Performance Consideration 2

Prof. Seokin Hong

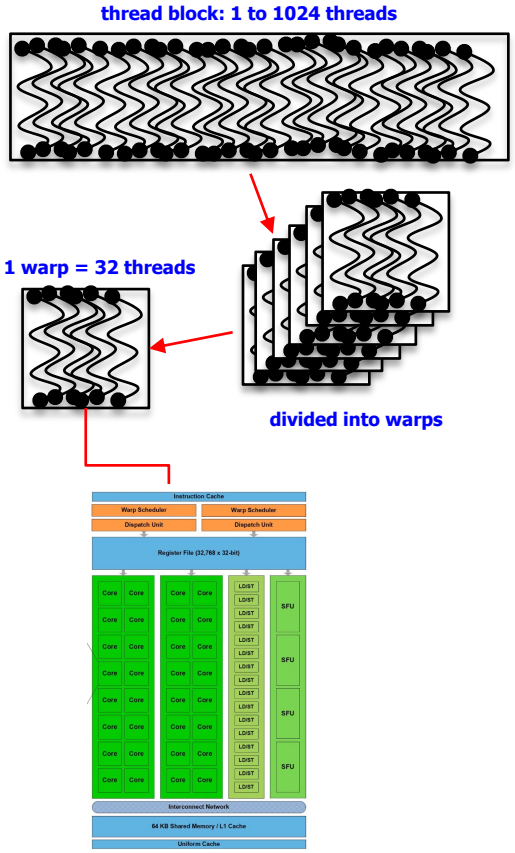
Agenda

- **Memory Optimizations**
  - **More about Global Memory**
  - **Memory Coalescing** to fully utilize global memory bandwidth
    - **Memory Coalescing-aware Memory Allocation**
  - **Reducing Bank Conflict** to fully utilize shared memory bandwidth
- **Considering Control-Flow Divergence**
  - Warps and SIMD Hardware
- **Considering Occupancy**
  - Dynamic Partitioning of Resources
- **Considering Thread Granularity**

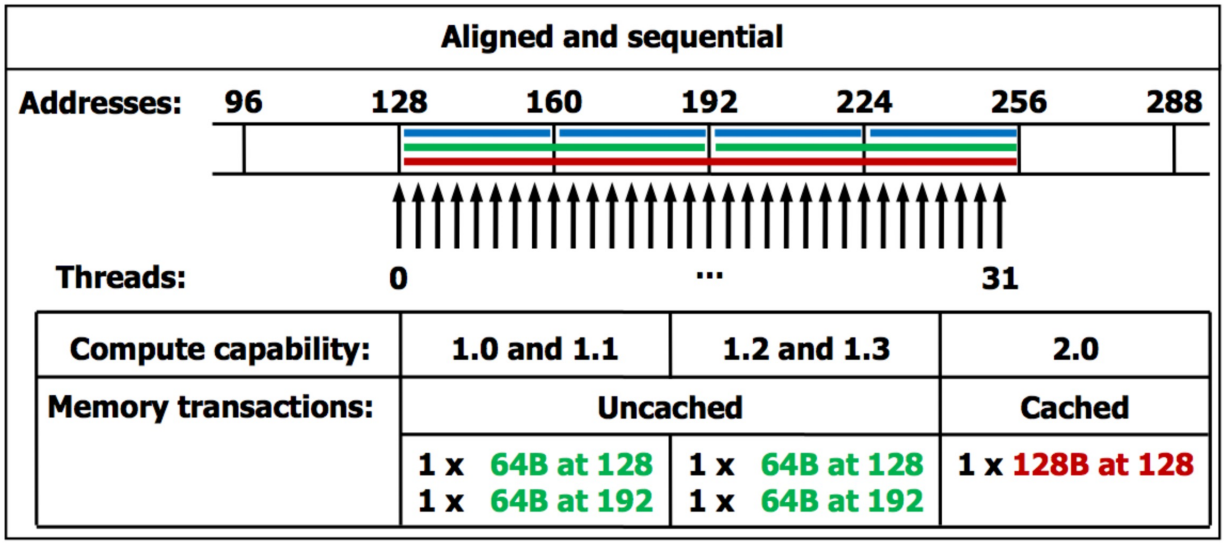
Memory Coalescing-aware Memory Allocation

