

CS 380 - GPU and GPGPU Programming

Lecture 17: CUDA Memory Access 1

Markus Hadwiger, KAUST

Quiz #4: Apr. 10



Organization

- First 30 min of lecture
- No material (book, notes, ...) allowed

Content of questions

- Lectures (both actual lectures and slides)
- Reading assignments
- Programming assignments (algorithms, methods)
- Solve short practical examples

Reading Assignment #9 (until Apr. 14)



Read (required):

- Programming Massively Parallel Processors book, Chapter 5 (*CUDA Memories*)

- **CUDA C Programming Guide 5.0**

Appendix F: Compute Capabilities

Study the different memory access requirements for different compute capabilities

Programming Assignments: Schedule



Assignment #3:

- Image Processing with (a) GLSL, and (b) CUDA due Apr 7

Assignment #4:

- Conjugate Gradient Linear Systems Solver (CUDA) due Apr 28

Stream Programming Abstraction



Let's think about our problem in a new way

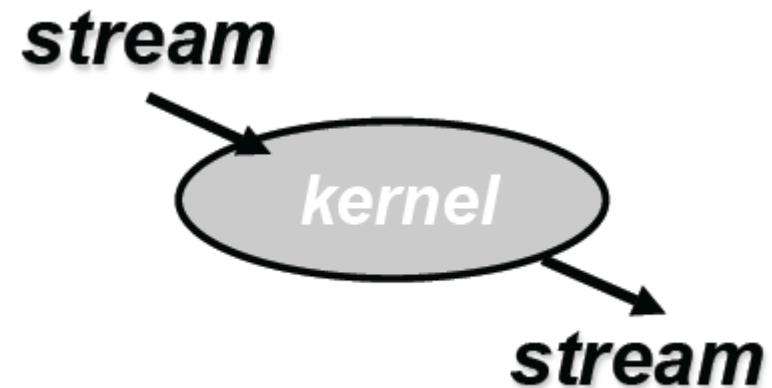
- Goal: SW programming model that matches today's VLSI

Streams

- Collection of data records
- All data is expressed in streams

Kernels

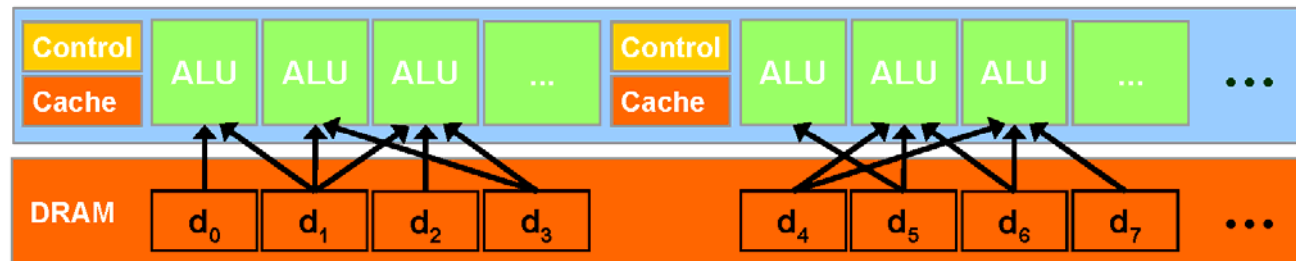
- Inputs/outputs are streams
- Perform computation on streams
- Can be chained together



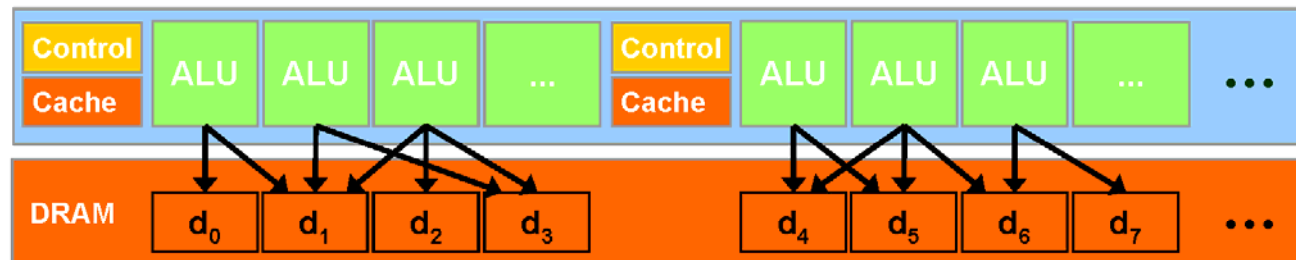
Courtesy John Owens

CUDA Highlights: Scatter

- **CUDA provides generic DRAM memory addressing**
 - Gather:



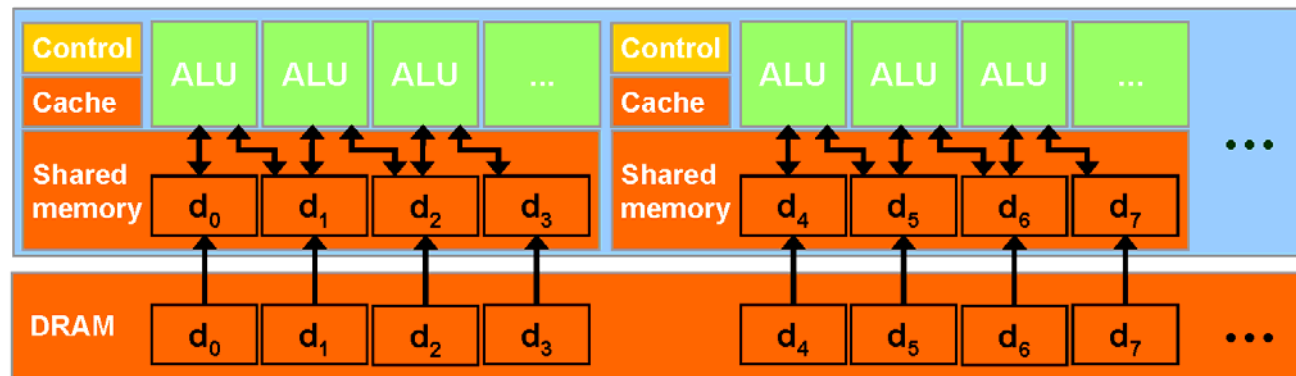
- And **scatter**: no longer limited to write one pixel



