GPU Programming

(in Cuda)

Julian Gutierrez

NUCAR

Session 5

Synchronization

Synchronization

- Many applications require different means of synchronization.
  - Intra Block
  - Inter Block
  - CPU/GPU concurrent execution

Synchronization

- Many applications require different means of synchronization.
  - Intra Block
  - Inter Block
  - CPU/GPU concurrent execution

How can we do these?

Synchronization

- Many applications require different means of synchronization.
  - Intra Block
    - __syncthreads()
      - Must be reached by all threads from the block.
      - Ensures that for all threads in the block, the code preceding the instruction is executed before the instructions following it.
      - Implies a memory fence function as well.
    - Atomic operations (in shared memory)

Synchronization

- Many applications require different means of synchronization.
  - Intra Block
    - __syncthreads()
      - Must be reached by all threads from the block.
      - Ensur
        the in
      - Implie
    - Atomic o

What is a memory fence?

Synchronization

- Many applications require different means of synchronization.
  - Memory Fence
    - __threadfence_block()
      - Stalls current thread until all writes by this thread to shared and global memory are visible to other threads from the same block.
      - It does not synchronize the threads and it is not necessary for all threads to actually reach this instruction.
    - __threadfence()
      - Stalls current thread until all writes to shared and global memory are visible for all the threads in the block, and writes to global memory are visible to all other threads on the device.
      - Acts as a __threadfence_block + global memory write order is kept. Ensures other threads in the device can see the writes to global memory from the thread executing the function call, before those threads write anything to global memory after the call.

Synchronization

- Many applications require different means of synchronization.
  - Inter Block
    - No explicit way of doing this
      - Terminate kernel at synchronization point, and then launch new kernel to continue after.
    - Atomic operations (in global memory, should be avoided)

Synchronization

- Many applications require different means of synchronization.
  - CPU/GPU concurrent execution
    - Copy data back and forth

Race Condition

- An undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time, but because of the nature of the device or system, the operations must be done in the proper sequence to be done correctly.

Race Condition

- An undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time, but because of the nature of the device or system, the operations must be done in the proper s y.

How can we avoid this on a GPU?

Atomic instructions

- **Atomic instructions** are special hardware **instructions** that perform an operation on one or more memory locations atomically (indivisible or uninterruptible).
- An **atomic** operation either succeeds or fails in its entirety, regardless of what **instructions** are being executed by other threads.

Atomic instructions

- Use of atomic operations
  - Collaboration
    - Atomics on an array that will be the output of the kernel
      - Histogram
  - Synchronization
    - Atomics on memory locations that are used for synchronization or coordination
      - Counters, locks, flags…
- CUDA provides atomic functions on shared memory and global memory

Atomic instructions

- Arithmetic functions (int, uint, ull, float
  - Add, sub, max, min, exch, inc, dec, CAS
- Bitwise functions
  - And, or, xor

```
CUDA: int atomicAdd(int*, int);
PTX: atom.shared.add.u32 &r25, [%rd14],1;
SASS:
```

GT200, Fermi, Kepler

```
/*00a0*/  LDSLK P0, R9, [R8];
/*00a8*/  @P0 IADD R10, R9, R7;
/*00b0*/  @P0 STSCUL P1, [R8], R10;
/*00b8*/  @!P1 BRA 0xa0;
```

Maxwell

```
/*01f8*/  ATOMS.ADD RZ, [R7], R11;
```

Native atomic operations for 32-bit integer, and 32-bit and 64-bit atomicCAS

Atomic instructions

- Arithmetic functions (int, uint, ull, float
  - Add, sub, max, min, exch, inc, dec, CAS
- Bitwise functions
  - And, or, xor

```
CUDA: int atomicAdd(int*, int);
PTX: atom.shared.add.u32 &r25, [%rd14],1;
SASS:
```

GT200, Fermi, Kepler

```
/*00a0*/  LDSLK P0, R9, [R8];
/*00a8*/  @P0 IADD R10, R9, R7;
/*00b0*/  @P0 STSCUL P1, [R8], R10;
/*00b8*/  @!P1 BRA 0xa0;
```

Maxwell

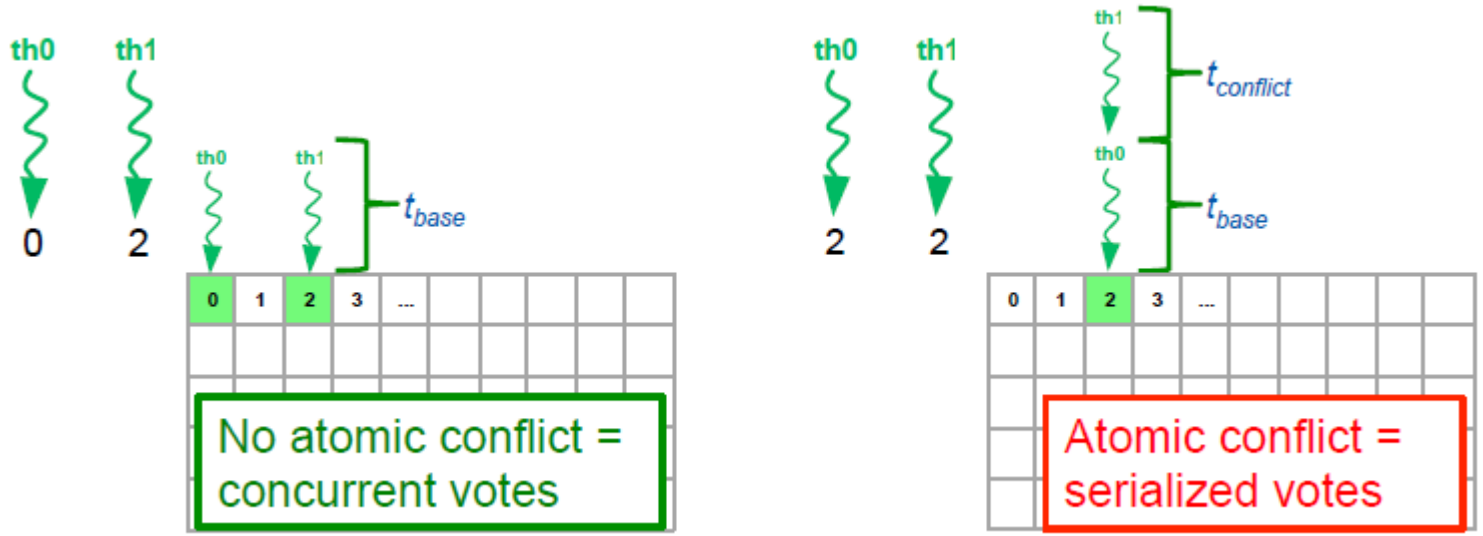
```
/*01f8*/  ATOMS.ADD RZ, [R7], R11;
```