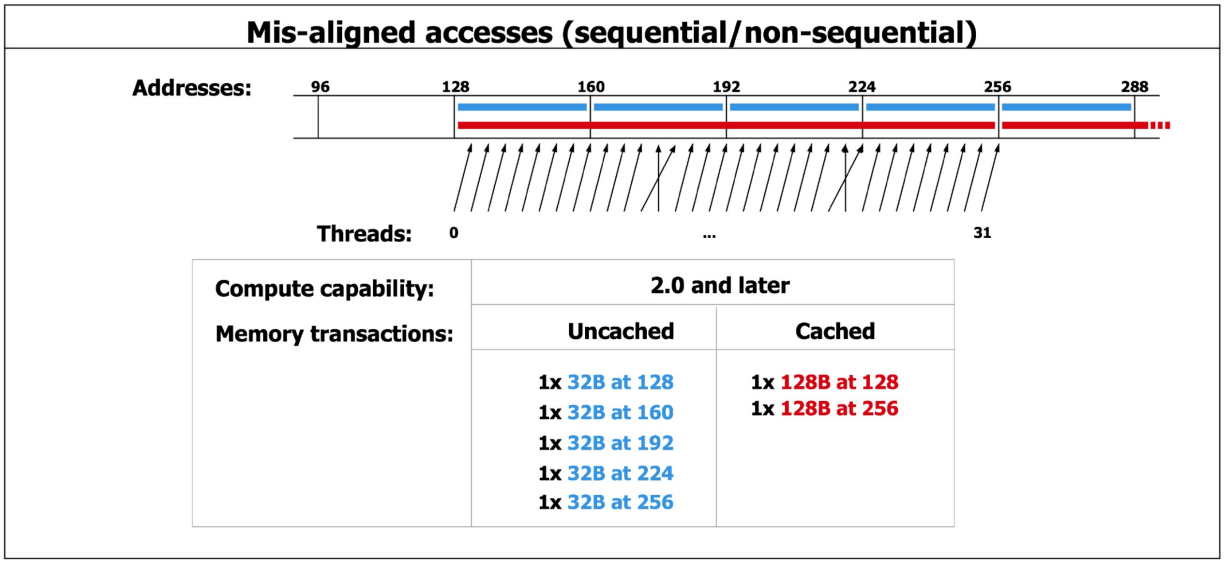
Review: Memory Coalescing

- When **all threads** in a **warp** execute a load instruction,
  - the **hardware** detects whether the **threads** access **consecutive memory addresses**
  - If so, the hardware ***coalesces*** all memory accesses into **a consolidated access** to consecutive DRAM locations
  - 32 threads x 4B = 128B
    - With Coalescing → 1 memory requests
    - Without Coalescing → 32 memory requests



Review: Memory Coalescing





cudaMalloc

- cudaError_t **cudaMalloc**(void** devPtr, size_t size);
  - devPtr (output) : a pointer to allocated device memory
  - size : requested allocation size in bytes

- **cudaFree**(void* devPtr);
  - frees the memory space pointed by devPtr.

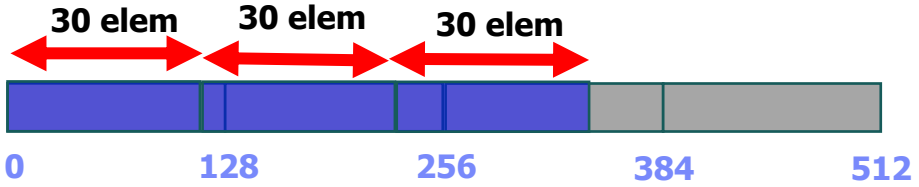
- memory blocks are aligned to 256 bytes boundary

- How about 2D and 3D arrays?
  - At the end of sub-dimension, we need to align to (128 or 256) bytes boundaries for more efficiency.

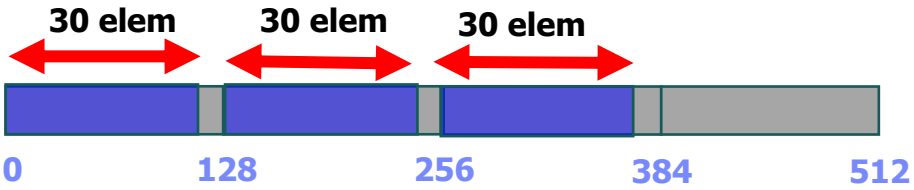
2D Array Cases

- float a[3][30];
  - sub-dimension: 30 elements → 120 bytes
  - totally: 90 elements → 360 bytes

- with cudaMalloc:
  - 360 bytes → 512 bytes allocated

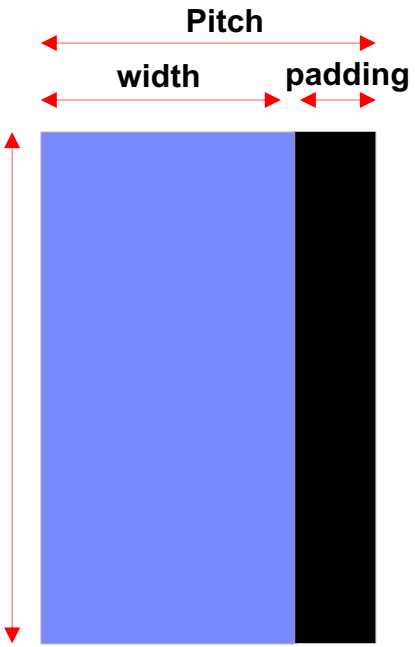


- more good allocation:
  - 360 bytes but different layout
  - each sub-dimensions are aligned