➔ **More programming flexibility**

CUDA Highlights: On-Chip Shared Memory

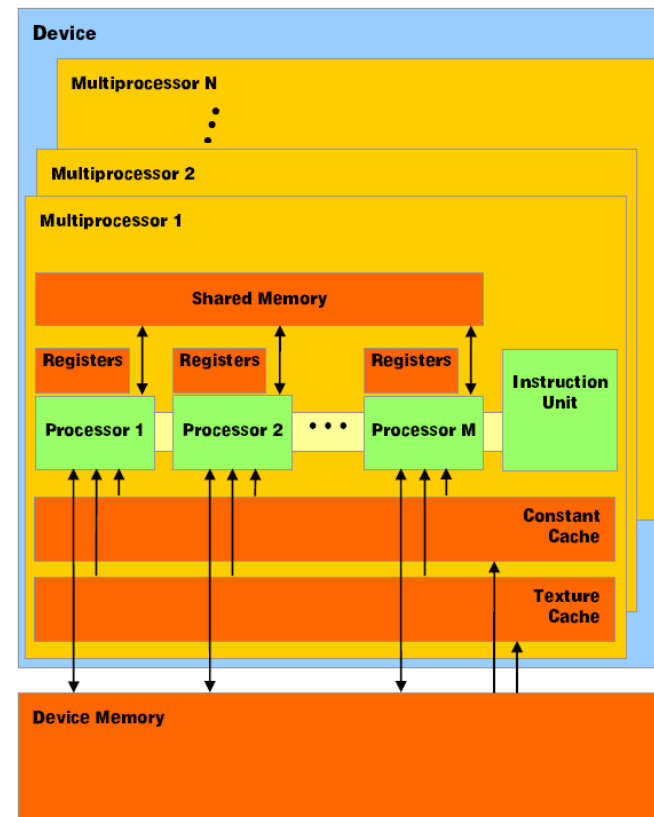
- CUDA enables access to a parallel **on-chip shared memory** for efficient inter-thread data sharing



➔ Big memory bandwidth savings

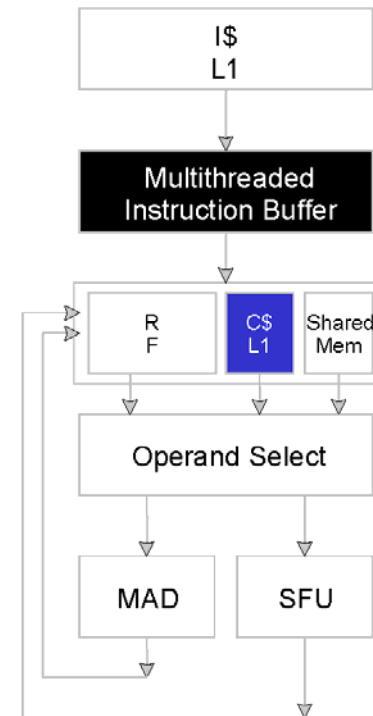
Programming Model: Memory Spaces

- **Global Memory**
 - Read-write per-grid
 - Hundreds of MBs
 - Very slow (600 clocks)
- **Texture Memory**
 - Read-only per-grid
 - Hundreds of MBs
 - Slow first access, but cached
 - Built-in filtering, clamping
- **Constant Memory**
- **Shared! Memory**
 - Read-write per-block
 - 16 KB per block
 - Very fast (4 clocks)
- **Registers**
 - Unique per thread



Constants

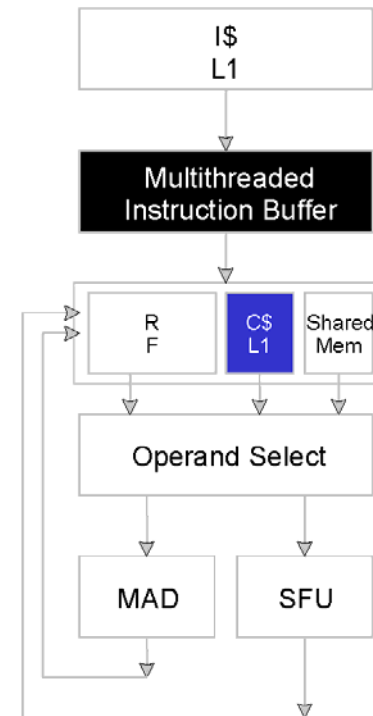
- Immediate address constants
- Indexed address constants
- Constants stored in DRAM, and cached on chip
 - L1 per SM
- **A constant value can be broadcast to all threads in a Warp**
 - Extremely efficient way of accessing a value that is common for all threads in a block!