Native atomic operations for 32-bit integer, and 32-bit and 64-bit atomicCAS

Lock/update/unlock vs native atomic

Atomic instructions

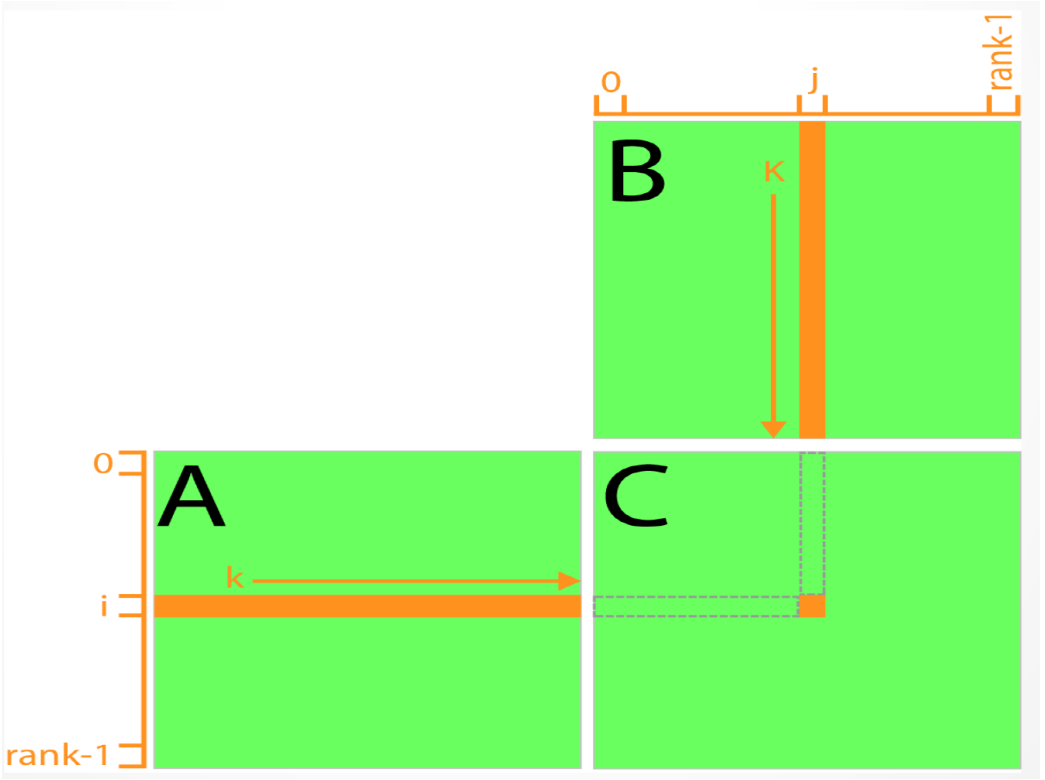
- Atomic conflict degree
  - Intra-warp conflict degree from 1 to 32



Considerations when using Memory

Matrix Multiplication

- Launch 2D grid
- Compute element-wise output
  - C[i][j] = sum (A[i][k] * B [k][j])



Matrix Multiplication

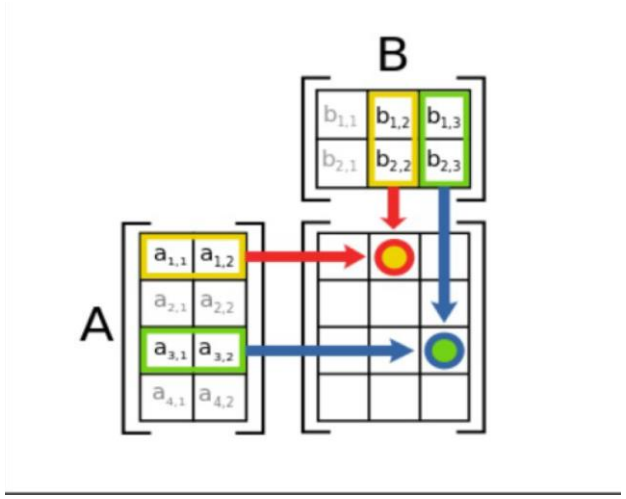
- Launch 2D grid
- Compute element-wise output
  - C[i][j] = sum (A[i][k] * B [k][j])

```
__global__ void MatrixMultiplyKernel_GlobalMem( float* C, const float* A, const float* B, unsigned int rank )
{
    // Compute the row index
    unsigned int i = ( blockDim.y * blockIdx.y ) + threadIdx.y;
    // Compute the column index
    unsigned int j = ( blockDim.x * blockIdx.x ) + threadIdx.x;

    unsigned int index = ( i * rank ) + j;
    float sum = 0.0f;
    for ( unsigned int k = 0; k < rank; ++k )
    {
        sum += A[i * rank + k] * B[k * rank + j];
    }
    C[index] = sum;
}
```

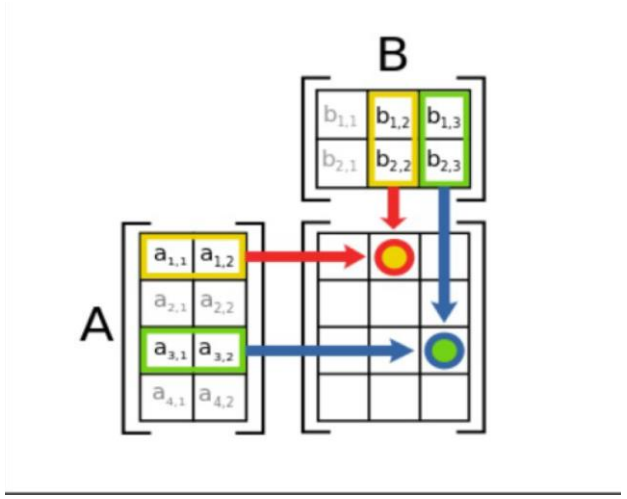
Matrix Multiplication

- Problem with naïve version

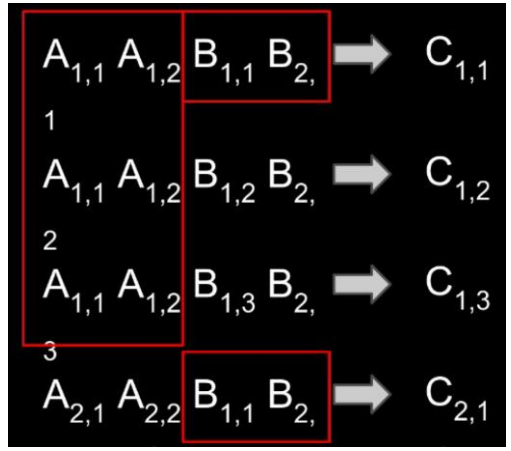


Matrix Multiplication

- Problem with naïve version

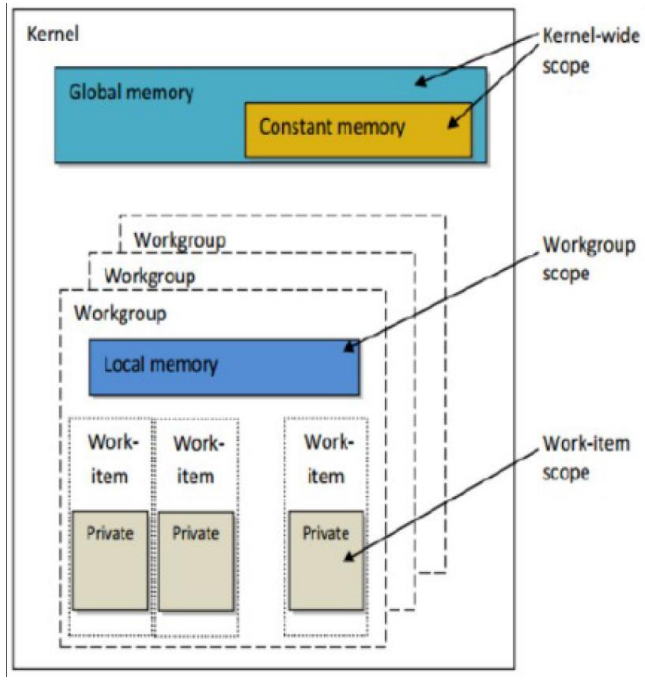


- We could have better data reuse



Matrix Multiplication

- Solution



- Use shared memory for data reuse
- Chunk the matrix into sub-matrices for parallel computation and good data locality

Shared Memory (REMEMBER)

- On-chip memory
- Per SM: Combination of L1 cache and shared memory
  - 16 KB + 48 KB shared memory + L1
  - 48 KB + 16 KB shared memory + L1
- L2 cache shared by all SMs

Shared Memory (REMEMBER)

- On-chip memory
- Per SM: Combination of L1 cache and shared memory
  - 16 KB
  - 48 KB
- L2 cac

What Happens if you allocate all of the Shared memory for the block ?