Malloc for 2D data

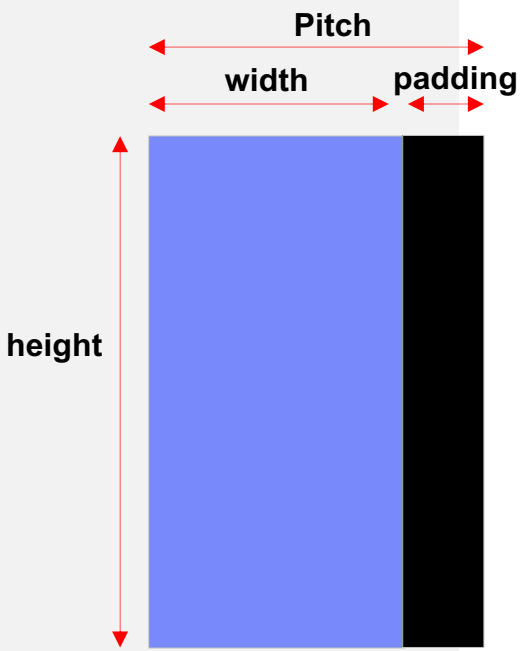
- cudaError_t cudaMallocPitch(void** devPtr,
  size_t* pitch, size_t width, size_t height);
  - pitch : (set by cudaMallocPitch) pitch for allocation (in bytes)
  - **width** : requested pitched allocation width (in bytes)
  - **height** : requested pitched allocation height
  - T* pElement = (T*)**((char*)baseAddr + Row * pitch**) + Col;



- **What cudaMallocPitch does?**
  - Allocate the first row
  - Check if the number of bytes allocated makes it correctly aligned for memory coalescing
  - If not, allocate further bytes
  - **the pitch is then the number of bytes allocated for a single row, including the extra bytes (padding bytes).**
  - Reiterate for each row.

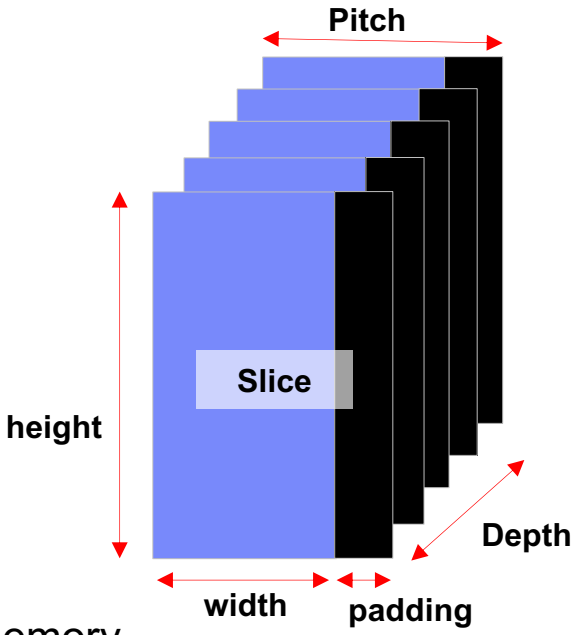
cudaMallocPitch example

```
// Host code
int  width = 100, height = 64;
float*  devPtr;
size_t  pitch;
cudaMallocPitch(&devPtr, &pitch, width * sizeof(float), height);
MyKernel<<<100, 512>>>(devPtr, pitch, width, height);


// Device code
__global__ void  MyKernel(float*  devPtr, size_t  pitch, int  width, int  height) {
    for (int r = 0; r < height; ++r) {
        float* row = (float*)((char*)devPtr + r * pitch);
        for (int c = 0; c < width; ++c) {
            float element = row[c];
        }
    }
}
```