Constants

- Immediate address constants
- Indexed address constants
- Constants stored in DRAM, and cached on chip
 - L1 per SM
- A constant value can be broadcast to all threads in a Warp
 - Extremely efficient way of accessing a value that is common for all threads in a block!

```
// specify as global variable  
__device__ __constant__ float gpuGamma[2];  
...  
// copy gamma value to constant device memory  
cudaMemcpyToSymbol(gpuGamma, &gamma, sizeof(float));  
// access as global variable in kernel  
res = gpuGamma[0] * threadIdx.x;
```

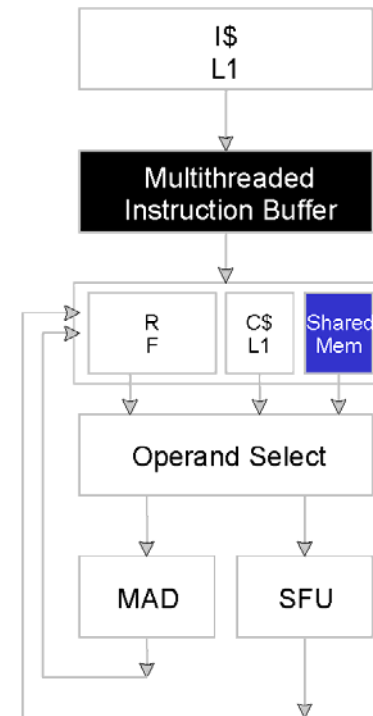


Shared Memory

- **Each SM has 16 KB of Shared Memory** or 48 KB on Fermi/Kepler (organized in 32 banks)
 - 16 banks of 32bit words
- **CUDA uses Shared Memory as shared storage visible to all threads in a thread block**
 - read and write access
- **Not used explicitly for pixel shader programs**
 - we dislike pixels talking to each other ☺

On Fermi/Kepler, there is a hardware-managed L1 cache in the same 64KB memory space as shared memory:

either 16KB shared + 48KB L1
or 48KB shared + 16KB L1



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
```

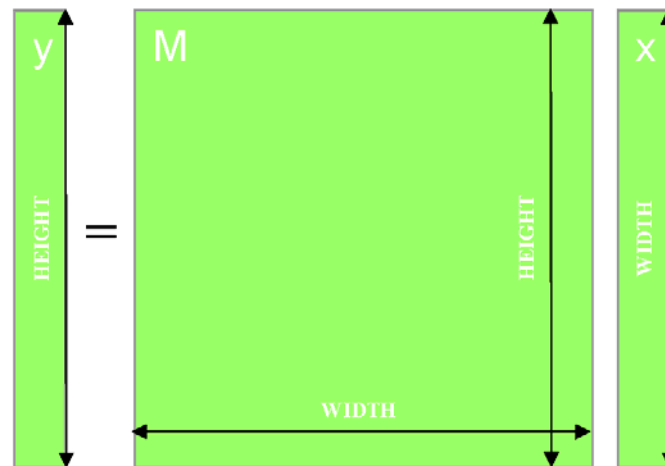
```
extern __shared__ float vec[];
```

Vector-Matrix Multiplication - data parallelism -

$$y = Mx$$

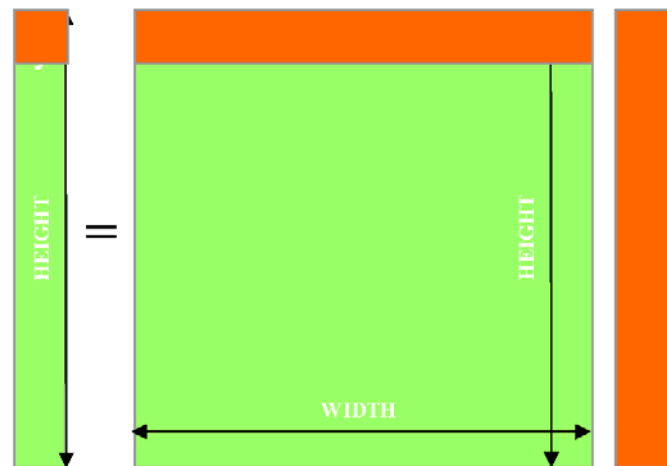
Vector-Matrix Multiplication V1

- Every thread computes a single output value in y
- Every thread computes the dot product between one line of M and x



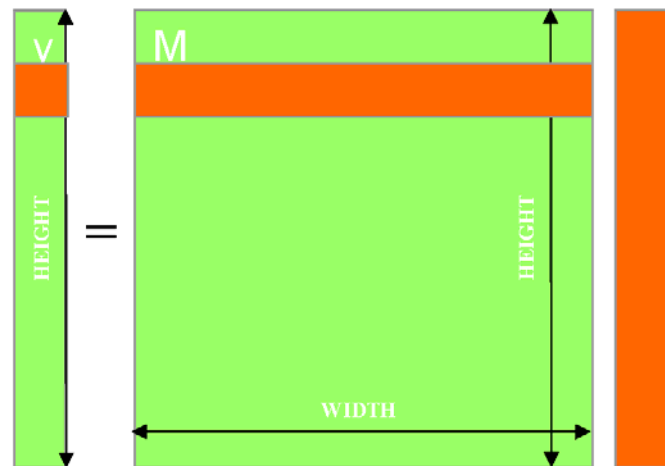
Vector-Matrix Multiplication V1

- Every thread computes a single output $y[i]$
- Every thread computes the dot product between one line of M and x



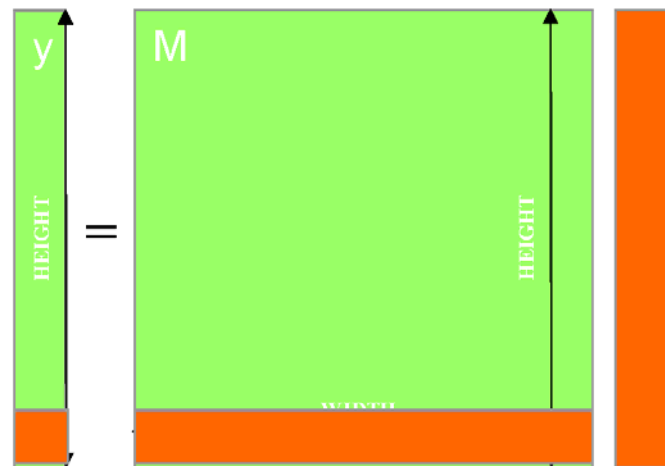
Vector-Matrix Multiplication V1

- Every thread computes a single output $y[i]$
- Every thread computes the dot product between one line of M and x



Vector-Matrix Multiplication V1

- Every thread computes a single output $y[i]$
- Every thread computes the dot product between one line of M and x
- Computations totally independent



Setup

