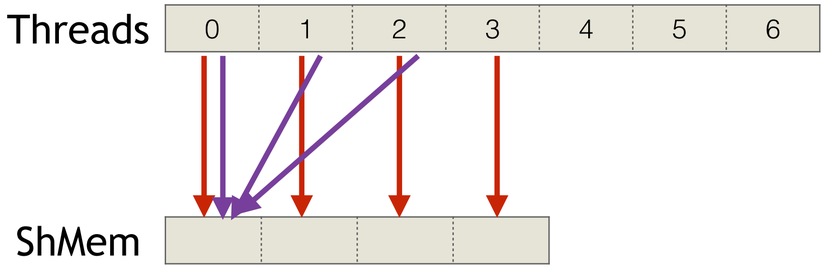
Lifetime: thread lifetime

Access costs in the best case equal register access

Organized in n banks

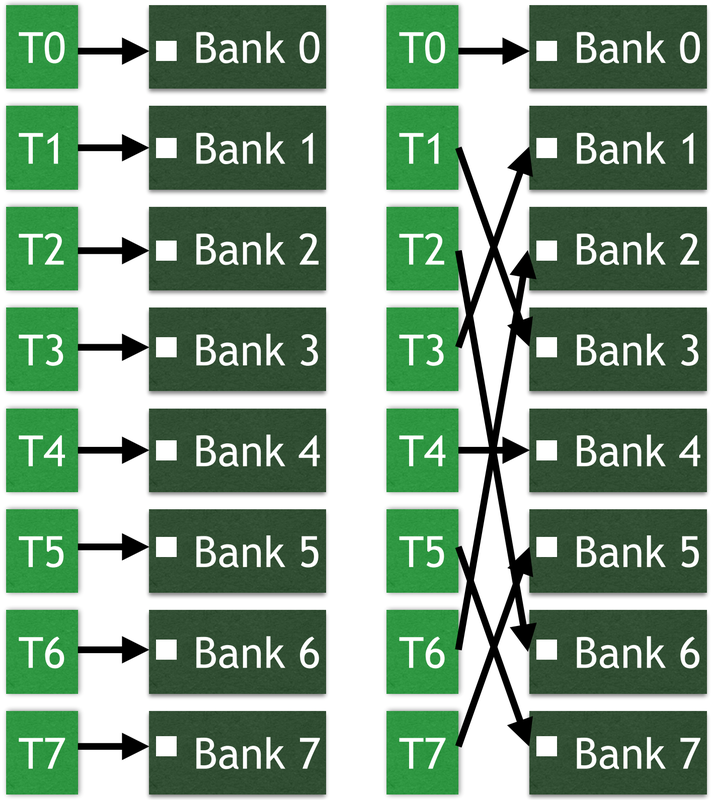
- Typ. 16-32 banks with 32bit width
- Low-order interleaving
- Parallel access if no conflict
- Conflicts result in access serialization





SHARED MEMORY BANK CONFLICTS

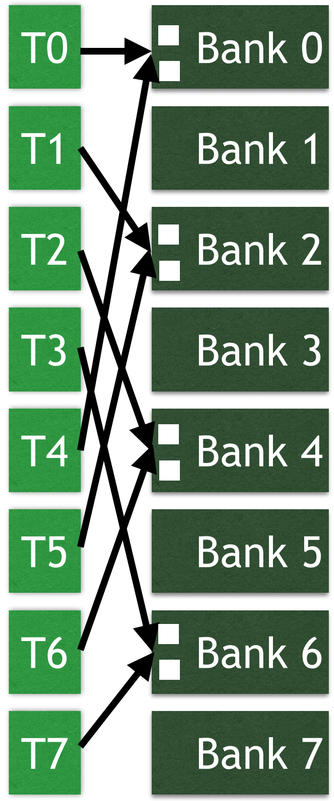
Shared memory bank access without blocking



