King Abdullah University of Science and Technology



KAUST

CS 380 - GPU and GPGPU Programming
Lecture 14: CUDA Memories, Pt. 1

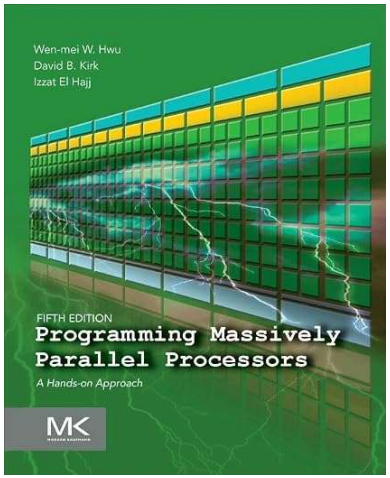
Markus Hadwiger, KAUST

Reading Assignment #8 (until Oct 26)



Read (required):

- Programming Massively Parallel Processors book, 5th edition, **Chapter 15** (*Advanced optimizations for matrix multiplication*)



Semester Project (proposal until Nov 1)



- Choosing your own topic encouraged!
  (see next slides for some suggestions)
  - Pick something that you think is really cool!
  - Can be completely graphics or completely computation, or both combined
  - Can be built on CS 380 frameworks, Vulkan SDK, CUDA SDK, ...
- Write short (1-2 pages) project proposal by Nov 1 at the latest
  - Talk to us before you start writing!
    (content and complexity should fit the lecture)
- Submit semester project with report (deadline: Dec 13)
- Present semester project, event in final exams week: Dec 14 (tbd!)

Semester Project Ideas (1)



Some ideas for topics

- Procedural shading with noise + marble etc. (GPU Gems 2, chapter 26)
- Procedural shading with noise + bump mapping (GPU Gems 2, chapter 26)
- Subdivision surfaces (GPU Gems 2, chapter 7)
- Ambient occlusion, screen space ambient occlusion
- Shadow mapping, hard shadows, soft shadows
- Deferred shading
- Particle system rendering + CUDA particle sort
- Advanced image filters: fast bilateral filtering, Gaussian kD trees
- Advanced image de-convolution (e.g., convex L1 optimization)
- PDE solvers (e.g., anisotropic diffusion filtering, 2D level set segmentation, 2D fluid flow)

Semester Project Ideas (2)



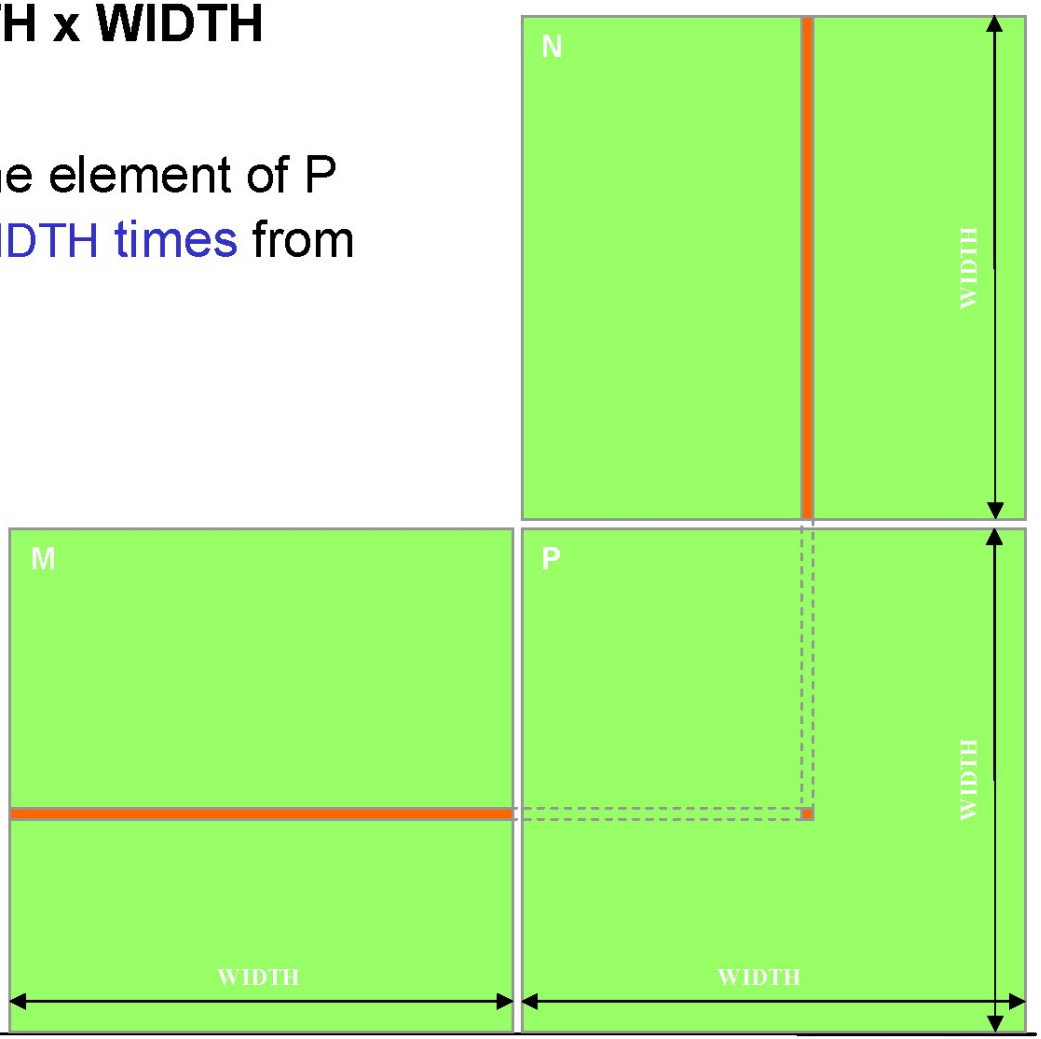
Some ideas for topics

- Distance field computation (GPU Gems 3, chapter 34)
- Livewire (“intelligent scissors“) segmentation in CUDA
- Linear systems solvers, matrix factorization (Cholesky, ...); with/without CUBLAS
- CUDA + matlab
- Fractals (Sierpinski, Koch, ...)
- Image compression
- Bilateral grid filtering for multichannel images
- Discrete wavelet transforms
- Fast histogram computations
- Terrain rendering from height map images; clipmaps or adaptive tesselation