```
... // allocate memory
float* gpuMat, gpuVec, gpuResVec;
CUDA_SAFE_CALL( cudaMalloc( (void**)&gpuMat, w*h* sizeof(float) ) );
CUDA_SAFE_CALL( cudaMalloc( (void**)&gpuVec, w * sizeof(float) ) );
CUDA_SAFE_CALL( cudaMalloc( (void**)&gpuResVec, h * sizeof(float) )
);
CUT_CHECK_ERROR("allocation failed\n");
// upload M and x
CUDA_SAFE_CALL( cudaMemcpy( gpuMat, hostMat, w*h * sizeof(float),
    cudaMemcpyHostToDevice) );
CUDA_SAFE_CALL( cudaMemcpy( gpuVec, hostVec, w * sizeof(float),
    cudaMemcpyHostToDevice) );
// compute the block and grid dimensions
dim3 threadIdx( MAX_THREADS, 1 );
dim3 blockIdx( h / MAX_THREADS + 1, 1, 1);
vecMat1<<< blockIdx, threadIdx >>>( gpuResVec, gpuMat, gpuVec,
    w,h);
CUT_CHECK_ERROR("vecMat filter failed\n");
CUDA_SAFE_CALL( cudaThreadSynchronize() );
// download result y
CUDA_SAFE_CALL( cudaMemcpy( hostResVec, gpuResVec, h * sizeof(float),
    cudaMemcpyDeviceToHost) );
cudaFree( gpuMat ); cudaFree( gpuVec ); cudaFree( gpuResVec );
```

VecMat Kernel – Version 1

```
__global__ void vecMat1(float * _dst, const float* _mat,
const float* _v, int _w, int _h ) {

    // row index the thread is operating on
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < _h) {
        float res = 0.;

        // dot product of one line
        for (int j = 0; j < _w; ++j) {
            res += _mat[i*_w + j] * _v[j];
        }
        // write result to global memory
        _dst[i] = res;
    }
}
```

Why is this slow?

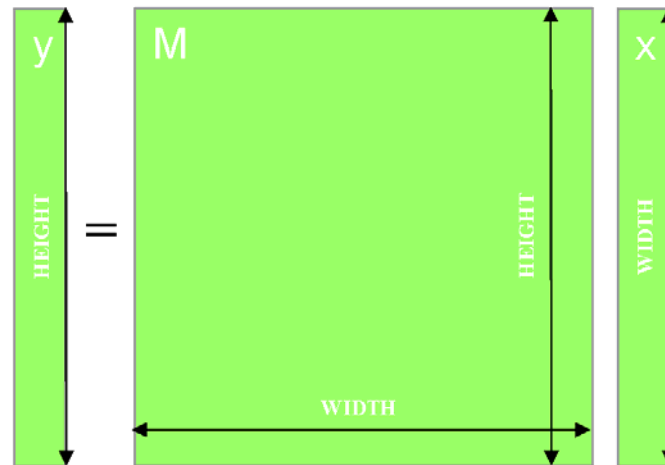
- **Problem is bandwidth limited (read)**
- **Each thread is accessing**
 - w elements of M
 - w elements of x**from global memory**
- **Total bandwidth: $2 * w * h$**
- **But all threads are accessing the same elements of x**
- **Load x into shared memory and reuse!**

Vector-Matrix Multiplication - using shared memory -

$$y = Mx$$

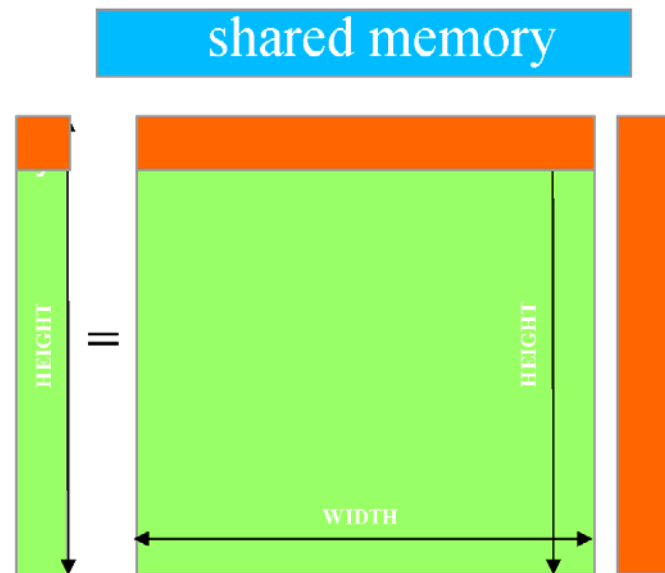
Vector-Matrix Multiplication V2

- Every thread uploads a couple of elements to shared memory
- Every thread computes the dot product between one line of M and x



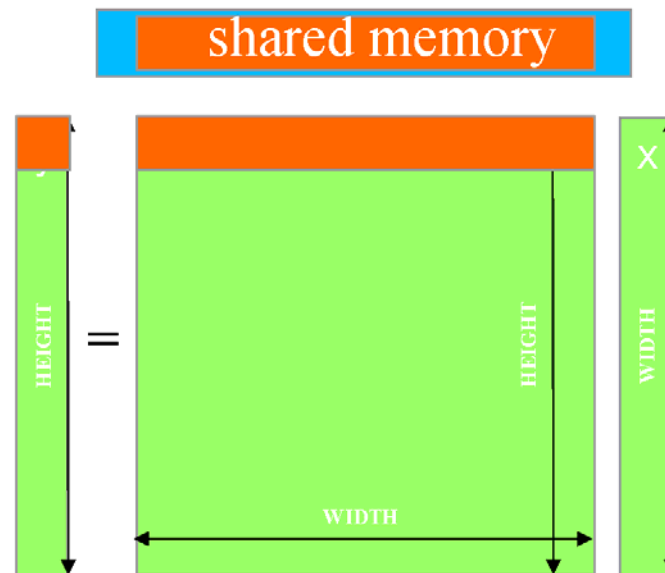
Vector-Matrix Multiplication V2

- Every thread uploads a couple of elements to shared memory
- Every thread computes the dot product between one line of M and x



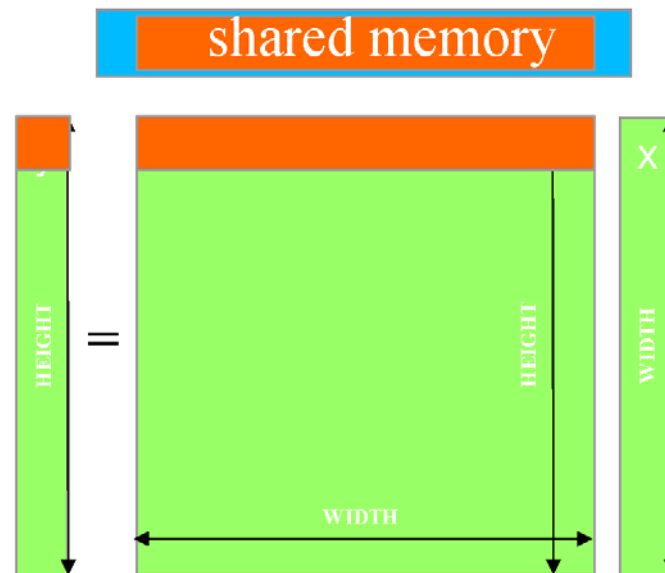
Vector-Matrix Multiplication V2

- Every thread uploads a couple of elements to shared memory
- Every thread computes the dot product between one line of M and x



Vector-Matrix Multiplication V2

- Every thread uploads a couple of elements to shared memory
- Every thread computes the dot product between one line of M and x



Setup – Version 2