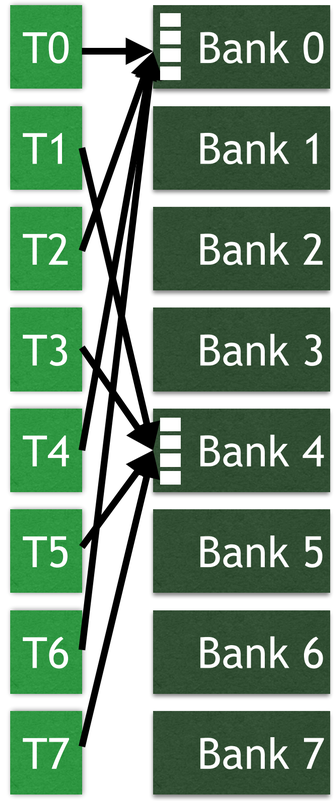
stride=1

stride=3

Shared memory bank access with blocking

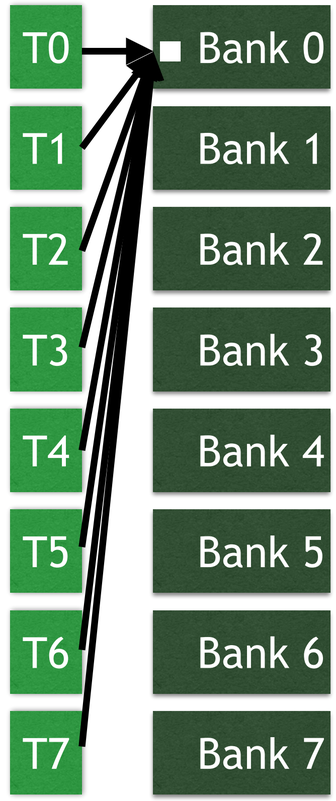


stride=2

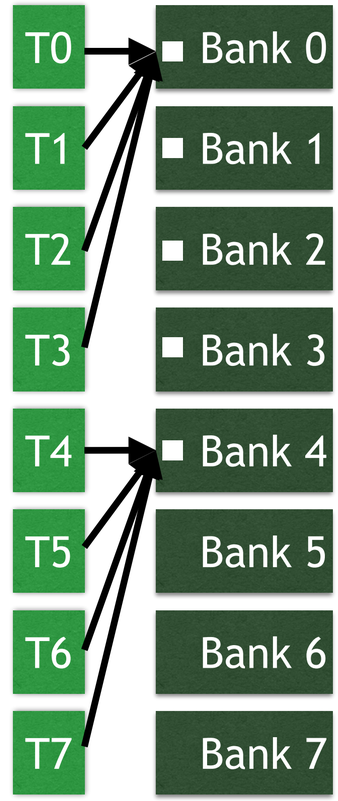


stride=4

Multi- and broadcast meantime supported



bcast



2x mcast

TILING/BLOCKING

Associativity of C = A * B

Resorting the summation of the pair-wise products

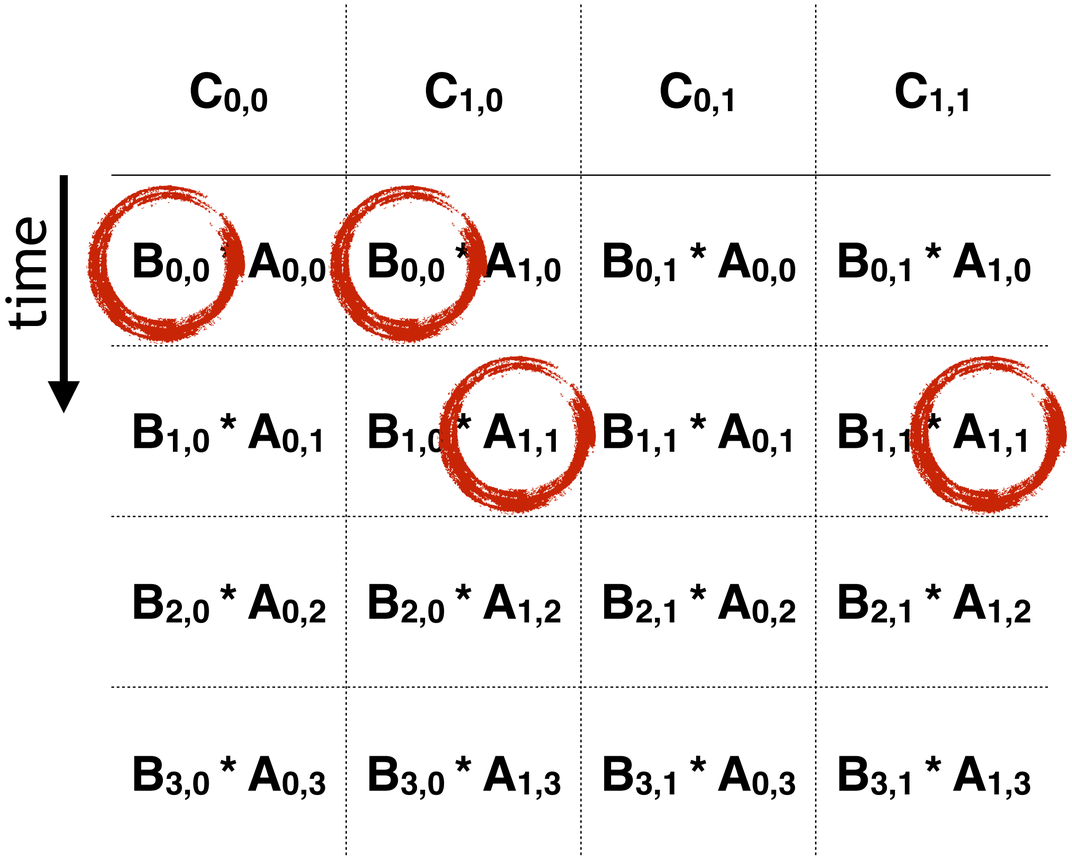