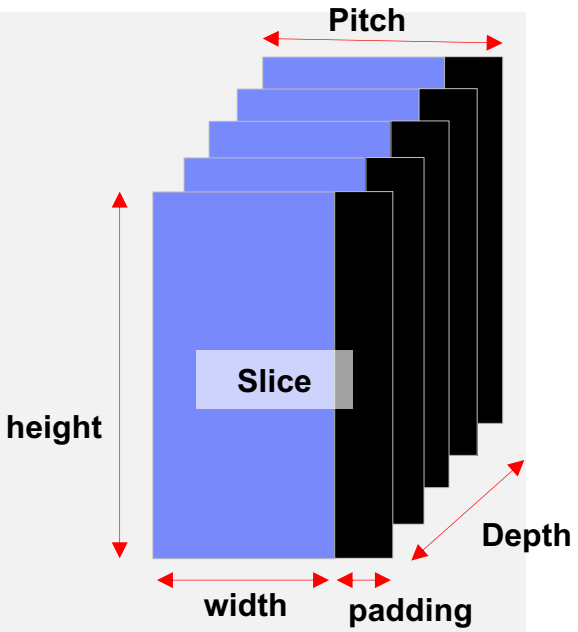
Malloc for 3D data

- **cudaMalloc3D**(struct cudaPitchedPtr* pitchedDevPtr,
  struct cudaExtent extent);
  - struct **cudaPitchedPtr** {
    size_t pitch;
    void* ptr;
    size_t xsize, ysize;
    }
  - struct **cudaExtent** {
    size_t depth, height, width;
    }
  - **pitchedDevPtr** : (set by cudaMalloc3D) pointer to allocated memory
  - **extent** : requested allocation size



cudaMalloc3D example

```
// Host code
int  width = 64, height = 64, depth = 64;
cudaExtent  extent = make_cudaExtent(width *sizeof(float), height, depth);
cudaPitchedPtr  devPitchedPtr;
cudaMalloc3D(&devPitchedPtr, extent);
myKernel<<<100, 512>>>(devPitchedPtr, width, height, depth);


// Device code
__global__ void myKernel(cudaPitchedPtr devPitchedPtr, int width, int height, int depth) {
    char*  devPtr = devPitchedPtr.ptr;
    size_t  pitch = devPitchedPtr.pitch;
    size_t  slicePitch = pitch * height;
    for (int z = 0; z < depth; ++z) {
        char* slice = devPtr + z * slicePitch;
        for (int y = 0; y < height; ++y) {
            float* row = (float*)(slice + y * pitch);
            for (int x = 0; x < width; ++x) {
                float element = row[x];
            }
        }
    }
}
```



Memcpy for 2D data

- **cudaMemcpy**(void* dst, const void* src,
  size_t count, enum cudaMemcpyKind kind);
- **cudaMemcpy2D**(void* dst, size_t dpitch,
  const void* src, size_t spitch,
  size_t width, size_t height, enum cudaMemcpyKind kind);
  - **dpitch, spitch :** pitch of destination / source memory
  - **width** : width of matrix transfer (columns in bytes)
  - **height** : height of matrix transfer (rows)

```
float * A, *dA;
A = (float *)malloc(sizeof(float)*N*N);

size_t pitch;
cudaMallocPitch(&dA, &pitch, sizeof(float)*N, N);
cudaMemcpy2D(dA, pitch, A, sizeof(float)*N, sizeof(float)*N, N, cudaMemcpyHostToDevice);
```

Memcpy for 3D data

- cudaMemcpy3D(const struct cudaMemcpy3DParams* p);
  - struct cudaExtent { size_t depth, height, width; }
  - struct cudaPos { size_t x, y, z; }
  - struct cudaMemcpy3DParms {

```
struct cudaArray*  srcArray;
struct cudaPos        srcPos;
struct cudaPitchedPtr srcPtr;
struct cudaArray*     dstArray;
struct cudaPos        dstPos;
struct cudaPitchedPtr dstPtr;
struct cudaExtent     extent;
enum cudaMemcpyKind   kind;
};
```

```
struct cudaPitchedPtr {
  size_t pitch;
  void* ptr;
  size_t xsize, ysize;
}
```