```
... // allocate memory
float* gpuMat, gpuVec, gpuResVec;
CUDA_SAFE_CALL( cudaMalloc( (void**)&gpuMat, w*h* sizeof(float) ) );
CUDA_SAFE_CALL( cudaMalloc( (void**)&gpuVec, w * sizeof(float) ) );
CUDA_SAFE_CALL( cudaMalloc( (void**)&gpuResVec, h * sizeof(float) )
);
CUT_CHECK_ERROR("allocation failed\n");
// upload M and x
CUDA_SAFE_CALL( cudaMemcpy( gpuMat, hostMat, w*h * sizeof(float),
    cudaMemcpyHostToDevice) );
CUDA_SAFE_CALL( cudaMemcpy( gpuVec, hostVec, w * sizeof(float),
    cudaMemcpyHostToDevice) );
// compute the block and grid dimensions
dim3 threadBlock( MAX_THREADS, 1 );
dim3 blockGrid( h / MAX_THREADS + 1, 1, 1);
vecMat2<<< blockGrid, threadBlock, w * sizeof(float) >>>( gpuResVec,
    gpuMat, gpuVec, w,h, w / MAX_THREADS);
CUT_CHECK_ERROR("vecMat filter failed\n");
CUDA_SAFE_CALL( cudaThreadSynchronize() );
// download result y
CUDA_SAFE_CALL( cudaMemcpy( hostResVec, gpuResVec, h * sizeof(float),
    cudaMemcpyDeviceToHost) );
cudaFree( gpuMat ); cudaFree( gpuVec ); cudaFree( gpuResVec );
```

VecMat Kernel – Version 2

```
__global__ void vecMat2(float *_dst, const float* _mat, const float*
    _v, int _w, int _h, int nIter) {
extern __shared__ float vec[];

int i = blockIdx.x * blockDim.x + threadIdx.x;
float res = 0.; int vOffs = 0;

// load x into shared memory
for (int iter = 0; iter < nIter; ++iter, vOffs += blockDim.x) {
vec[vOffs + threadIdx.x] = _v[vOffs + threadIdx.x];
}
// make sure all threads have written their parts
__syncthreads();

// now compute the dot product again
// use elements of x loaded by other threads!
if (i < _h) {
    for (int j = 0; j < _w; ++j) {
        res += _mat[offs + j] * vec[j];
    }
    _dst[i] = res;
}}
```

VecMat Kernel – Version 2

```
__global__ void vecMat2(float * _dst, const float* _mat, const float*  
_v, int _w, int _h, int nIter) {
```

```
extern __shared__ float vec[];
```

```
int i = blockIdx.x * blockDim.x + threadIdx.x;  
float res = 0.; int vOffs = 0;
```

```
// load x into shared memory  
for (int iter = 0; iter < nIter; ++iter, vOffs += blockDim.x) {  
vec[vOffs + threadIdx.x] = _v[vOffs + threadIdx.x];  
}
```