Shared Memory (REMEMBER)

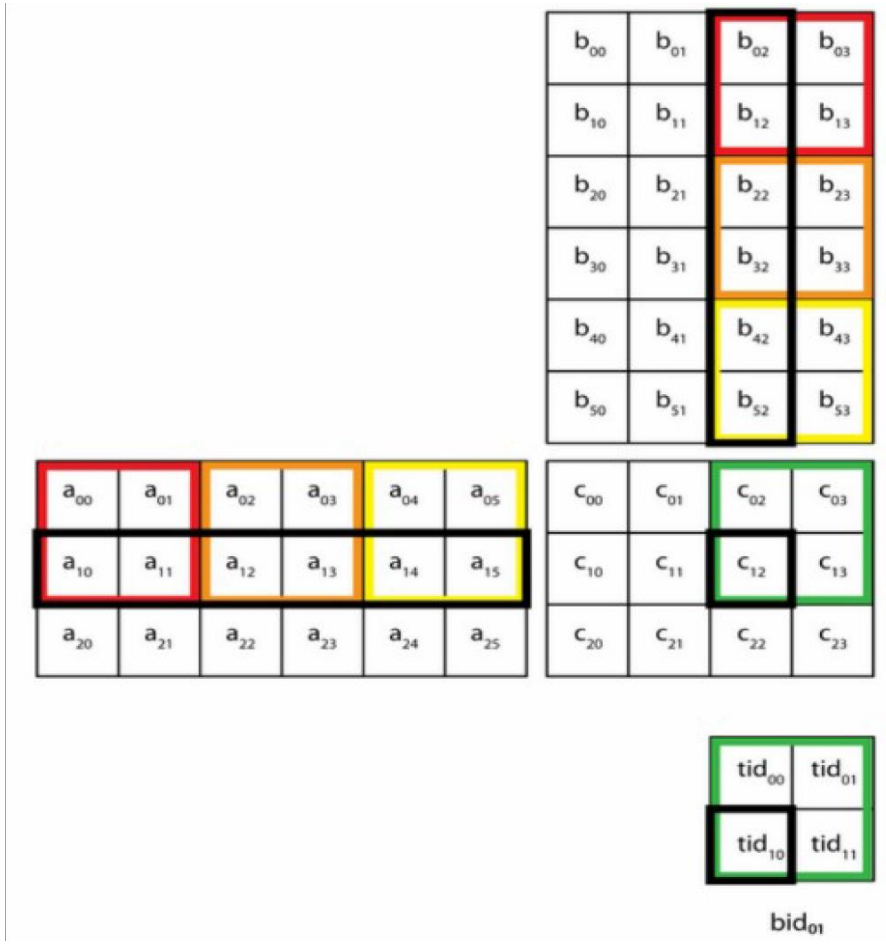
- On-chip memory
- Per SM
  shared
  - 16 KB
  - 48 KB
- L2 cac

You can only have 1 block running at a time in your SM (max use of resources).

This could happen when you have multiple blocks assigned to an SM and your blocks have few threads but take up all the shared memory. Only one of the blocks can run at a time.

Matrix Multiplication (Shared Memory)

- Scheme
  - The actual implementation is quite extensive



main

```
int main(int argC, char** argV) {

        //
        // Setup
        ////////////////////

        // Time Variables
        float time;
        cudaEvent_t start, stop;
        cudaEventCreate (&start);
        cudaEventCreate (&stop);

        // Matrices
        float *a, *b;
        float *c_cpu, *c_gpu_basic, *c_gpu_improved;

        //Setting matrix parameters.
        int row = ROW;
        int col = COL;
        int   k = COL;
        int sum = 0;

        // Process input arguments (if specified)
        switch (argC) {
                case 2: {
                        row = atoi(argV[1]);
                        col = row;
                          k = col;
                        break;
                }
                case 3: {
                        row = atoi(argV[1]);
                        col = atoi(argV[2]);
                          k = col;
                        break;
                }
                default: {
                        //Nothing
                }
        }
```

main

```
//Setting host memory space.
a                = (float *) malloc(row*k*sizeof(float));
b                = (float *) malloc(k*col*sizeof(float));
c_cpu            = (float *) malloc(row*col*sizeof(float));
c_gpu_basic      = (float *) malloc(row*col*sizeof(float));
c_gpu_improved   = (float *) malloc(row*col*sizeof(float));

//Initializing [A] and [B] with random values from 1 to 10, and C to 0
printf ("Initializing Matricies, could take some time...\n");
for(int i = 0 ; i < row ; i++ ){
        for(int j = 0 ; j < k ; j++ ){
                a[i*k+j] = rand()%10;
        }
}
for(int i = 0 ; i < k ; i++ ){
        for(int j = 0 ; j < col ; j++ ){
                b[i*col+j] = rand()%10;
        }
}
for(int i = 0 ; i < k ; i++ ){
        for(int j = 0 ; j < col ; j++ ){
                c_cpu            [i*col+j] = 0;
                c_gpu_basic      [i*col+j] = 0;
                c_gpu_improved   [i*col+j] = 0;
        }
}
```

main

```
//
// CPU Calculation
//////////////////

printf("Running sequential job.\n");
cudaEventRecord(start,0);
for(int i = 0 ; i < row ; i++ ){
        for(int j = 0 ; j < col ; j++ ){
                sum = 0;
                for(int w = 0 ; w < k ; w++ ){
                        sum += a[i*k+w] * b[w*col+j];
                }
                c_cpu[i*col+j] = sum;
        }
}
cudaEventRecord(stop,0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time, start, stop);
printf("\tSequential Job Time: %.2f ms\n", time);
```

main

```
//
// Basic GPU Calculation
//////////////////////////

printf("Running Basic parallel job.\n");

cudaEventRecord(start,0);
MM_Basic(a, b, c_gpu_basic, row, col, k);
cudaEventRecord(stop,0);
cudaEventSynchronize(stop);

cudaEventElapsedTime(&time, start, stop);
printf("\tBasic Parallel Job Time: %.2f ms\n", time);

// Compares matrices to make sure answer is correct, initializes c for next kernel.
bool error = false;
for(int i = 0 ; i < k ; i++ ){
        for(int j = 0 ; j < col ; j++ ){
                if (c_cpu[i*col+j] != c_gpu_basic[i*col+j]) {
                        printf("\tError: Starting at [%d][%d]\n", i, j);
                        error = true;
                }
                if (error) break;
        }
        if (error) break;
}
if (!error) printf("\tNo errors found.\n");
```

Function

```
void MM_Basic(float *a, float *b, float *c, int row, int col, int k) {

        cudaEvent_t kernelstart, kernelstop;
        float time;
        cudaEventCreate (&kernelstart);
        cudaEventCreate (&kernelstop);

        int sizeA = row*k*sizeof(float);
        int sizeB = k*col*sizeof(float);
        int sizeC = row*col*sizeof(float);
        float *devA, *devB, *devC;

        cudaMalloc((void**)&devA, sizeA);
        cudaMalloc((void**)&devB, sizeB);
        cudaMalloc((void**)&devC, sizeC);

        cudaMemcpy(devA, a, sizeA, cudaMemcpyHostToDevice);
        cudaMemcpy(devB, b, sizeB, cudaMemcpyHostToDevice);

        dim3 dimBlock(16, 16, 1);
        dim3 dimGrid((COL+dimBlock.x-1)/dimBlock.x, (ROW+dimBlock.y-1)/dimBlock.y, 1);

        cudaEventRecord(kernelstart,0);
        MM_Basic_kernel<<<dimGrid, dimBlock>>>(devA, devB, devC, row, col, k);
        cudaEventRecord(kernelstop,0);
        cudaEventSynchronize(kernelstop);

        cudaEventElapsedTime(&time, kernelstart, kernelstop);
        printf("\tKernel Job Time: %.2f ms\n", time);

        cudaMemcpy(c, devC, sizeC, cudaMemcpyDeviceToHost);

        //Freeing device matrices.
        cudaFree(devA); cudaFree(devB); cudaFree(devC);
}
```

kernel

```
__global__ void MM_Basic_kernel( float *devA, float *devB, float *devC, int row, int col, int k) {
        int txID = blockIdx.x * blockDim.x + threadIdx.x;
        int tyID = blockIdx.y * blockDim.y + threadIdx.y;

        if ((txID < col) && (tyID < row)) {
                float Pvalue = 0;
                for(int w = 0 ; w < k ; w++) {
                        Pvalue += devA[tyID*k+w] * devB[w*k+txID];
                }
                devC[tyID*k+txID] = Pvalue;
        }
}
```