Increase locality by reordering memory accesses

=> Tiling or blocking

Each TxT tile uses each element T times

Calculate only parts of the elements of C, so that access pattern has high locality

Beneficial for both sequential and parallel algorithms



MATRIX MULTIPLY - SHARED MEMORY

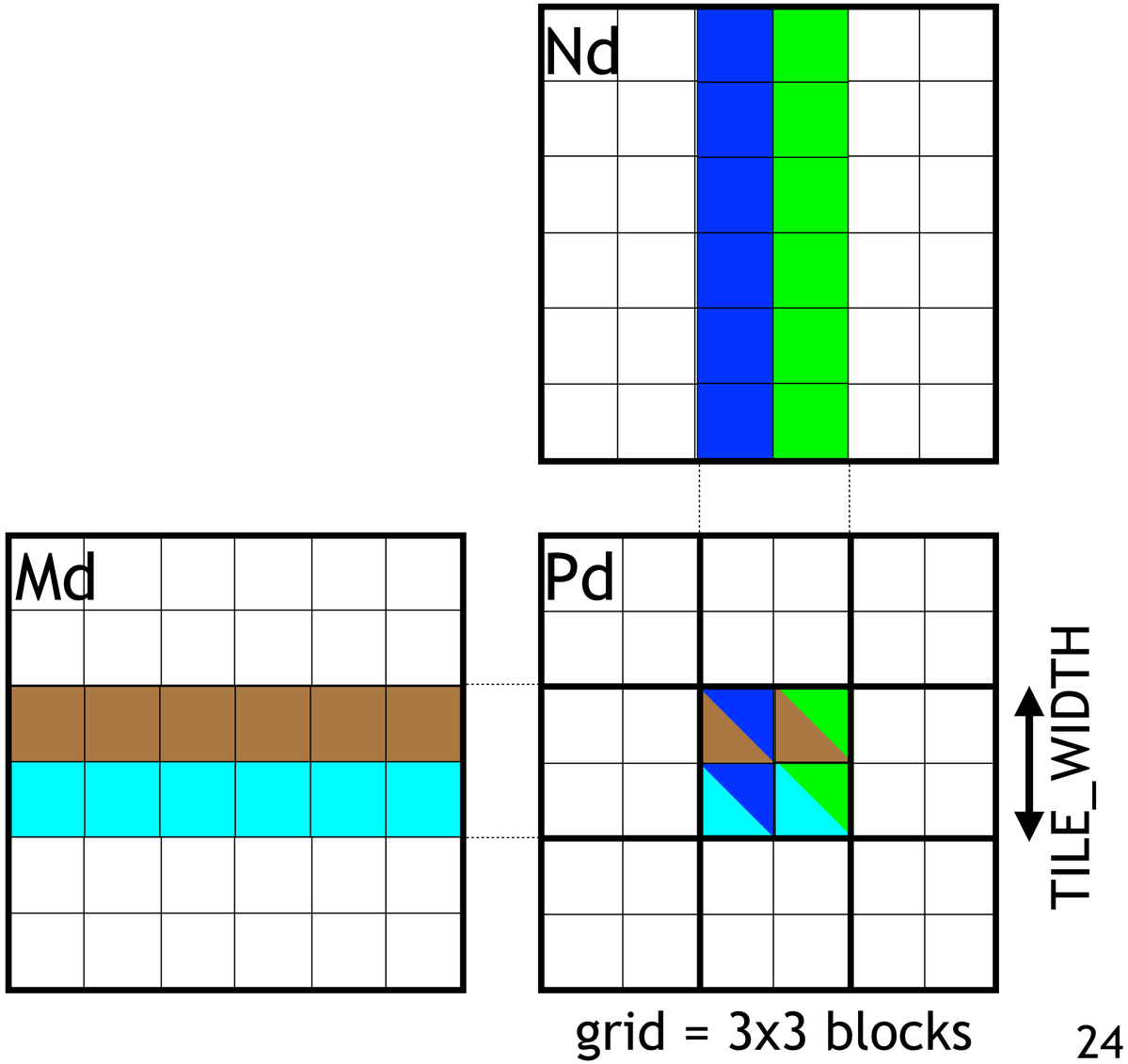
Old: each input element is being read by `TILE_WIDTH` threads