```
// make sure all threads have written their parts  
__syncthreads();
```

```
// now compute the dot product again  
// use elements of x loaded by other threads!  
if (i < _h) {  
for (int j = 0; j < _w; ++j) {  
res += _mat[offs + j] * vec[j];  
}  
_dst[i] = res;  
}}
```

Memory Access Patterns



1. Global Memory Accesses

- Memory coalescing
- Cached memory access

Memory Layout of a Matrix in C

$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$
$M_{0,1}$	$M_{1,1}$	$M_{2,1}$	$M_{3,1}$
$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{3,2}$
$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$

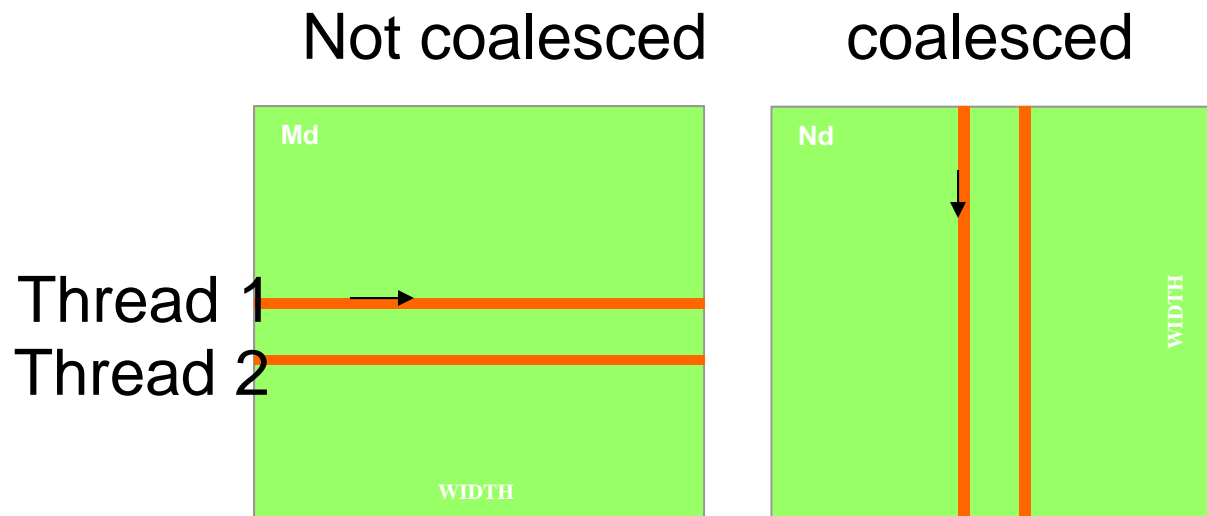
M



$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$	$M_{0,1}$	$M_{1,1}$	$M_{2,1}$	$M_{3,1}$	$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{3,2}$	$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$
-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------