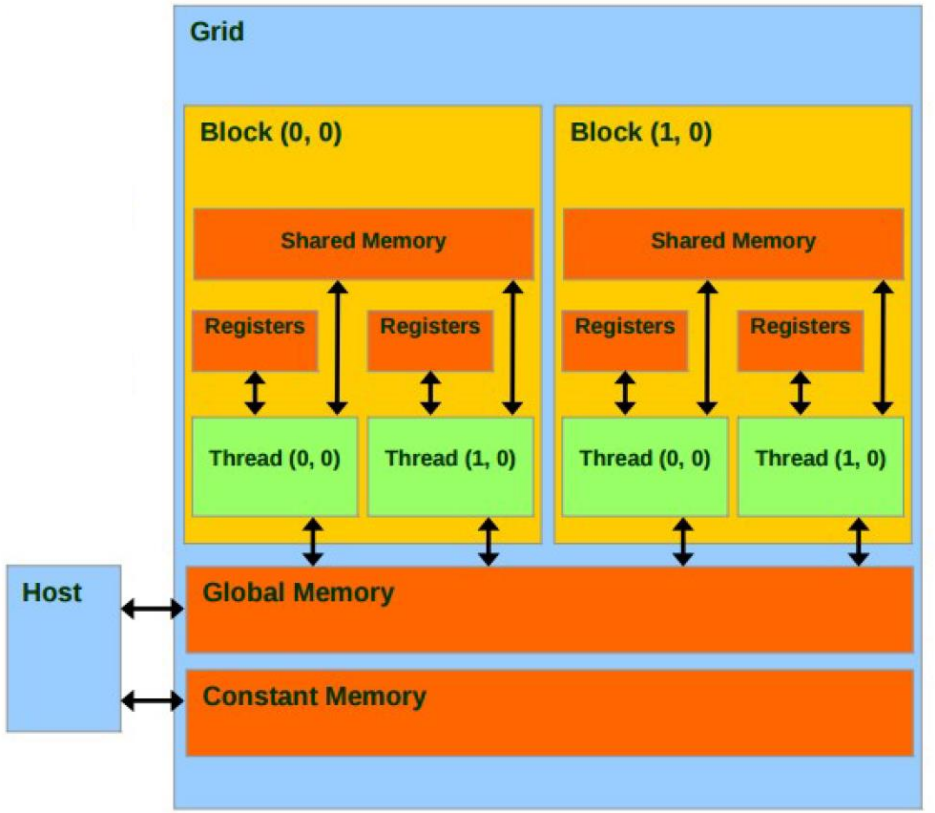
Improved Kernel

- Open file

How to Properly use Memory

CUDA Memory Model

- Each thread can:
  - Read/write per-thread **registers**
  - Read/write per-thread **local memory**
  - Read/write per-block **shared memory**
  - Read/write per-grid **global memory**
  - Read/only per-grid **constant memory**



CUDA Memory Model

- Type Qualifier table
- Notes:
  - __device__ not required for __local__, __shared__, or __constant__
  - Automatic variables without any qualifier reside in a register
    - Except arrays that reside in local memory
    - Or not enough registers available for automatic variables

Variable declaration	Memory	Scope	Lifetime
Int LocalVar;	Register	Thread	Thread
Int LocalArray[10];	Local	Thread	Thread
[__device__] __shared__ int SharedVar;	Shared	Block	Block
__device__ int GlobalVar;	Global	Grid	Application
[__device__] __constant__ int ConstantVar;	Constant	Grid	Application

CUDA Memory Model

- Type Qualifier table
- Notes:
  - Scalar variables reside in on-chip registers (fast)
  - Shared variables reside in on-chip memory (fast)
  - Local arrays and global variables reside in off-chip memory (slow)
  - Constants reside in cached off-chip memory

Variable declaration	Memory	Scope	Lifetime
Int LocalVar;	Register	Thread	Thread
Int LocalArray[10];	Local	Thread	Thread
[__device__] __shared__ int SharedVar;	Shared	Block	Block
__device__ int GlobalVar;	Global	Grid	Application
[__device__] __constant__ int ConstantVar;	Constant	Grid	Application

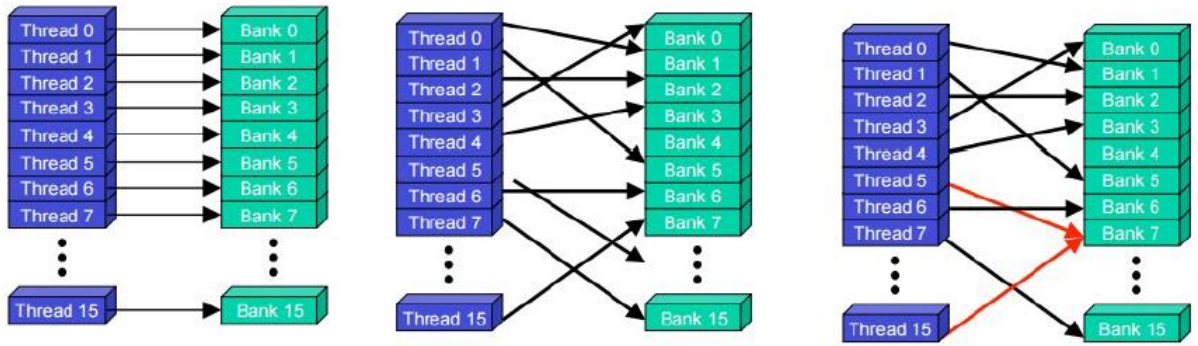
Shared Memory Structure

- Why is Shared memory so fast?

Shared Memory Structure

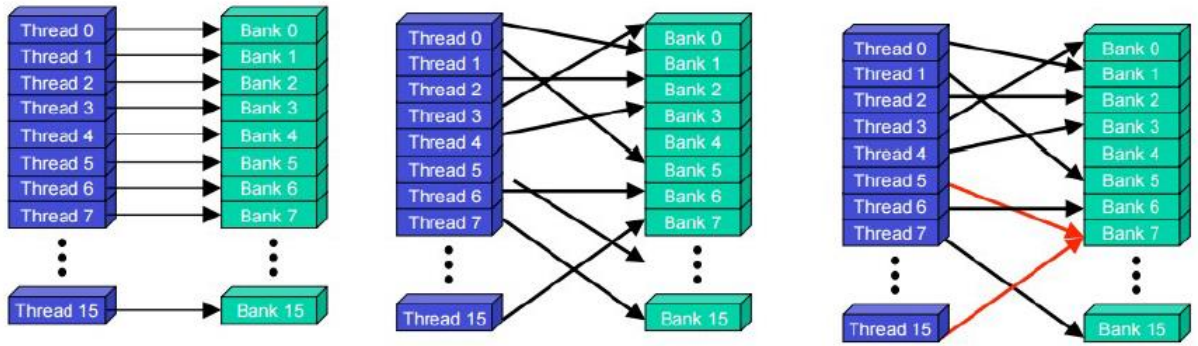
- Shared memory is divided into banks.