Memcpy for 3D data - Example

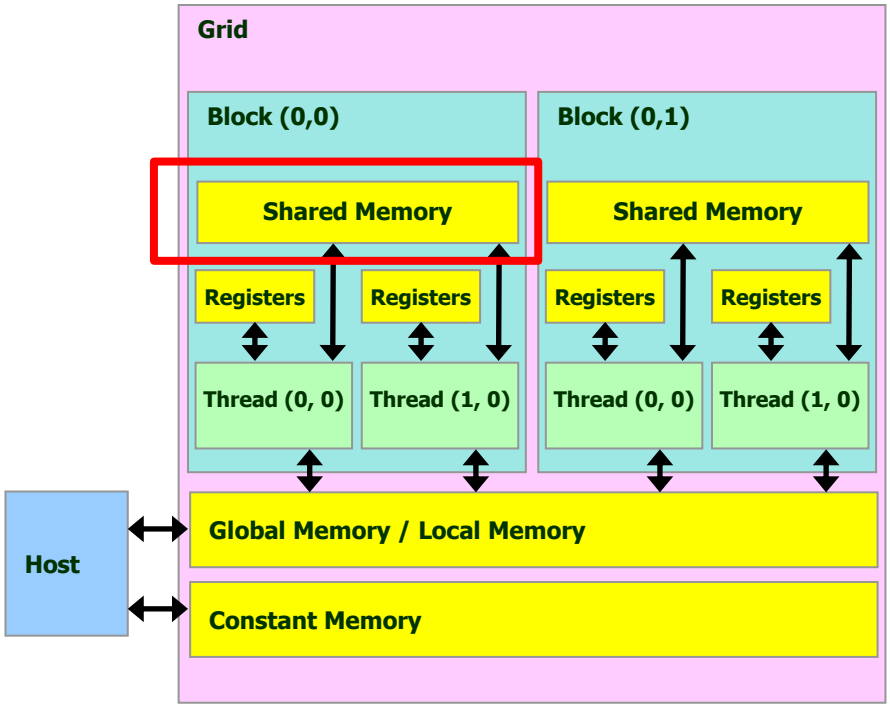
```
int i, j, k, value = 0;
int Nx = 2, Ny = 6, Nz = 4;
int ***h_tensor;
struct cudaPitchedPtr d_tensor;
h_tensor = alloc_tensor(Nz, Ny, Nx);
cudaMalloc3D(&d_tensor, make_cudaExtent(Nx * sizeof(int), Ny, Nz));

for(i = 0; i < Nx; i++) {
  for(j = 0; j < Ny; j++) {
    for(k = 0; k < Nz; k++) {
      h_tensor[k][j][i] = value++; } } }

cudaMemcpy3DParms cpy = { 0 };
cpy.srcPtr = make_cudaPitchedPtr(h_tensor[0][0], Nx * sizeof(int), Nx, Ny);
cpy.dstPtr = d_tensor;
cpy.extent = make_cudaExtent(Nx * sizeof(int), Ny, Nz);
cpy.kind = cudaMemcpyHostToDevice;
cudaMemcpy3D(&cpy);
```


https://stackoverflow.com/questions/23310520/using-cudamemcpy3d-to-transfer-pointer

Reducing Bank Conflict to fully utilize shared memory bandwidth

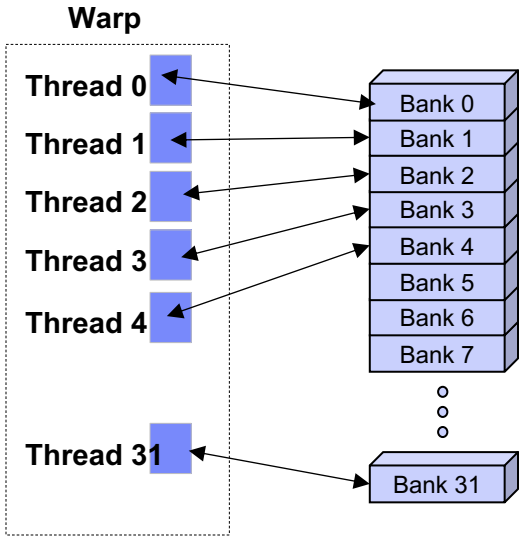
Shared Memory

- **Uses**:
  - Inter-thread communication within a block
  - Cache data to reduce global memory accesses



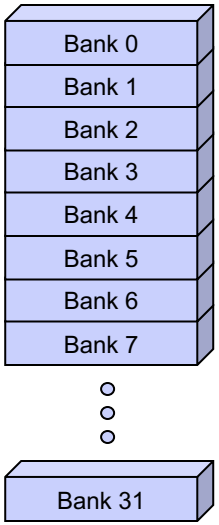
Shared Memory Organization

- Shared memory is divided into banks !
  - Essential to achieve high bandwidth
  - 16 banks, 32-bit wide banks (old architecture)
  - 32 banks, 32-bit wide banks (current architecture)
- Each bank can service one memory access per cycle
  - A memory can service as many simultaneous accesses as it has banks
- **Each threads within a warp can access different banks** or can all access the same value



Banked Memory Layout

- How addresses map to banks in CUDA?
  - Each bank has a bandwidth of 32 bits per clock cycle
  - Successive 32-bit words are assigned to successive banks
  - If the number of banks is 32,
    - Bank number = address % 32



- **Example:** __shared__ float a[1024];

Bank 0: a[0], a[32], a[64], …

Bank 1: a[1], a[33], a[65], …

Bank 2: a[2], a[34], a[66], …

Bank 3: a[3], a[35], a[67], …

Bank 4: a[4], a[36], a[68], …

Bank 5: a[5], a[37], a[69], …

Bank 6: a[6], a[38], a[70], …

...