Code Example: Matrix Multiply

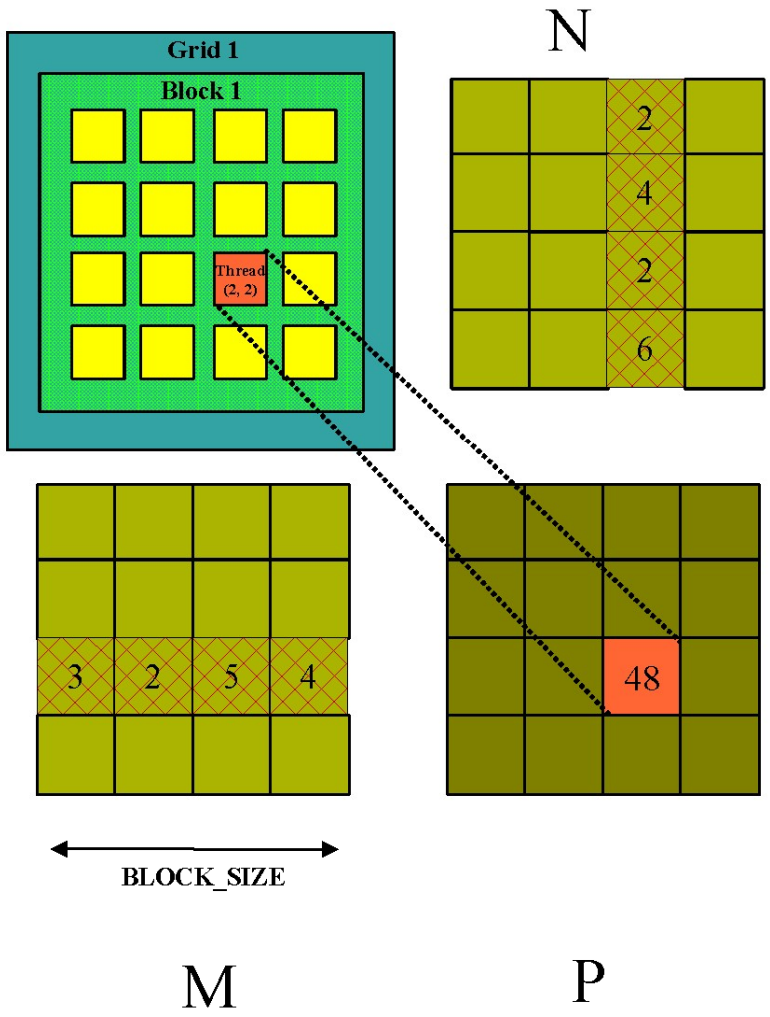
Programming Model: Square Matrix Multiplication

- **P = M * N of size WIDTH x WIDTH**
- **Without tiling:**
  - One thread handles one element of P
  - M and N are loaded WIDTH times from global memory



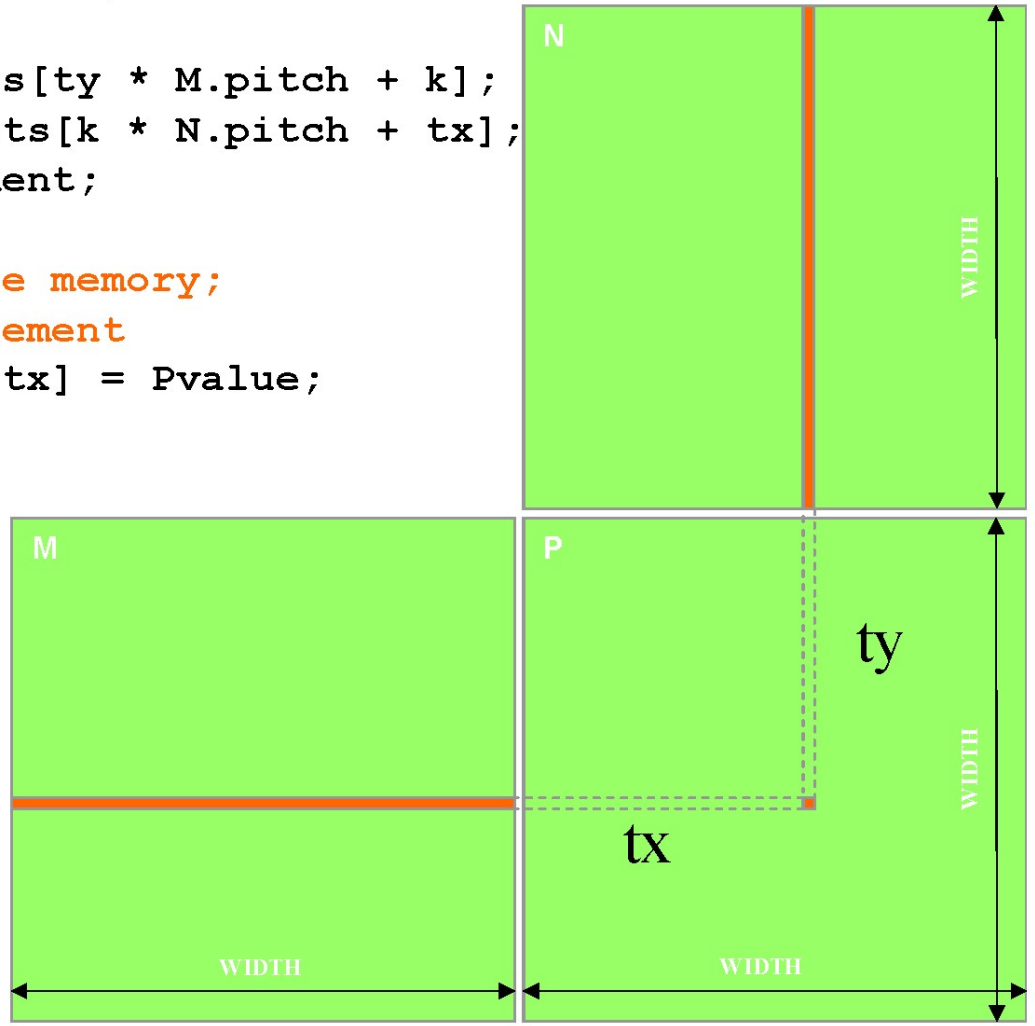
Multiply Using One Thread Block

- **One block of threads computes matrix P**
  - Each thread computes one element of P
- **Each thread**
  - Loads a row of matrix M
  - Loads a column of matrix N
  - Perform one multiply and addition for each pair of M and N elements
  - Compute to off-chip memory access ratio close to 1:1 (not very high)
- **Size of matrix limited by the number of threads allowed in a thread block**



Matrix Multiplication
Device-Side Kernel Function (cont.)