Shared Memory Banks



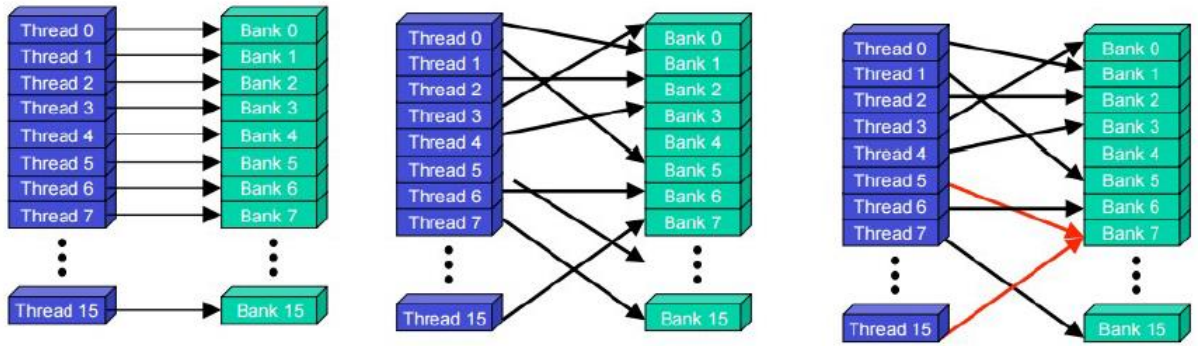
Shared Memory Structure

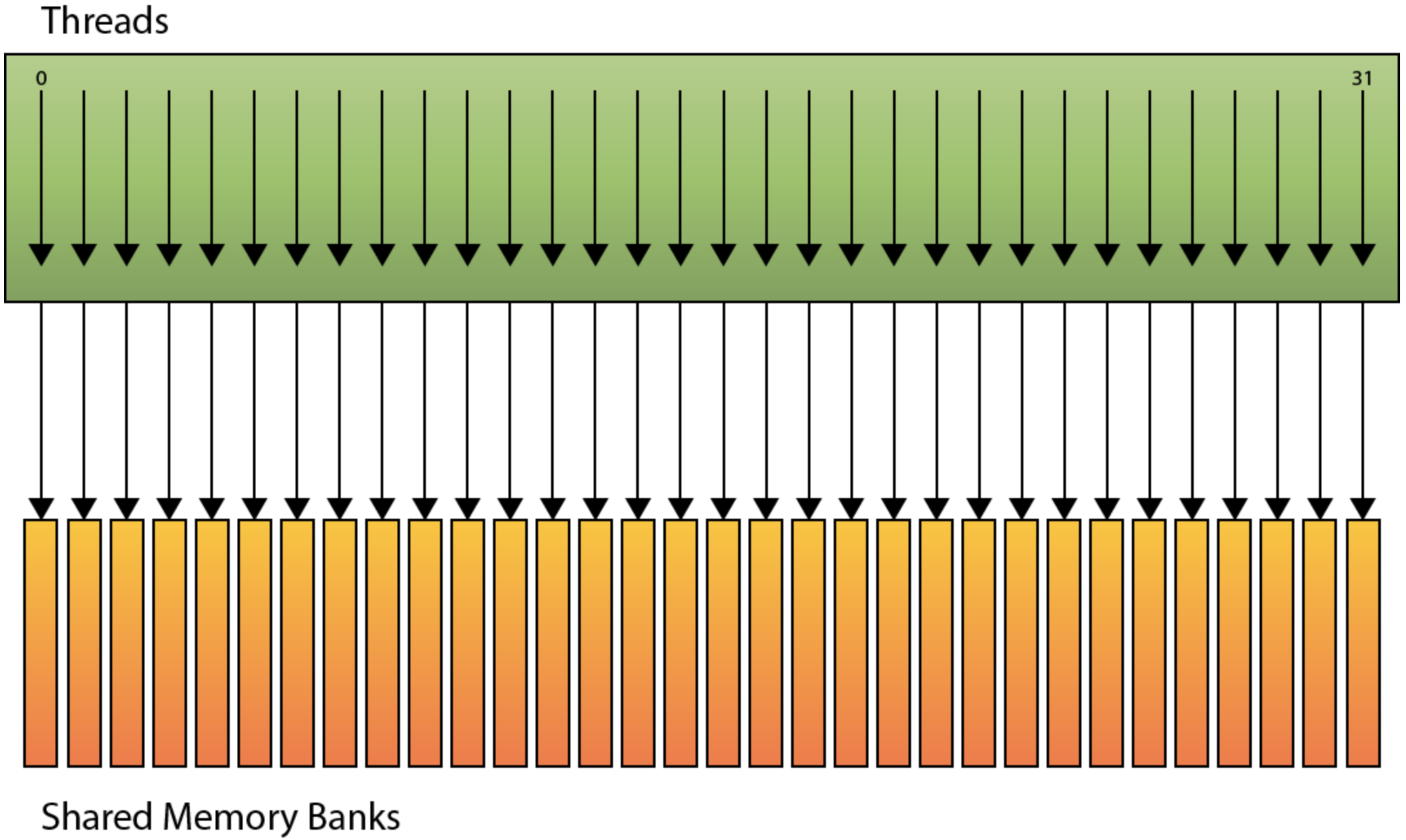
- Successive 32 bit words are assigned to successive banks
- For devices of compute capability 2.x (fermi)
  - Number of banks = 32
  - Bandwidth is 32 bits per bank per 2 clock cycles
  - Shared memory request for a warp is not split
  - Increased susceptibility to conflicts
  - But no conflicts if access to bytes in same 32 bit word