Bank 31: a[31], a[63], a[95], …

Broadcast Cases

- **Example:** __shared__ float a[1024];

thread 0 reads a[32] → bank 0

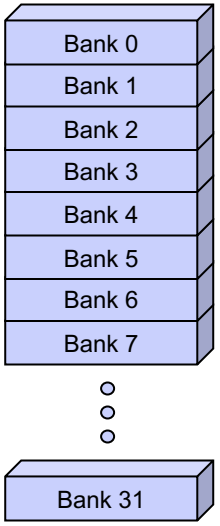
thread 1 reads a[32] → bank 0

thread 2 reads a[32] → bank 0

thread 3 reads a[32] → bank 0

thread 4 reads a[32] → bank 0

…



- Read a[32] from bank 0 and broadcast it to multiple threads in a single clock cycle

Multicast Cases

- only on the device with the compute capability 2.0 or above
- **Example**: __shared__ float a[1024];

thread 0 reads a[0] → bank 0

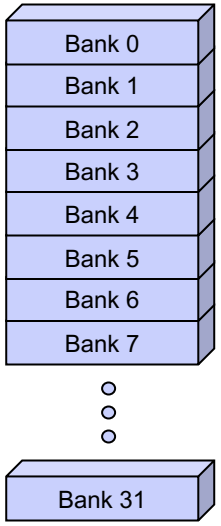
thread 1 reads a[0] → bank 0

thread 2 reads a[1] → bank 1

thread 3 reads a[1] → bank 1

thread 4 reads a[2] → bank 2

…

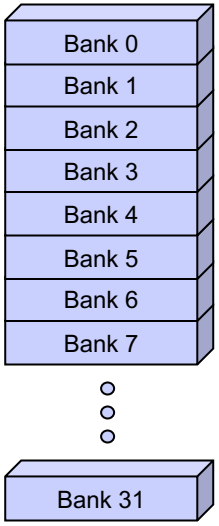


- Read a[0] from bank 0 and send it to thread0 and thread1 in a single clock cycle
- Read a[1] from bank 1 and send it to thread2 and thread3 in a single clock cycle

Serialization Cases

- thread 0 reads a[0] → bank 0
- thread 1 reads a[32] → bank 0
- thread 2 reads a[64] → bank 0
- thread 3 reads a[96] → bank 0
- thread 4 reads a[128] → bank 0
- …

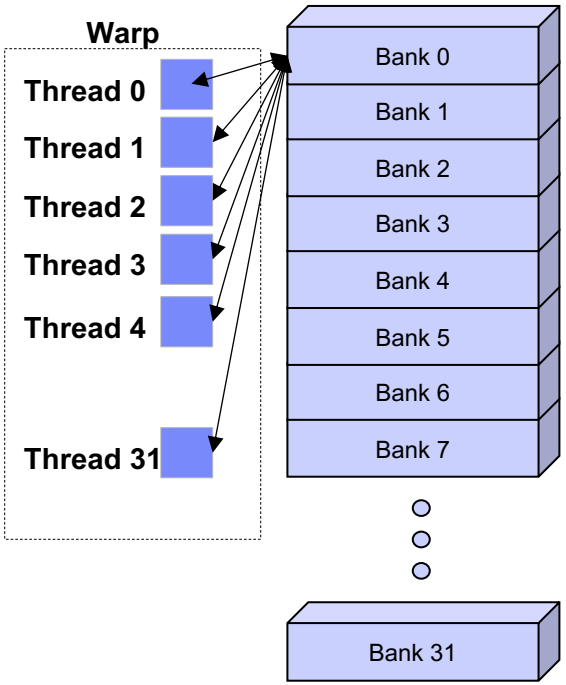
- bank 0 returns
  a[0], a[32], a[64], a[96], …
  serially,
  in 32 clock cycles → **Bank Conflict**



Bank Conflict

- Multiple simultaneous accesses to a bank result in a **bank conflict**

- **The fast case:**
  - If all threads of a warp access different banks, there is no bank conflict
  - If **all** threads of a warp access the same word, there is no bank conflict (**broadcast**)
  - On devices with compute capability 2.0 and above, if **more than one thread** of a warp access exactly the same word, there is no bank conflict (**Multicast**)
- **The slow case:**
  - **Bank conflict:** multiple threads in the same warp access different words stored in the same bank
  - Must serialize the accesses
  - Cost= max # of simultaneous access to a single bank