New: is read by one thread, but used by multiple threads

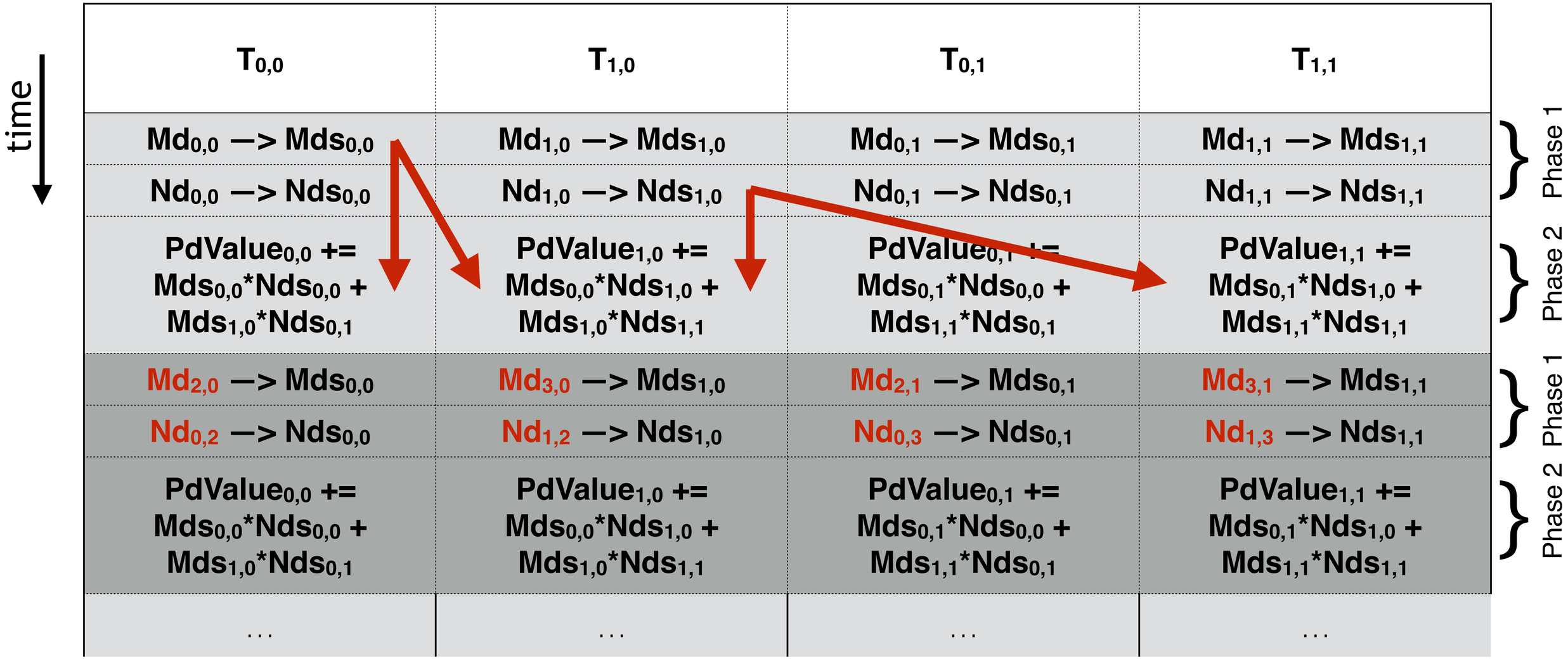
- Size of a sub-set should match a tile size