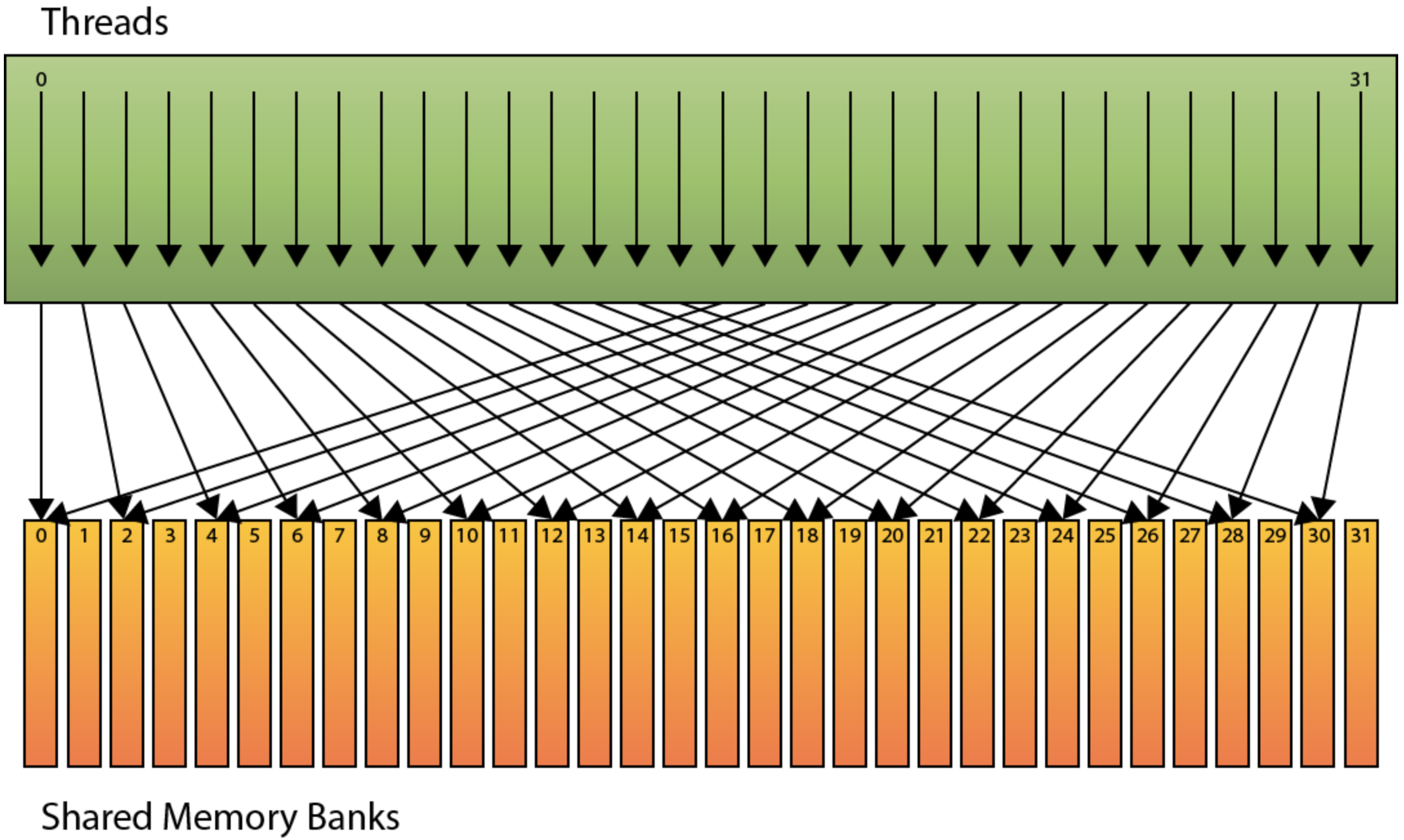


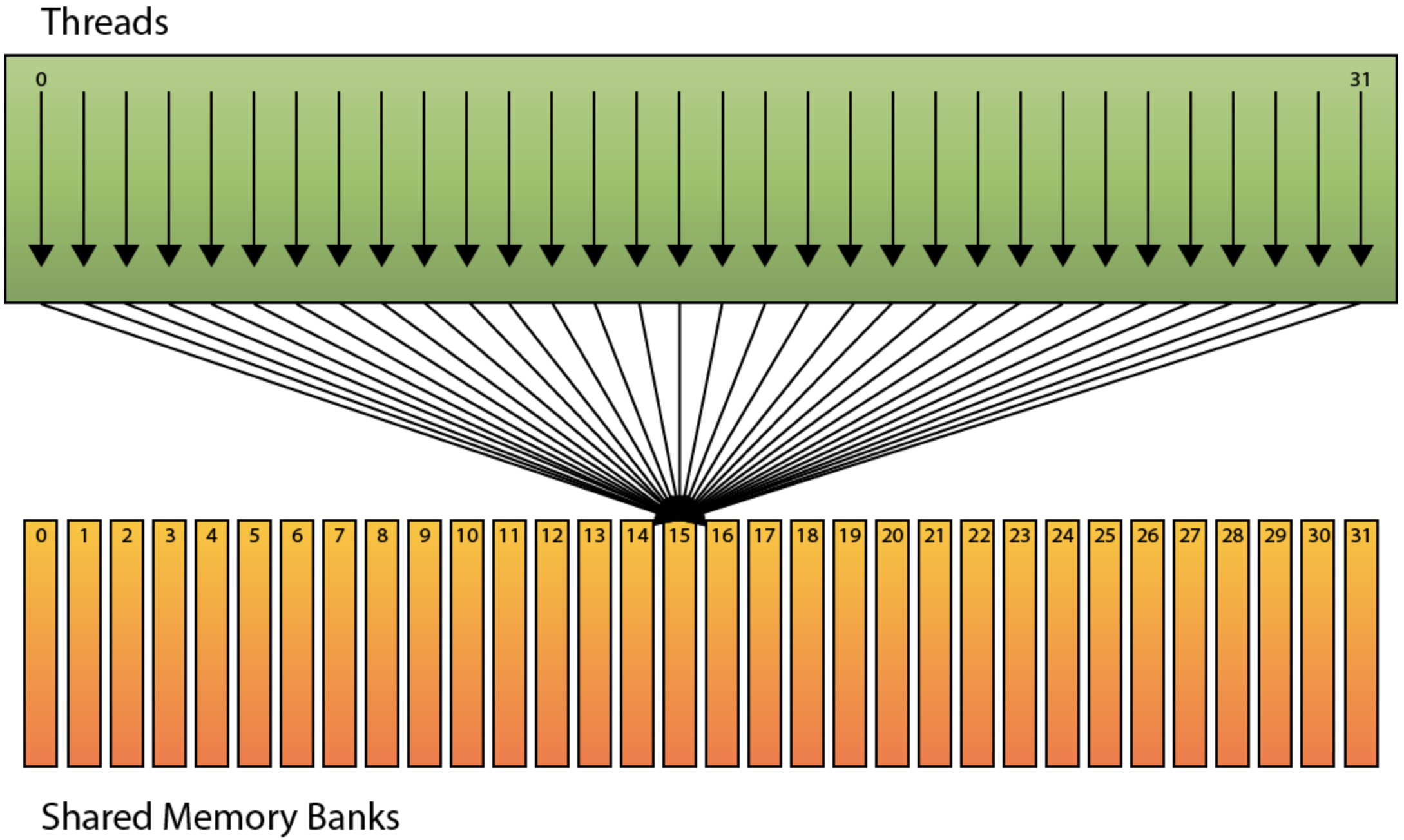
Bank Conflicts in Shared Memory

- Load/store of a addresses spanning n distinct memory banks can be serviced simultaneously, effective BW = n*a single banks
- Each bank can service 1 address / cycle (bcast too)
- Access to shared memory is fast unless..
  - 2 or more instructions in a warp access different banks: we have a conflict
  - Exception: if all access the same bank: broadcast



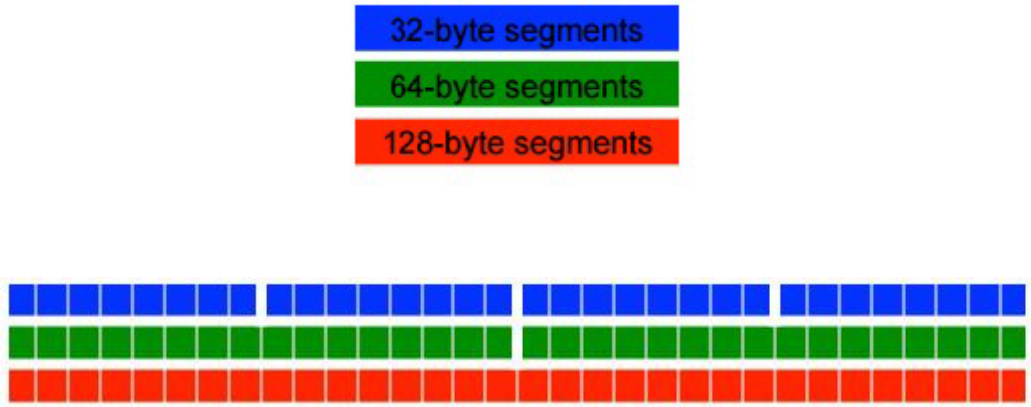

Threads
0
31
Shared Memory Banks


Threads
0
31
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Shared Memory Banks


Threads
0
31
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Shared Memory Banks

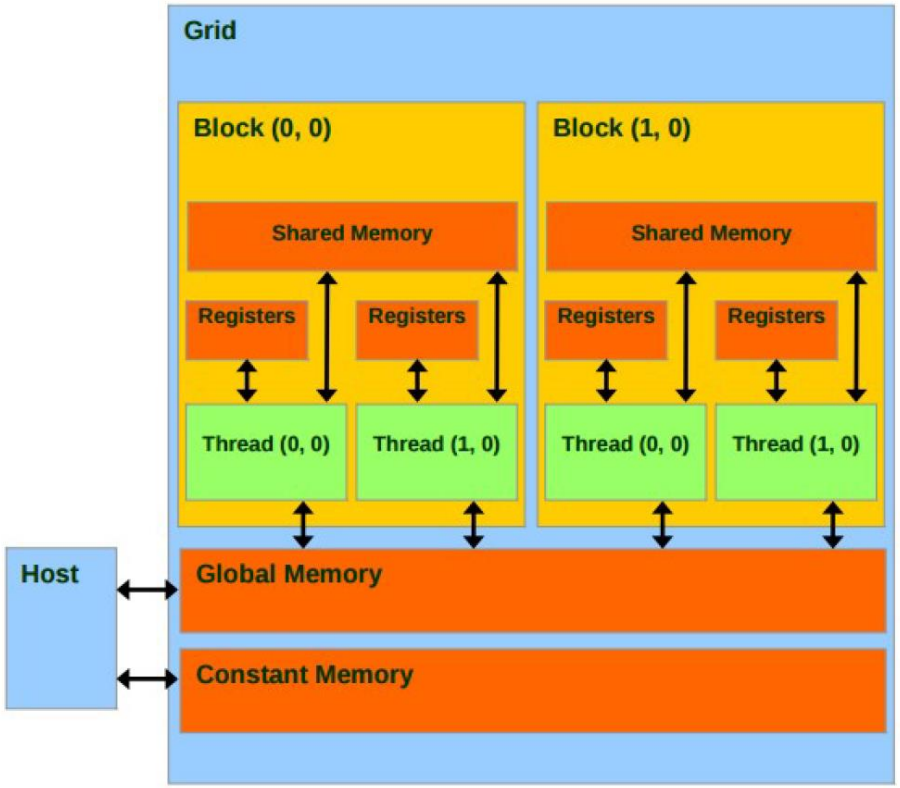
Global Memory

- Global memory accesses are in units of 32, 64, 128 B
- Consecutive addresses read quickly.
- Certain non-sequential access patterns to global memory degrade performance
- Accesses organized by half warps (16 threads) can be done in one or two transactions, under certain conditions (32, 64 and 128 byte segments)