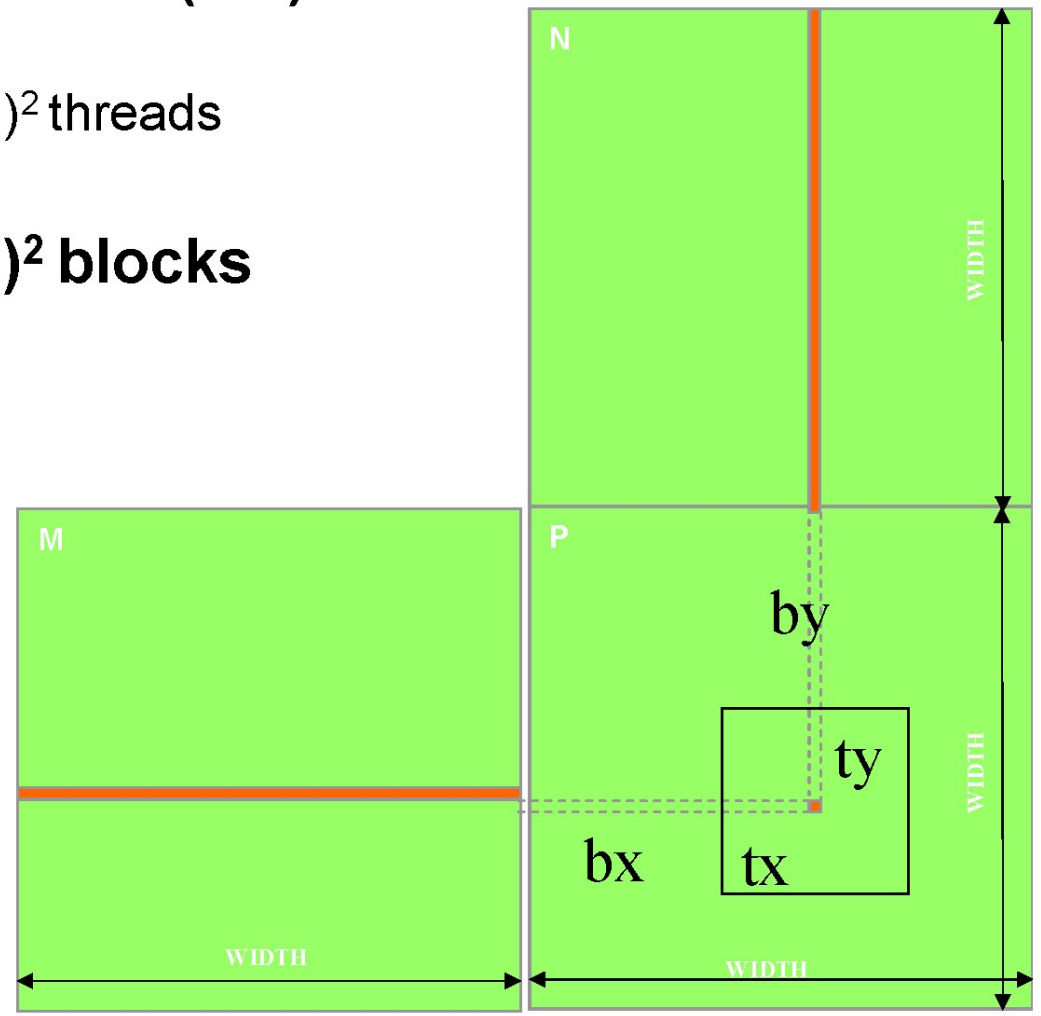
```
...
    for (int k = 0; k < M.width; ++k)
    {
      float Melement = M.elements[ty * M.pitch + k];
      float Nelement = Nd.elements[k * N.pitch + tx];
      Pvalue += Melement * Nelement;
    }
    // Write the matrix to device memory;
    // each thread writes one element
    P.elements[ty * blockDim.x+ tx] = Pvalue;
}
```