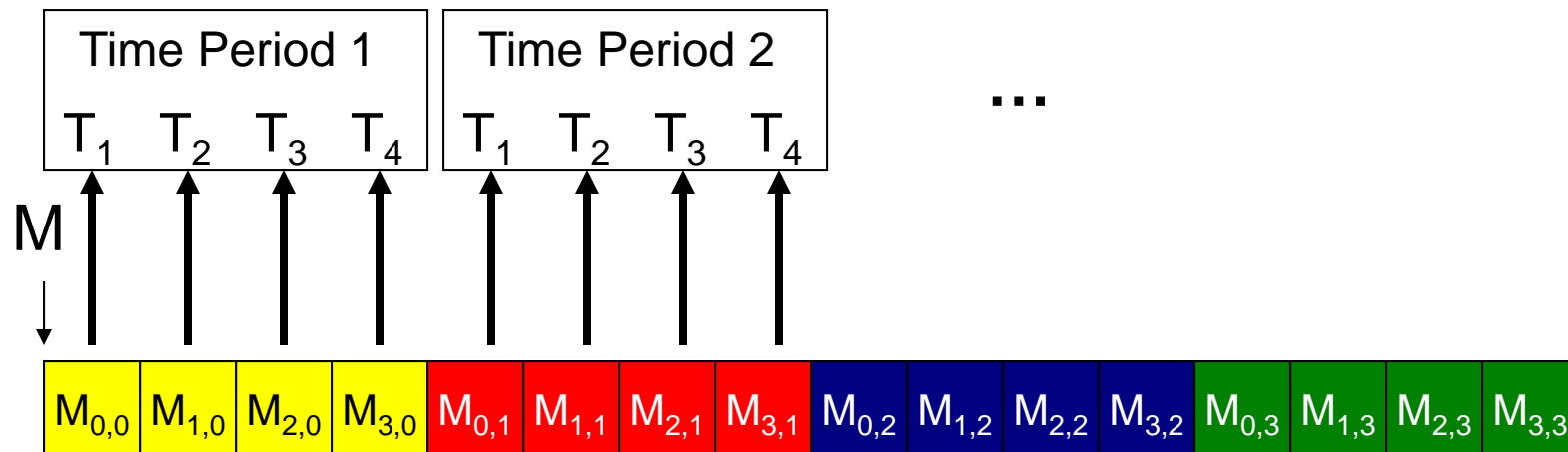
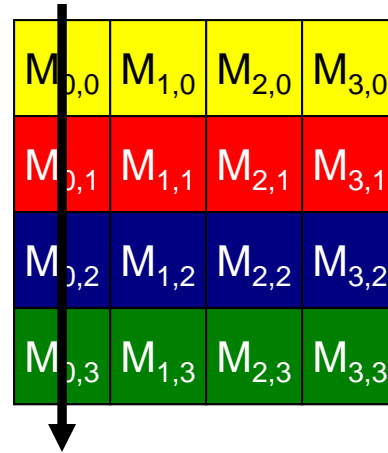
Memory Coalescing

- When accessing global memory, peak performance utilization occurs when all threads in a half warp (**full warp on Fermi**) access continuous memory locations.
- Requirements relaxed on ≥ 1.2 devices; L1 cache on Fermi!

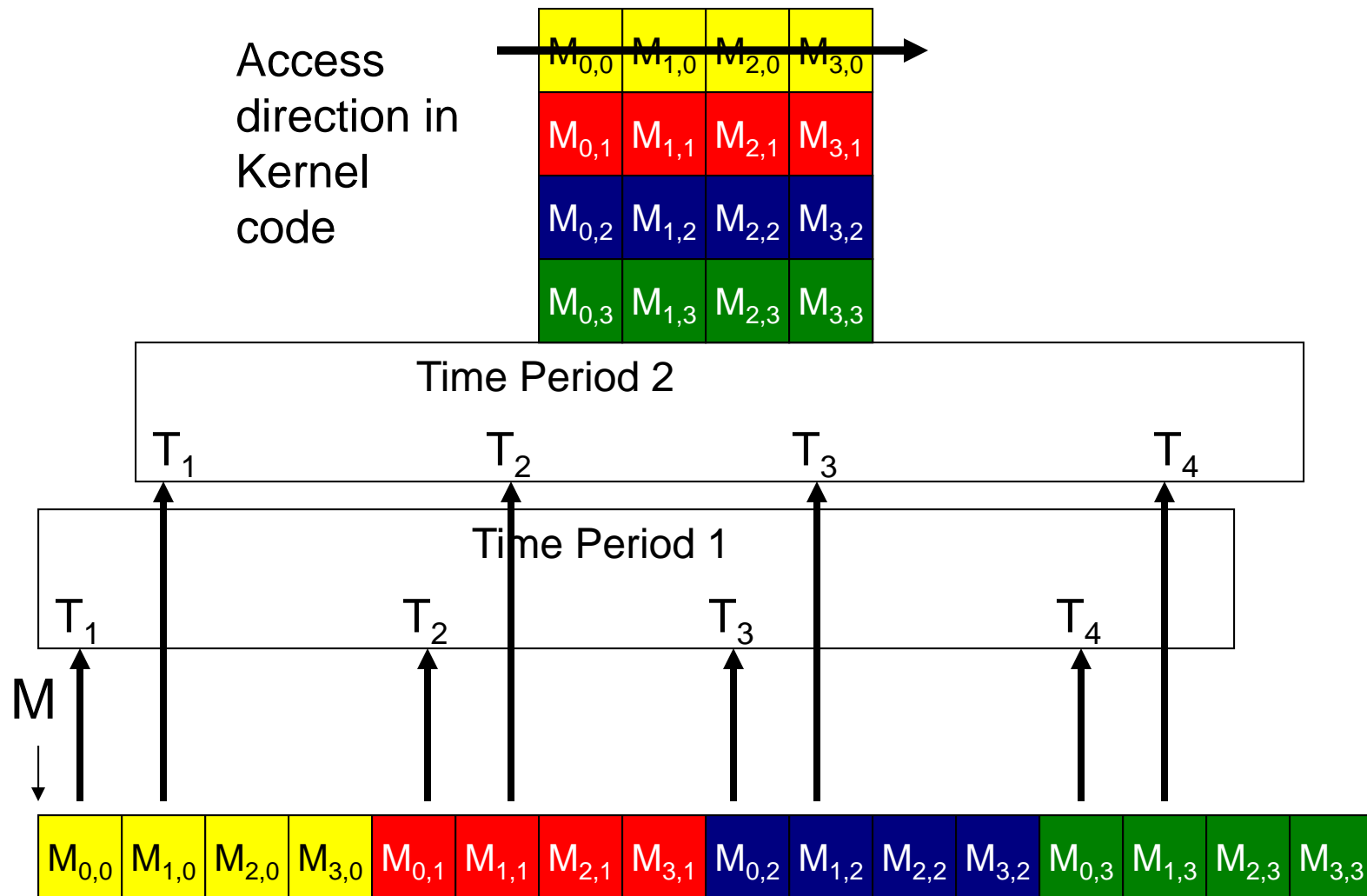


Memory Layout of a Matrix in C

Access
direction in
Kernel
code



Memory Layout of a Matrix in C



Memory Access Patterns

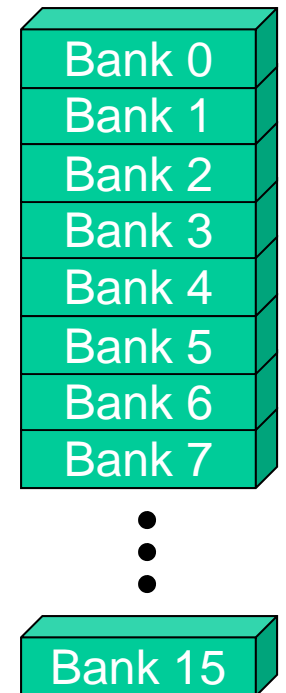


2. Shared Memory Accesses

- Banked memory access / bank conflicts

Parallel Memory Architecture

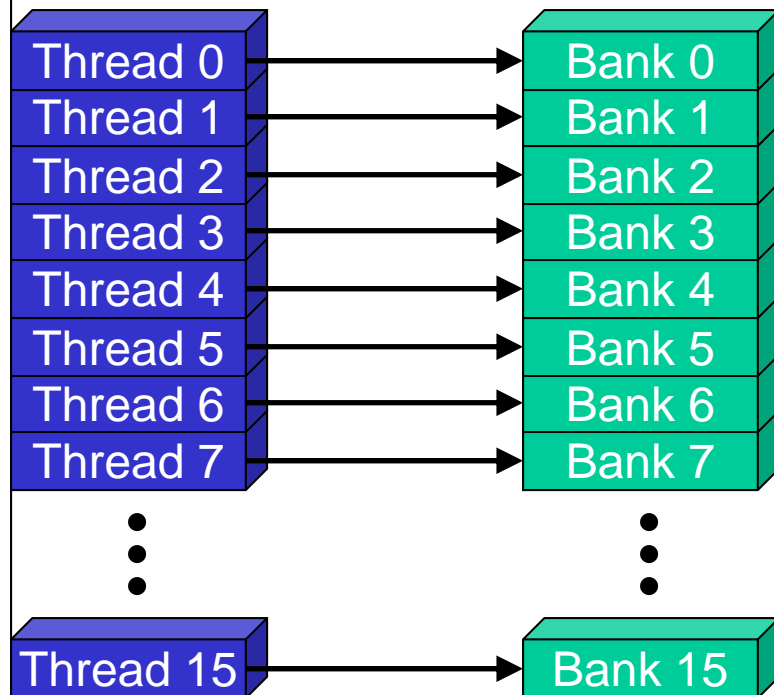
- In a parallel machine, many threads access memory
 - Therefore, memory is divided into **banks**
 - Essential to achieve high bandwidth
- Each bank can service one address per cycle