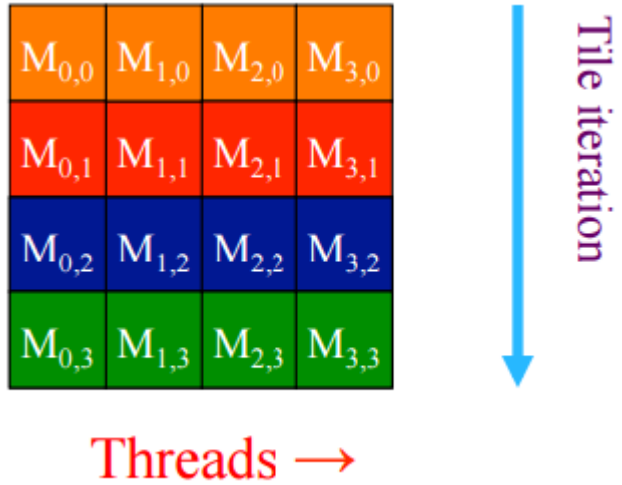
Global Memory

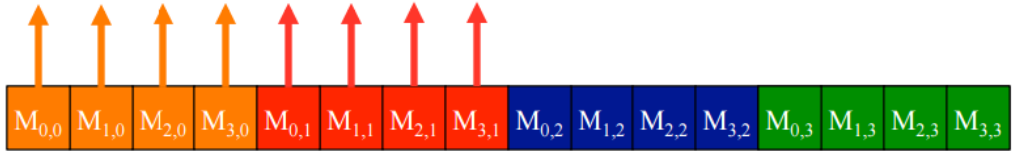
- If accessed word > 4 bytes, warp's memory request is split into separate, independently issued 128 byte memory requests



Global memory

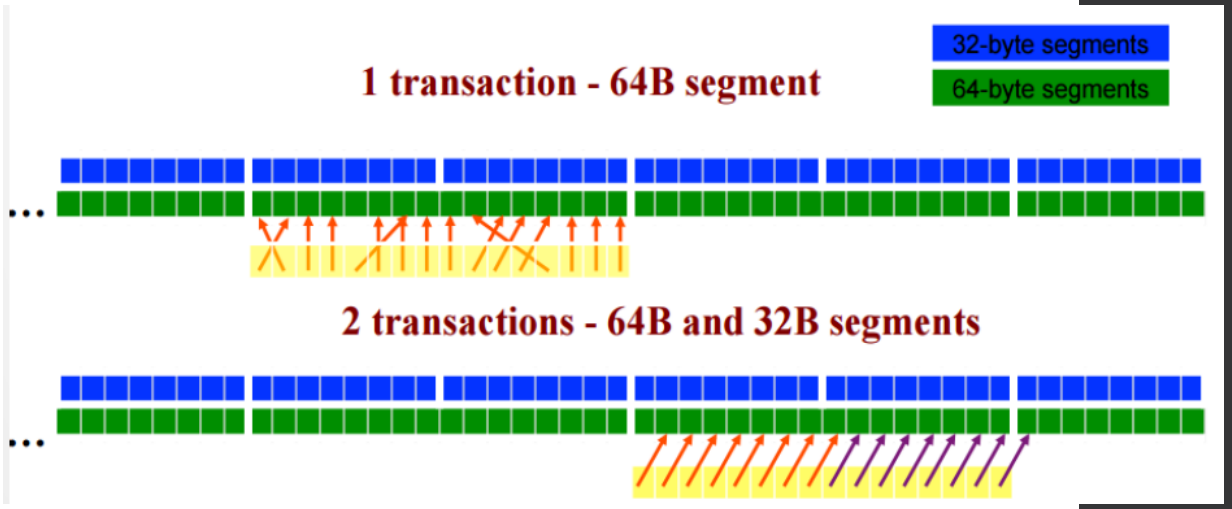
- Simplest: addresses are contiguous across threads
- Accesses organized by half warps (16 threads)





Memory coalescing (compute capability >= 1.2)

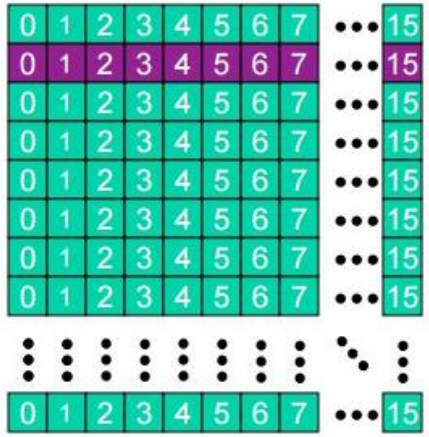
- Find the segment containing the address request of the lowest numbered active thread
- Find all other active threads requesting in same segment
- Reduce transaction size (if possible)
- Mark the service threads as inactive
- Repeat until all threads in ½ warp are complete



Coalescing with 2d Arrays

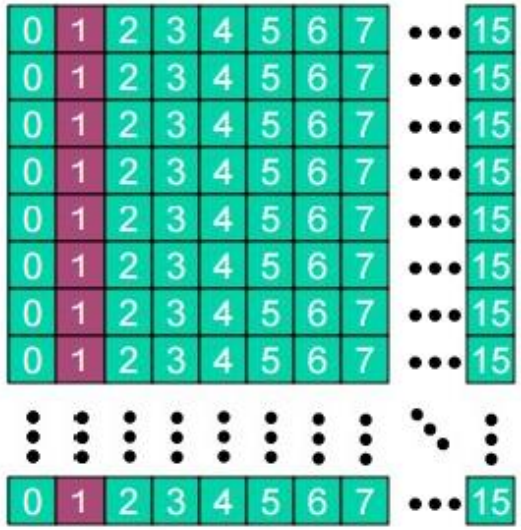
- All warps in a block access consecutive elements within a row as they step through neighboring columns.

```
I =  blockIdx.y*by + ty;
J =  blockIdx.x*bx + tx;
int tx = threadIdx.x
a[ty][tx] = A[I*N+k*by+tx]
b[ty][tx] = B[J+N*(k*bx+ty)]
```



- Accesses by threads in a block along a column don't coalesce.

```
I =  blockIdx.x*bx + tx;
J =  blockIdx.y*by + ty;
a[tx][ty] = A[I*N+k*by+ty]
b[ty][tx] = B[J+N*(k*bx+tx)]
```