Handling Arbitrary Sized Square Matrices

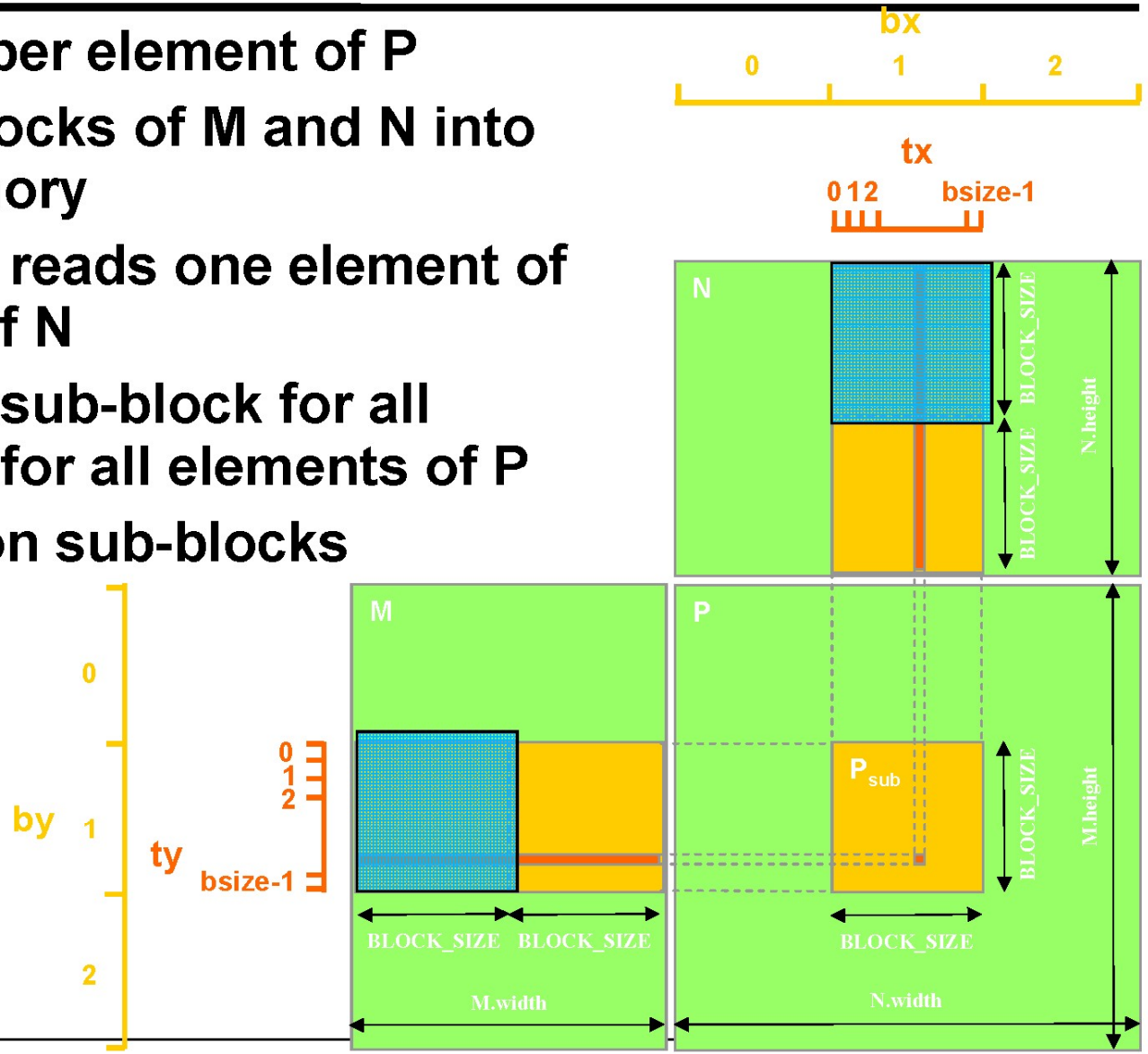
- **Have each 2D thread block to compute a $(BLOCK_WIDTH)^2$ sub-matrix (tile) of the result matrix**
  - Each has $(BLOCK_WIDTH)^2$ threads
- **Generate a 2D Grid of $(WIDTH/BLOCK_WIDTH)^2$ blocks**

You still need to put a loop around the kernel call for cases where WIDTH is greater than Max grid size!



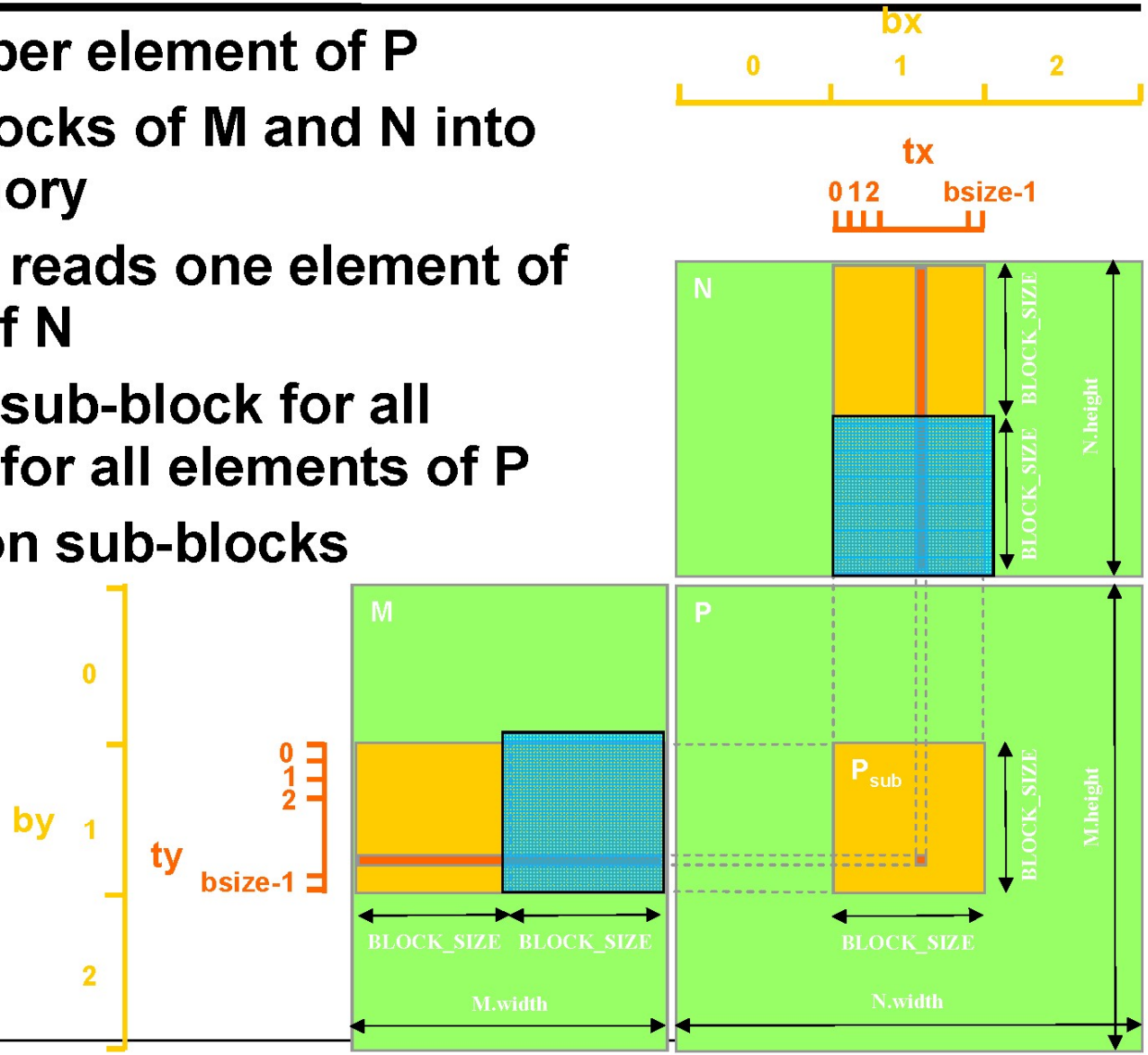
Multiply Using Several Blocks - Idea

- One thread per element of P
- Load sub-blocks of M and N into shared memory
- Each thread reads one element of M and one of N
- Reuse each sub-block for all threads, i.e. for all elements of P
- Outer loop on sub-blocks