 - A memory can service as many simultaneous accesses as it has banks
- Multiple simultaneous accesses to a bank result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



Bank Addressing Examples

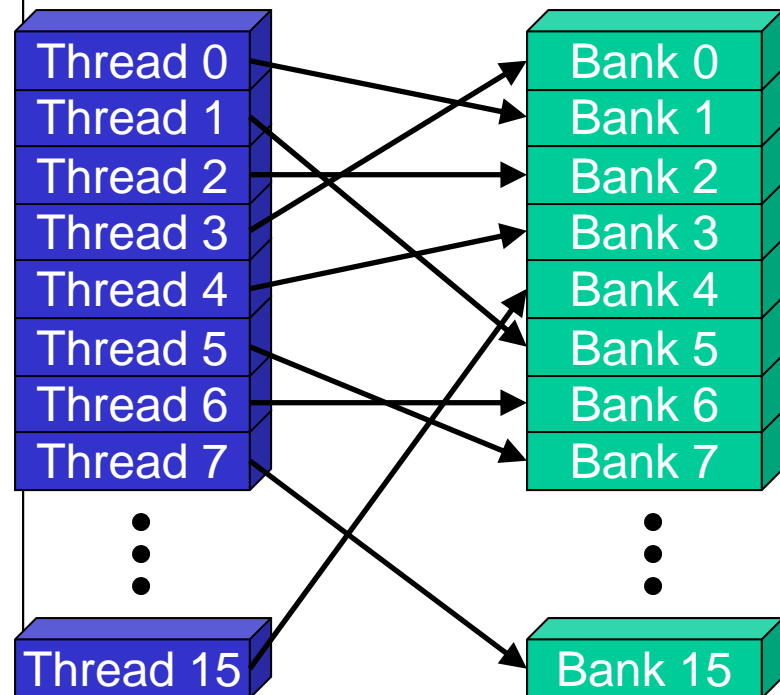
- No Bank Conflicts

- Linear addressing
stride == 1



- No Bank Conflicts

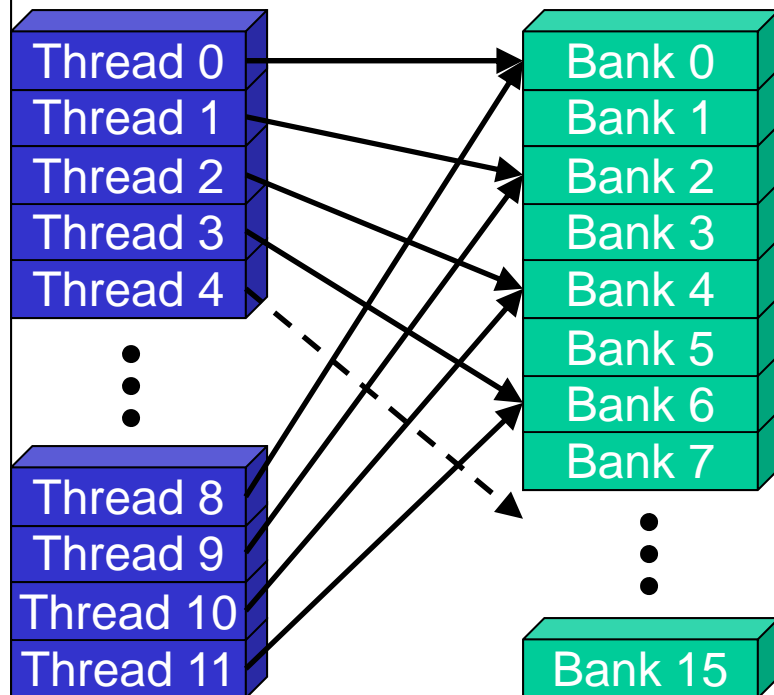
- Random 1:1 Permutation



Bank Addressing Examples

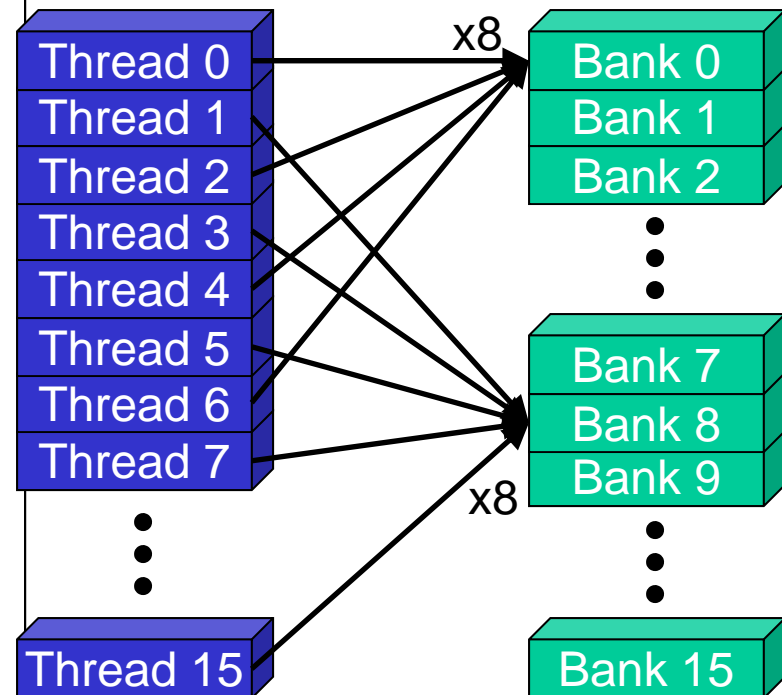
- 2-way Bank Conflicts

- Linear addressing
stride == 2



- 8-way Bank Conflicts

- Linear addressing
stride == 8



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
 - So bank = address % 16
 - Same as the size of a half-warp
 - No bank conflicts between different half-warps, only within a single half-warp

**Fermi has 32 banks,
considers full warps instead of half warps!**

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!

Thank you.

- Hendrik Lensch, Robert Strzodka