Separate kernel execution into phases

1. Fill shared memory
2. Execute
3. Repeat



SHARED MEMORY - PHASES FOR 2X2 TILE



NEW FLOP/MEMORY RATIO

Assuming a TILE_WIDTH of 16 and a 1k x 1k matrix

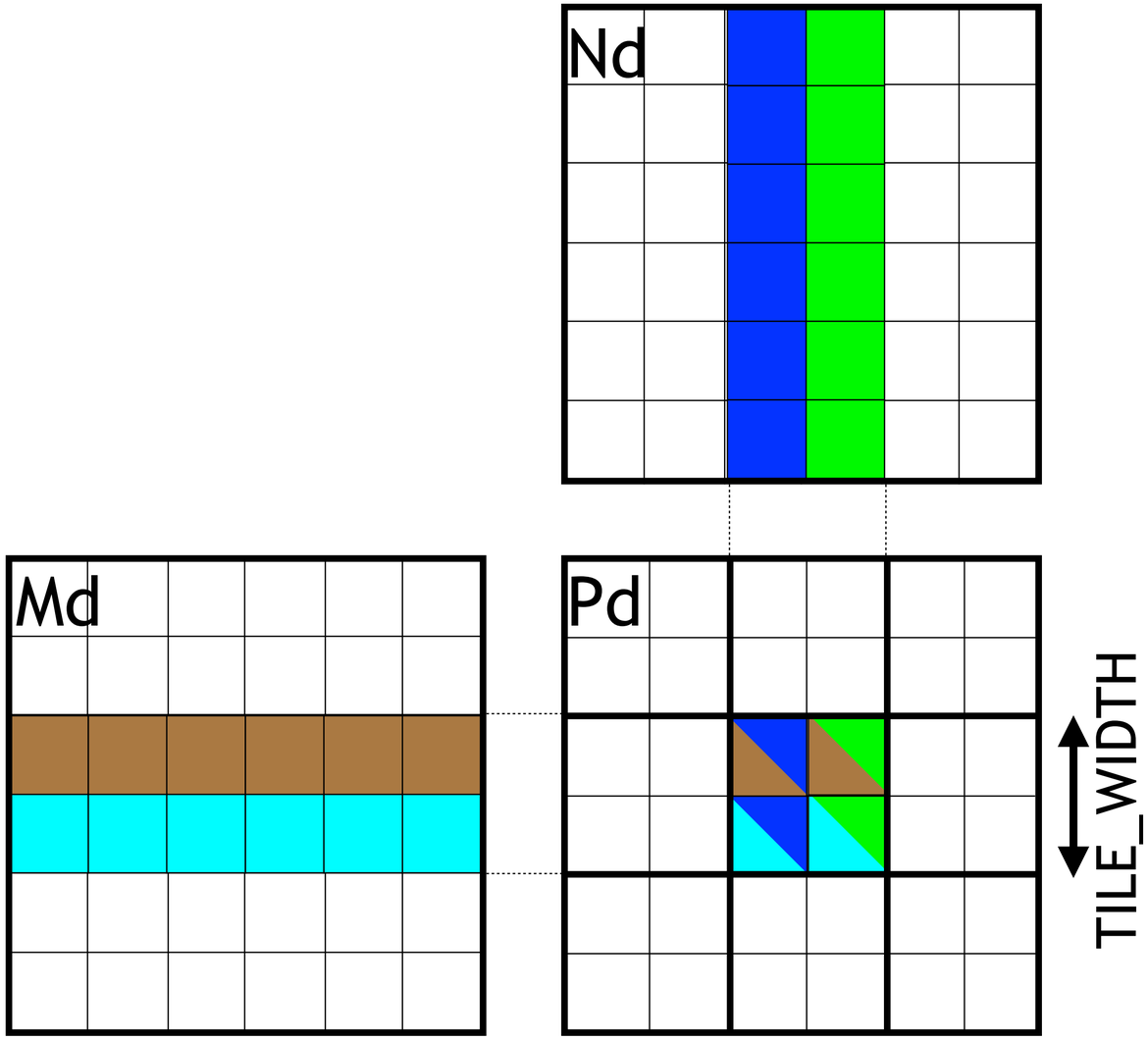
- 256 threads per block
- 1k/16 => 64 x 64 blocks