Multiply Using Several Blocks - Idea

- One thread per element of P
- Load sub-blocks of M and N into shared memory
- Each thread reads one element of M and one of N
- Reuse each sub-block for all threads, i.e. for all elements of P
- Outer loop on sub-blocks



Example: Matrix Multiplication (1)



- Copy matrices to device; invoke kernel; copy result matrix back to host

```
// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = d_A.stride = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc((void**)&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size,
               cudaMemcpyHostToDevice);
    Matrix d_B;
    d_B.width = d_B.stride = B.width; d_B.height = B.height;
    size = B.width * B.height * sizeof(float);
    cudaMalloc((void**)&d_B.elements, size);
    cudaMemcpy(d_B.elements, B.elements, size,
               cudaMemcpyHostToDevice);
```

Example: Matrix Multiplication (2)



```
    // Allocate C in device memory
    Matrix d_C;
    d_C.width = d_C.stride = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc((void**)&d_C.elements, size);

    // Invoke kernel
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

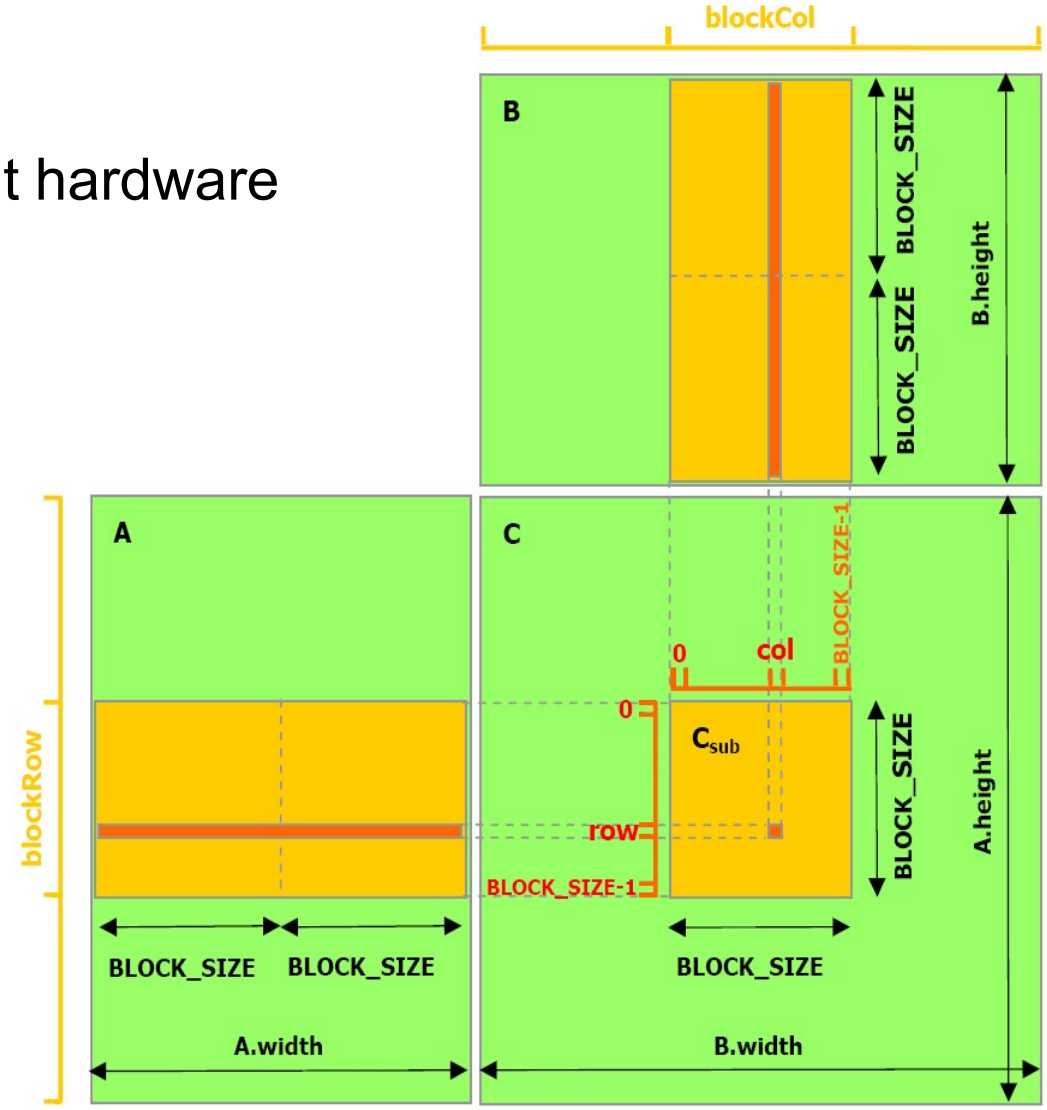
    // Read C from device memory
    cudaMemcpy(C.elements, d_C.elements, size,
               cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A.elements);
    cudaFree(d_B.elements);
    cudaFree(d_C.elements);
}
```

Example: Matrix Multiplication (3)

- Multiply matrix block-wise
- Set BLOCK_SIZE for efficient hardware use, e.g., to 16 on cc. 1.x or 16 or 32 on cc. 2.x +

- Maximize parallelism
  - Launch as many threads per block as block elements
  - Each thread fetches one element per block
  - Perform row * column dot products in parallel