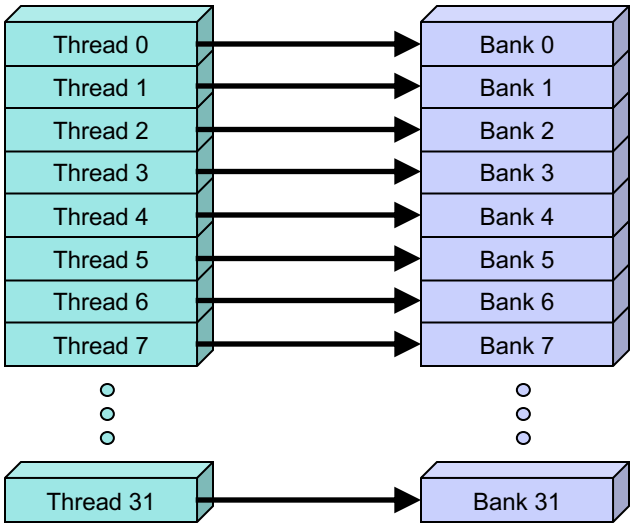


- Shared memory is as fast as registers if there are no bank conflicts

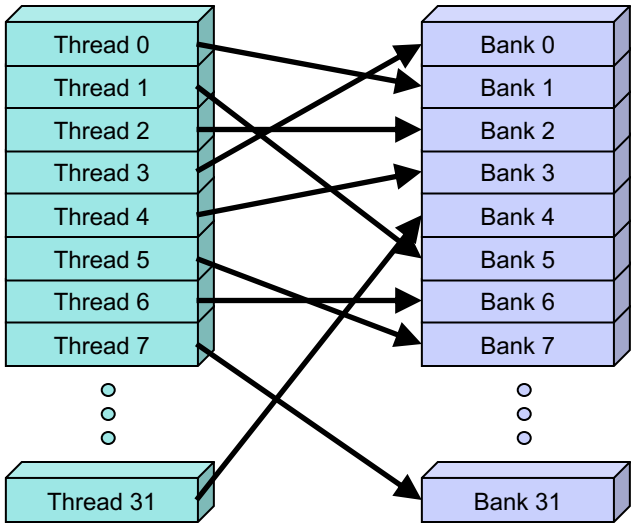
→ **To fully utilize the bandwidth of shared memory, need to avoid bank conflicts**

Bank Addressing Examples

- No Bank Conflicts

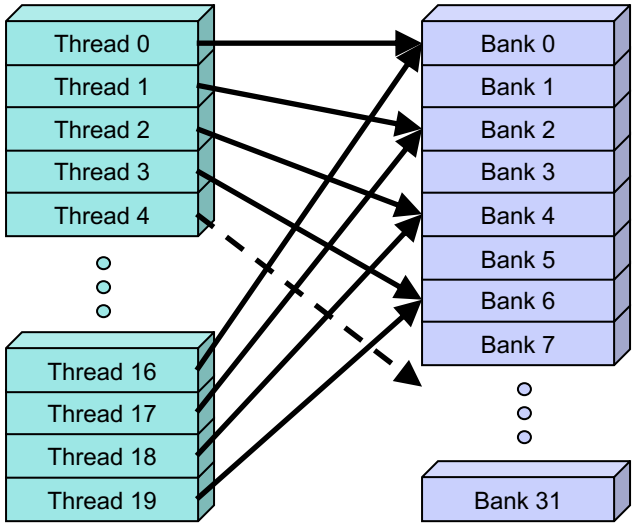


- No Bank Conflicts

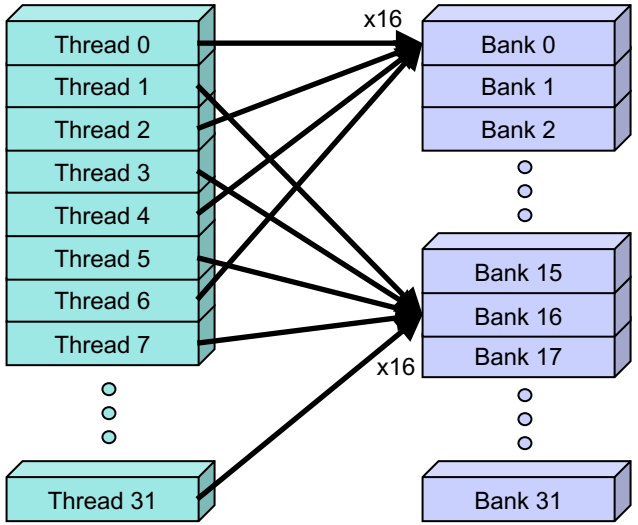


Bank Addressing Examples (Cont'd)