Each block

- 2 loads per thread = 512 loads
- 16 MADDs per thread = 8k flops

New flop/memory ratio

- 8k:512 = 16:1
- 16 FLOPS : 4 Bytes = 4 (good!)
- Plus improved coalescing



SHARED MEMORY IMPLEMENTATION

```
__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;
  int row = by * TILEWIDTH + ty;
  int col = bx * TILEWIDTH + tx;
  float Pvalue = 0;

  if !(Row > Width || Col > Width) {
    for ( int m = 0; m < Width / TILEWIDTH; ++m ) { // loop over tiles
      // 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 ];


      for ( int k = 0; k < TILEWIDTH; ++k )
        Pvalue += Mds[ty][k] * Nds[k][tx];

    }
    Pd[row * Width + col] = Pvalue;
  }
}
```

Something is missing here!

SHARED MEMORY IMPLEMENTATION

```
__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;
  int row = by * TILEWIDTH + ty;
  int col = bx * TILEWIDTH + tx;
  float Pvalue = 0;

  if !(Row > Width || Col > Width) {
    for ( int m = 0; m < Width / TILEWIDTH; ++m ) { // loop over tiles
      // 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;
  }
}
```

Dependencies resolved using synchronization

SHARED MEMORY RESULTS

Performance comparison for RTX 2080

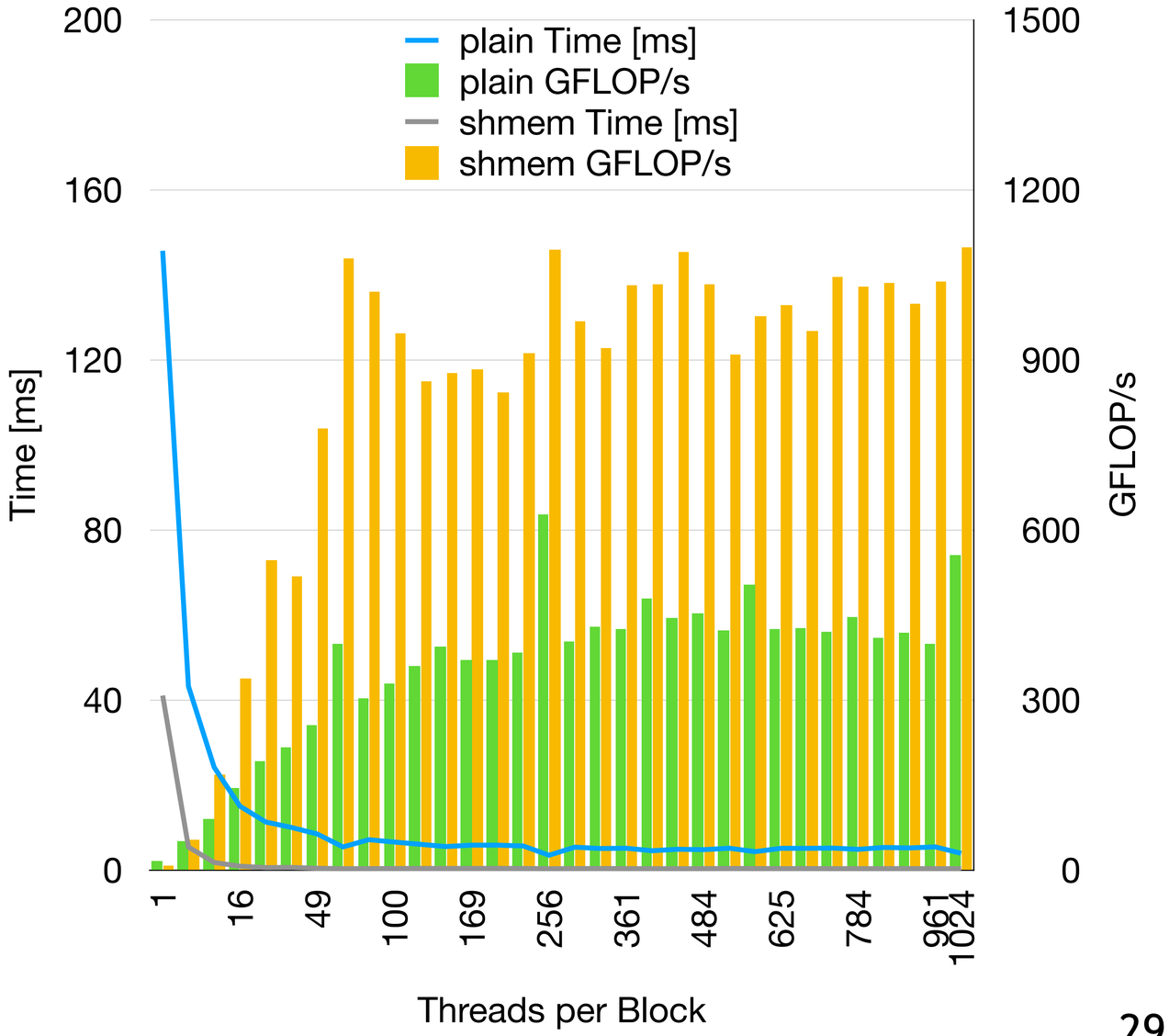
- Tiled only
- Use of shared memory

Block size <= 1k

Both code optimizations and kernel launch configuration matters!

Still a gap of about 10x to peak performance

Note: GPU's L1 cache was turned off



POSSIBLE FURTHER OPTIMIZATIONS

- Multiple output values per thread
  - Reduces the pressure on shared memory as tile data can be used in multiple calculations
  - ILP optimization
- Bank conflicts in shared memory
  - Use `nvprof` to find out about such conflicts
- Vectorize shared memory loads and stores by using compound data types
  - float2/float4
  - Effective shared memory bandwidth should increase
- Double buffering by overlapping shared memory load for first set while second set is computed
  - Effective latency hiding
  - Requirements on shared memory capacity double

THREE TYPES OF DEPENDENCIES

```
<snip>
    for ( int m = 0; m < Width / TILEWIDTH; ++m ) { // loop over tiles
      // 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>
```

RAW: true or data dependency

True dependency, so only solvable using synchronization, see (1)

WAR: anti dependency

Is a name dependency, solvable using synchronization or renaming, see (2)

WAW: output dependency

Is a name dependency, solvable using synchronization or renaming, n.a. here

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

WRAPPING UP

SUMMARY

Matrix multiply as a good example to leverage locality using the shared memory

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

Shared memory 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

- ILP, vectorized memory accesses, fix bank conflicts, double buffering