Example: Matrix Multiplication (4)



```
__global__ void MatrixMul( float *matA, float *matB, float *matC, int w )
{
      __shared__ float blockA[ BLOCK_SIZE ][ BLOCK_SIZE ];
      __shared__ float blockB[ BLOCK_SIZE ][ BLOCK_SIZE ];

      int bx = blockIdx.x; int tx = threadIdx.x;
      int by = blockIdx.y; int ty = threadIdx.y;

      int col = bx * BLOCK_SIZE + tx;
      int row = by * BLOCK_SIZE + ty;

      float out = 0.0f;
      for ( int m = 0; m < w / BLOCK_SIZE; m++ ) {

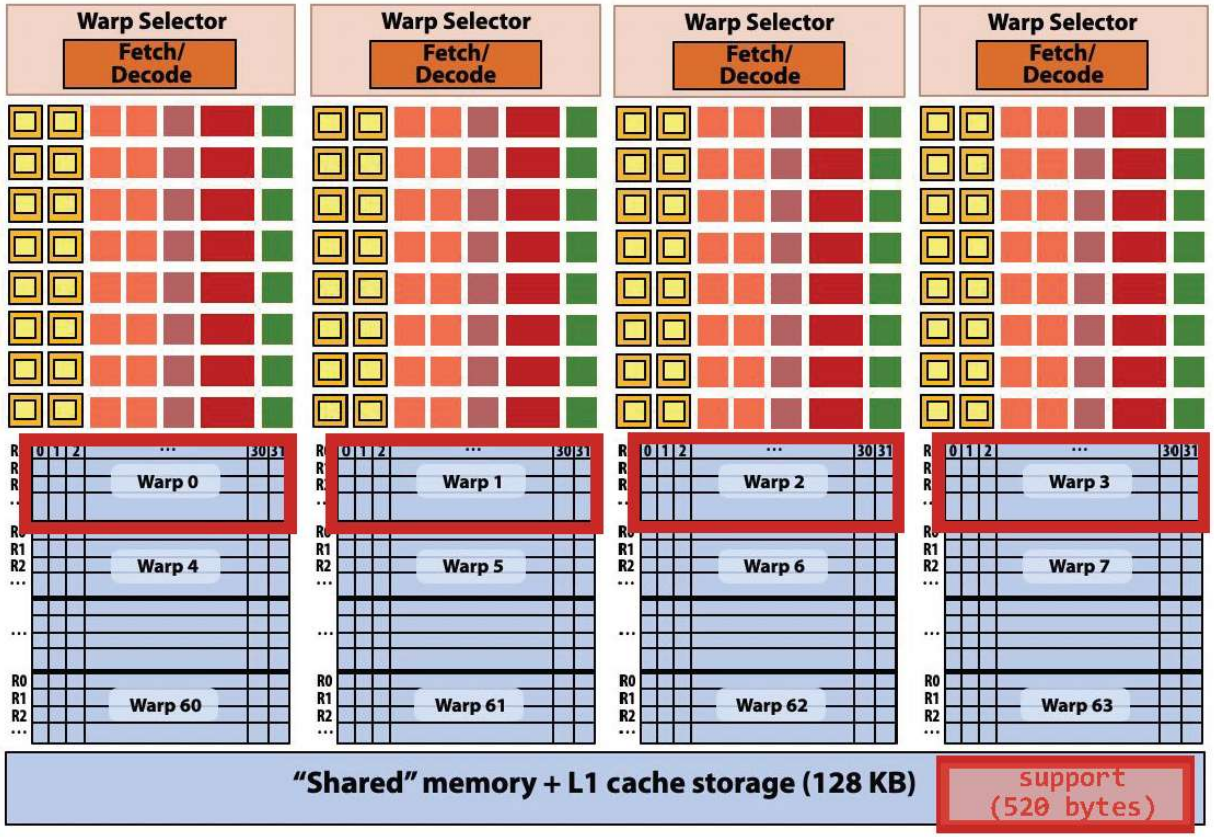
          blockA[ ty ][ tx ] = matA[ row * w +   m * BLOCK_SIZE + tx       ];
          blockB[ ty ][ tx ] = matB[ col     + ( m * BLOCK_SIZE + ty ) * w ];
          __syncthreads();

          for ( int k = 0; k < BLOCK_SIZE; k++ ) {
              out += blockA[ ty ][ k ] * blockB[ k ][ tx ];
          }
          __syncthreads();
      }

      matC[ row * w + col ] = out;
}
```

Caveat: for brevity, this code assumes matrix sizes are a multiple of the block size (either because they really are, or because padding is used; otherwise guard code would need to be added)

Running on a V100 (Volta) SM




```
#define THREADS_PER_BLK 128

__global__ void convolve(int N, float* input,
                         float* output)
{
   __shared__ float support[THREADS_PER_BLK+2];
   int index = blockIdx.x * blockDim.x +
               threadIdx.x;

   support[threadIdx.x] = input[index];
   if (threadIdx.x < 2) {
      support[THREADS_PER_BLK+threadIdx.x]
        = input[index+THREADS_PER_BLK];
   }

   __syncthreads();

   float result = 0.0f;  // thread-local
   for (int i=0; i<3; i++)
     result += support[threadIdx.x + i];

   output[index] = result / 3.f;
}
```

A `convolve` thread block is executed by 4 warps
(4 warps x 32 threads/warp = 128 CUDA threads per block)

(sub-core == SM partition)

SM core operation each clock:

- **Each sub-core selects one runnable warp (from the 16 warps in its partition)**
- **Each sub-core runs next instruction for the CUDA threads in the warp (this instruction may apply to all or a subset of the CUDA threads in a warp depending on divergence)**

courtesy Kayvon Fatahalian

Limits in CUDA Programming Guide



	Compute Capability													
Technical Specifications	5.0	5.2	5.3	6.0	6.1	6.2	7.0	7.2	7.5	8.0	8.6	8.7	8.9	9.0
Maximum number of resident grids per device (Concurrent Kernel Execution)	32		16	128	32	16	128	16	128					
Maximum dimensionality of grid of thread blocks	3													
Maximum x -dimension of a grid of thread blocks [thread blocks]	$2^{31}-1$													
Maximum y- or z-dimension of a grid of thread blocks	65535													
Maximum dimensionality of thread block	3													
Maximum x- or y-dimensionality of a block	1024													
Maximum z-dimension of a block	64													
Maximum number of threads per block	1024													
Warp size	32													
Maximum number of resident blocks per SM	32								16	32	16		24	32
Maximum number of resident warps per SM	64								32	64	48			64
Maximum number of resident threads per SM	2048								1024	2048	1536			2048