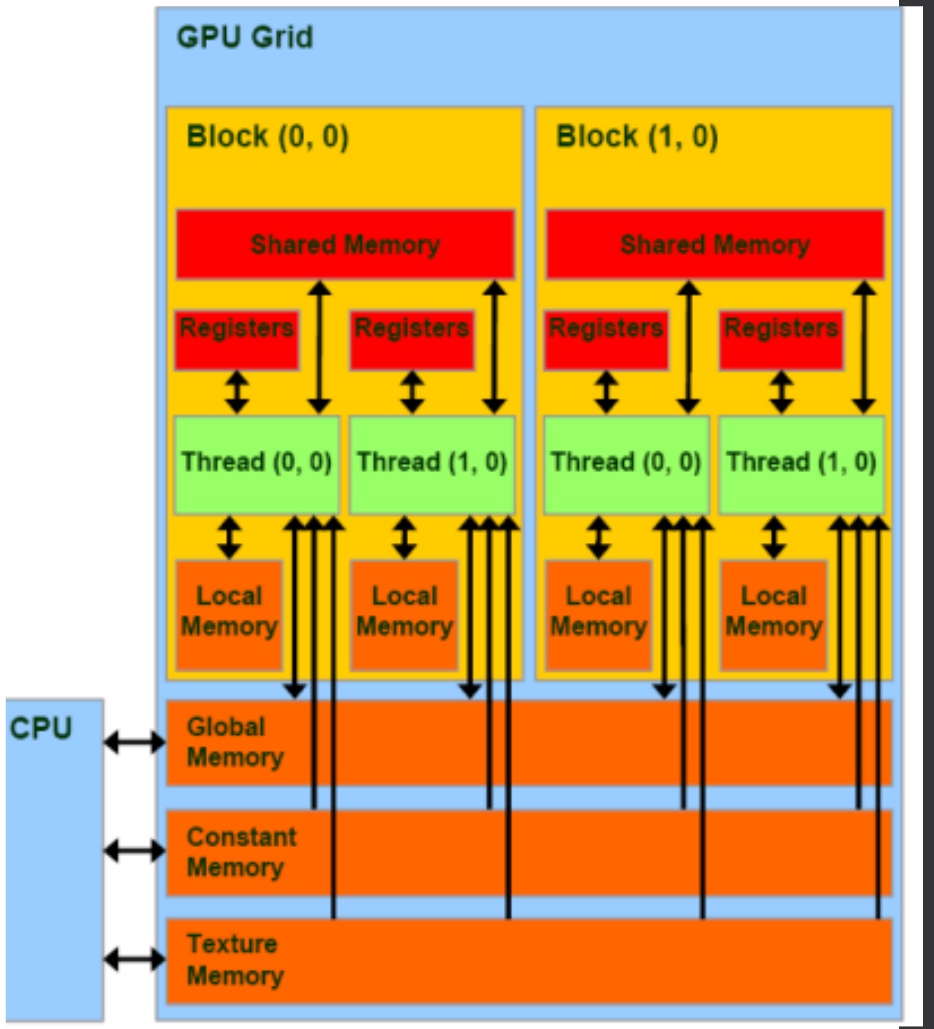


Quick Summary

- Shared Memory
  - Very Fast (2 cycles per read)
  - Divided into banks (32)
  - We don't want to access the same bank with threads in the same warp (bank conflicts, serialize the reads) unless they all access the same 32 bit address (we can broadcast the value to all the threads)
- Global Memory
  - Very Slow (~100 cycles per read)
  - The RAM memory of the GPU
  - We want half a warp to access addresses that fall into same 32, 64 or 128 byte blocks to coalesce the reads (convert multiple memory instructions into a single one)

Constant Memory

- The mechanism for declaring memory constant is similar to the one we used for declaring a buffer as shared memory.
- The instruction to define constant memory is: __constant__
- It must be declare out of the main body and the kernel.
- The instruction cudaMemcpyToSymbol must be used in order to copy the values to be used in the kernel.



Constant Memory

- The mechanism for declaring memory constant is similar to the one we used for declaring a buffer as shared memory.
- The instruction to define constant memory is: __constant__
- It must be declare out of the main body and the kernel.
- The instruction cudaMemcpyToSymbol must be used in order to copy the values to be used in the kernel.

```
// CUDA global constants
  __constant__ int M;

int main(void)
{
...
  cudaMemcpyToSymbol("M", &M, sizeof(M));
...
}
```

Constant Memory

- The variables in constant memory don’t have to be declared in the kernel invocation.

```
__global__ void kernel(float *a, int N)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx<N)
  {
    a[idx] =  a[idx] *M;
  }
}
```

Constant Memory

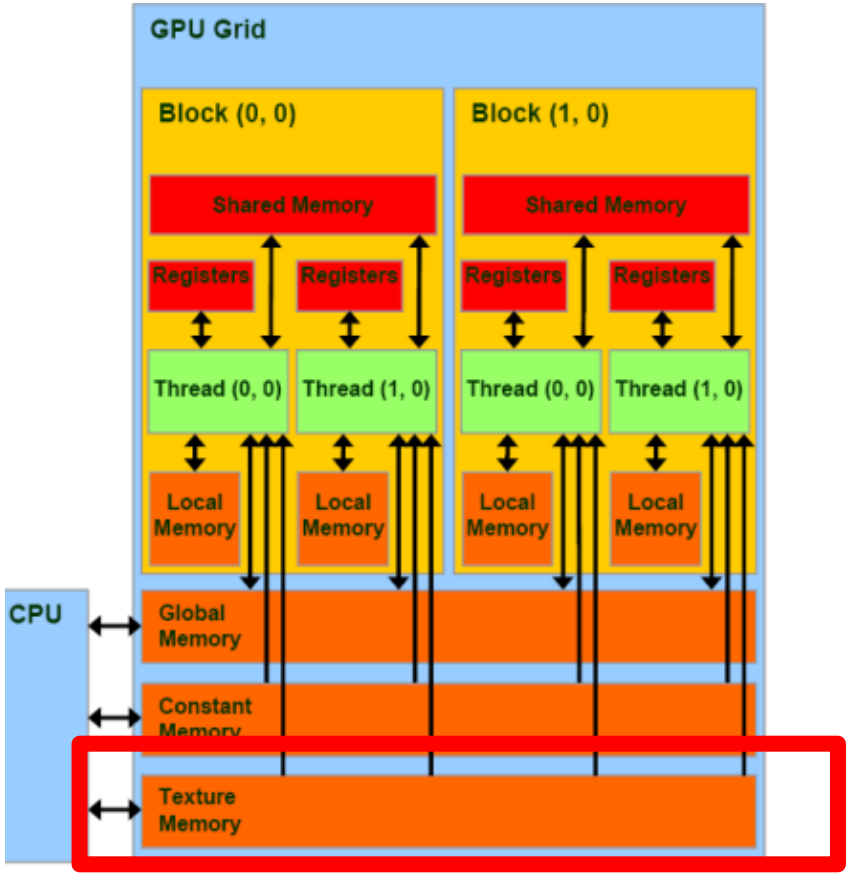
- Performance Considerations
  - Declaring memory as constant constrains the usage to read-only
  - Reading from constant memory can conserve memory bandwidth when compared to reading the same data from global memory.
  - There are 2 reasons why reading from 64 KB constant memory saves bandwidth:
    - A single read from constant memory can be broadcasted to other threads, effectively saving up to 15 reads.
    - Constant memory is cached, so consecutive read of the same address will not incur any additional memory traffic.

Constant Memory

- Performance Considerations
  - There can be a downgrade of the performance when using constant memory
  - However, the half-warp broadcast feature can degrade the performance when all 16 threads read different addresses.
  - If all 16 threads in a half-warp need different data from constant memory, the 16 different reads get serialized.

Texture Memory

- Resides in Global Memory but has its own separate cache
- There are two ways of using TM. The best one is with:
  - Texture references



Texture Memory

- Resides in Global Memory but has its own separate cache
- There are two ways of using TM. The best one is with:
  - Texture references

```
#define N 1024
texture<float, 1, cudaReadModeElementType> tex;

// texture reference name must be known at compile time
__global__ void kernel() {
  int i = blockIdx.x *blockDim.x + threadIdx.x;
  float x = tex1Dfetch(tex, i);
  // do some work using x...
}

void call_kernel(float *buffer) {
  // bind texture to buffer
  cudaBindTexture(0, tex, buffer, N*sizeof(float));

  dim3 block(128,1,1);
  dim3 grid(N/block.x,1,1);
  kernel <<<grid, block>>>();

  // unbind texture from buffer
  cudaUnbindTexture(tex);
}

int main() {
  // declare and allocate memory
  float *buffer;
  cudaMalloc(&buffer, N*sizeof(float));
```