Limits in CUDA Programming Guide



Chapter 20.2
(CUDA 13,
Oct 2, 2025

	Compute Capability								
Technical Specifications	7.5	8.0	8.6	8.7	8.9	9.0	10.0	11.0	12.0
Maximum number of resident grids per device (Concurrent Kernel Execution)	128								
Maximum dimensionality of grid of thread blocks	3								
Maximum x -dimension of a grid of thread blocks	2^{31}-1								
Maximum y- or z-dimension of a grid of thread blocks	65535								
Maximum dimensionality of thread block	3								
Maximum x- or y-dimensionality of a block	1024								
Maximum z-dimension of a block	64								
Maximum number of threads per block	1024								
Warp size	32								
Maximum number of resident blocks per SM	16	32	16		24	32		24	
Maximum number of resident warps per SM	32	64	48			64		48	
Maximum number of resident threads per SM	1024	2048	1536			2048		1536	
Number of 32-bit registers per SM	64 K								
Maximum number of 32-bit registers per thread block	64 K								
Maximum number of 32-bit registers per thread	255								
Maximum amount of shared memory per SM	64 KB	164 KB	100 KB	164 KB	100 KB	228 KB			100 KB
Maximum amount of shared memory per thread block[27]	64 KB	163 KB	99 KB	163 KB	99 KB	227 KB			99 KB
Number of shared memory banks	32								
Maximum amount of local memory per thread	512 KB								
Constant memory size	64 KB								

Limits in CUDA Programming Guide



Chapter 20.2
(CUDA 13,
Oct 2, 2025

	Compute Capability								
Technical Specifications	7.5	8.0	8.6	8.7	8.9	9.0	10.0	11.0	12.0
Maximum number of resident grids per device (Concurrent Kernel Execution)	128								
Maximum dimensionality of grid of thread blocks	3								
Maximum x -dimension of a grid of thread blocks	2^{31}-1								
Maximum y- or z-dimension of a grid of thread blocks	65535								
Maximum dimensionality of thread block	3								
Maximum x- or y-dimensionality of a block	1024								
Maximum z-dimension of a block	64								
Maximum number of threads per block	1024								
Warp size	32								
Maximum number of resident blocks per SM	16	32	16		24	32		24	
Maximum number of resident warps per SM	32	64	48			64		48	
Maximum number of resident threads per SM	1024	2048	1536			2048		1536	
Number of 32-bit registers per SM	64 K								
Maximum number of 32-bit registers per thread block	64 K								
Maximum number of 32-bit registers per thread	255								
Maximum amount of shared memory per SM	64 KB	164 KB	100 KB	164 KB	100 KB	228 KB			100 KB
Maximum amount of shared memory per thread block[27]	64 KB	163 KB	99 KB	163 KB	99 KB	227 KB			99 KB
Number of shared memory banks	32								
Maximum amount of local memory per thread	512 KB								
Constant memory size	64 KB								

What About Memory Performance?

Memory and Cache Types



Global memory

- [Device] **L2 cache**
- [SM] **L1 cache** (shared mem carved out; *or* L1 shared with tex cache)
- [SM/TPC] **Texture cache** (separate, or shared with L1 cache)
- [SM] **Read-only data cache** (storage might be same as tex cache)

Shared memory

- [SM] Shareable only between threads in same thread block
  (Hopper/CC 9.x: also thread block clusters)

Constant memory: Constant (uniform) cache

Unified memory programming: Device/host memory sharing

CUDA Memory: Shared Memory

Memory and Cache Types



Global memory

- [Device] **L2 cache**
- [SM] **L1 cache** (shared mem carved out; *or* L1 shared with tex cache)
- [SM/TPC] **Texture cache** (separate, or shared with L1 cache)
- [SM] **Read-only data cache** (storage might be same as tex cache)

Shared memory

- [SM] Shareable only between threads in same thread block
  (Hopper/CC 9.x: also thread block clusters)

Constant memory: Constant (uniform) cache

Unified memory programming: Device/host memory sharing

Shared Memory Allocation

- **2 modes**
- **Static size within kernel**

```
__shared__ float vec[256];
```

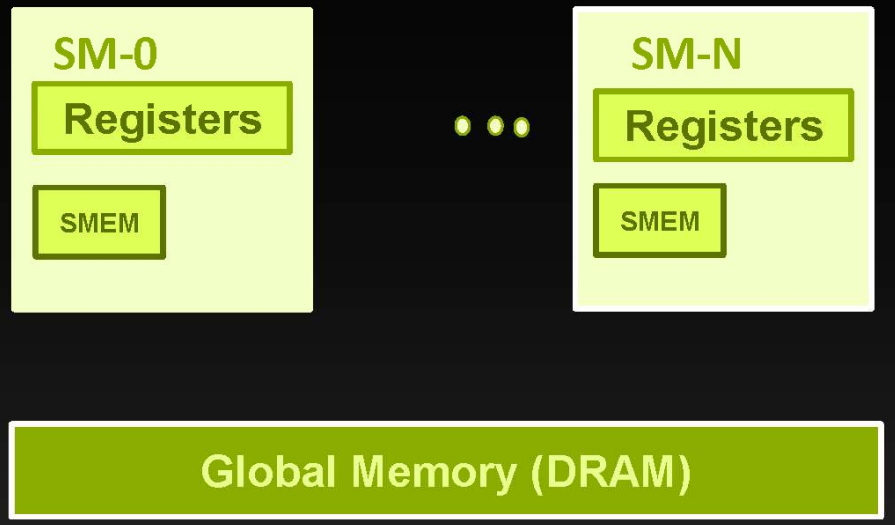
- **Dynamic size when calling the kernel**

```
// in main
int VecSize = MAX_THREADS * sizeof(float4);
vecMat<<< blockGrid, threadBlock, VecSize >>>( p1, p2, …);

// declare as extern within kernel
extern __shared__ float vec[];
```

Shared Memory

- Accessible by all threads in a block
- Fast compared to global memory
  - Low access latency
  - High bandwidth
- Common uses:
  - Software managed cache
  - Data layout conversion



© NVIDIA 2013

Shared Memory/L1 Sizing

- Shared memory and L1 use the same 64KB
  - Program-configurable split:
    - Fermi: 48:16, 16:48
    - Kepler: 48:16, 16:48, 32:32
  - CUDA API: ~~cudaDeviceSetCacheConfig(), cudaFuncSetCacheConfig()~~

later: use *carveout*

use cudaFuncSetAttribute()

- Large L1 can improve performance when:
  - Spilling registers (more lines in the cache -> fewer evictions)
- Large SMEM can improve performance when:
  - Occupancy is limited by SMEM

© NVIDIA 2013

Shared Memory

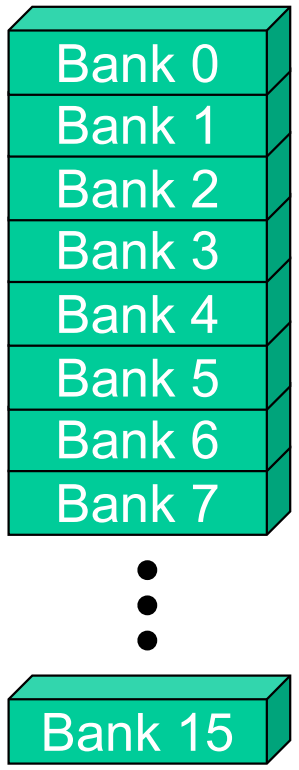
- Uses:
  - Inter-thread communication within a block
  - Cache data to reduce redundant global memory accesses
  - Use it to improve global memory access patterns
- Organization:
  - 32 banks, 4-byte (or 8-byte) banks
  - Successive words accessed through different banks

8-byte bank size is/was only available on Kepler !
(using the now deprecated cudaSharedMemConfig()
all other architectures so far use a fixed bank size of 4 bytes!

© NVIDIA 2013

Parallel Memory Architecture

- In a parallel machine, many threads access memory
  - Therefore, memory is divided into banks
  - Essential to achieve high bandwidth
- Each bank can service one address per cycle
  - A memory can service as many simultaneous accesses as it has banks
- Multiple simultaneous accesses to a bank result in a bank conflict
  - Conflicting accesses are serialized



now: 32 banks!

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009

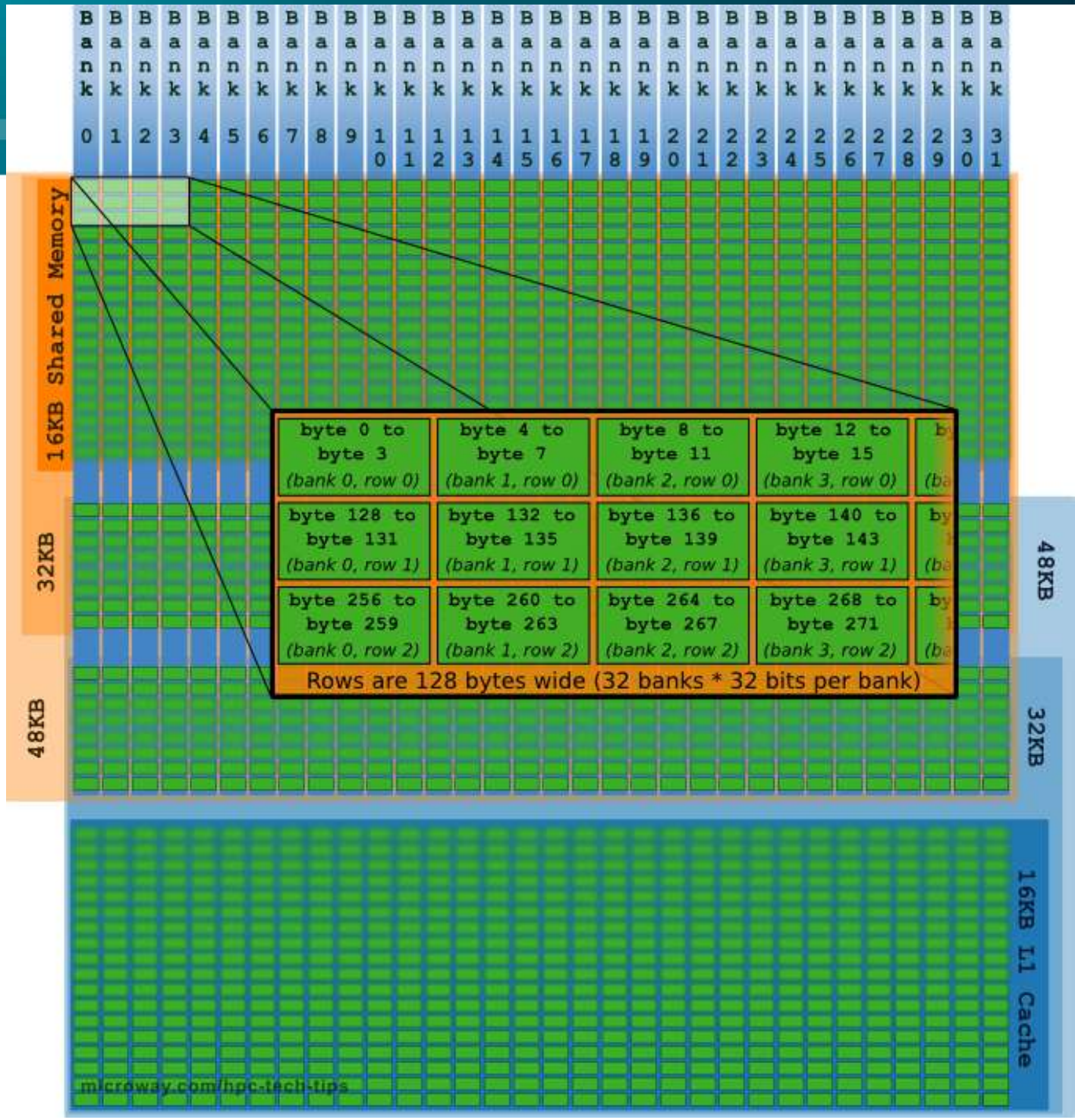
Memory Banks

all architectures
since Fermi:

32 banks

bank size:
4 bytes / bank

(only on Kepler:
optionally
configurable to
8 bytes / bank)



How addresses map to banks on G80

- Each bank has a bandwidth of 32 bits per clock cycle
- Successive 32-bit words are assigned to successive banks
- G80 has 16 banks now: 32 banks!
  - So bank = address % 16 now: % 32
  - Same as the size of a half-warp now: full warps
    - No bank conflicts between different half-warps, only within a single half-warp

Fermi and newer have 32 banks,
considers full warps instead of half warps!

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009

Shared Memory Bank Conflicts

- **Shared memory is as fast as registers if there are no bank conflicts**

- **The fast case:**
  - If all threads of a half-warp access different banks, there is no bank conflict
  - If all threads of a half-warp access the identical address, there is no bank conflict (broadcast)
- **The slow case:**
  - Bank Conflict: multiple threads in the same half-warp access the same bank
  - Must serialize the accesses
  - Cost = max # of simultaneous accesses to a single bank

full warps instead of half warps on Fermi and newer!

Thank you.