- 2-way Bank Conflicts
  - stride 2 case



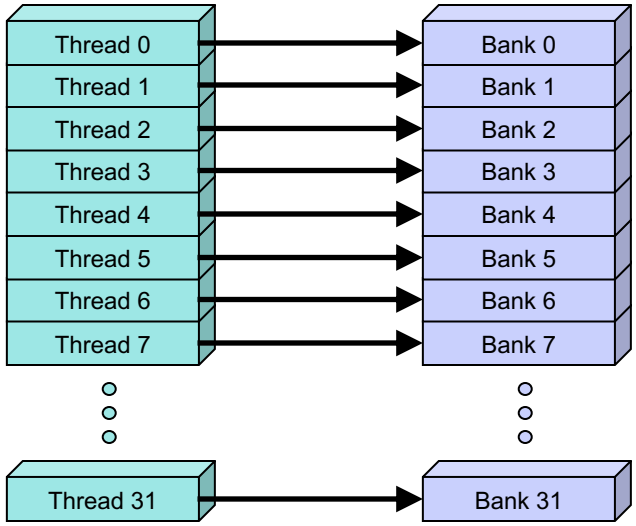
- 16-way Bank Conflicts
  - even or odd case



Bank Addressing Examples (Cont'd)

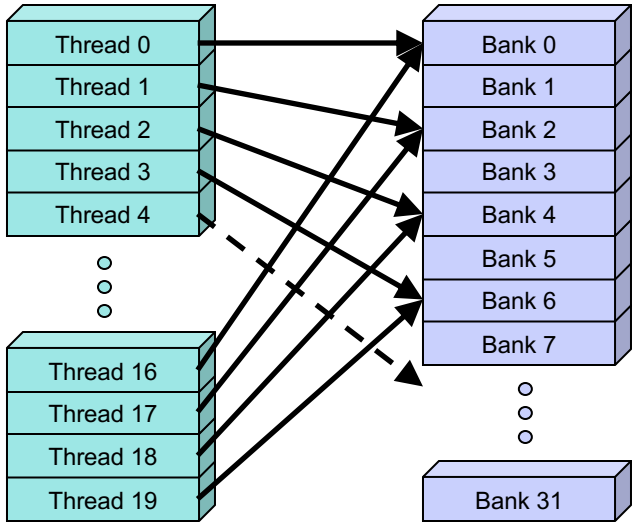
```
__shared__ float shared[256];
float data = shared[ BaseIndex + tid];
```

No Bank Conflicts




```
__shared__ float shared[256];
float data = shared[ BaseIndex + 2*tid];
```

2-way Bank Conflicts

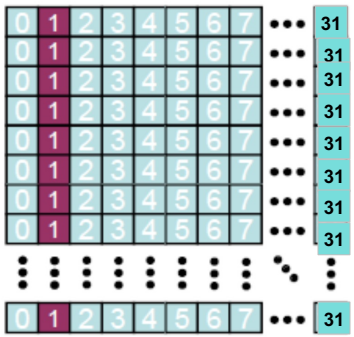


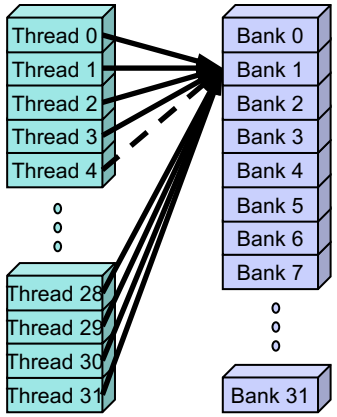
Avoiding Bank Conflict

- **Memory padding:** In case of 2D array, add one float to end of each row

- **Example: 32x32 Array**
  - Each thread processes a row
  - So threads in a block access the elements in each column simultaneously
  - 32-way bank conflicts

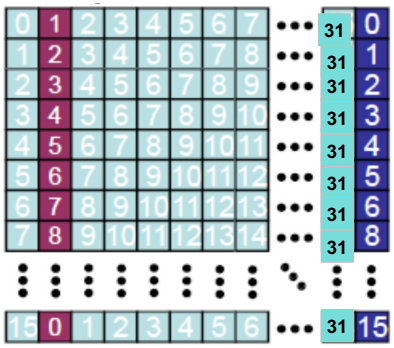
Bank indices without padding

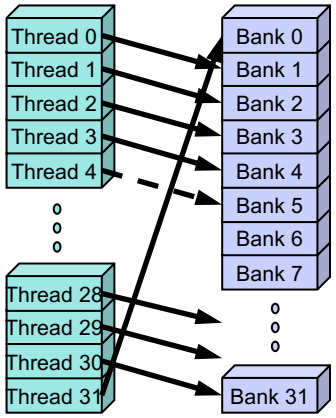




  - After padding one float to the end of each row,
    - The elements in each column is mapped to different banks

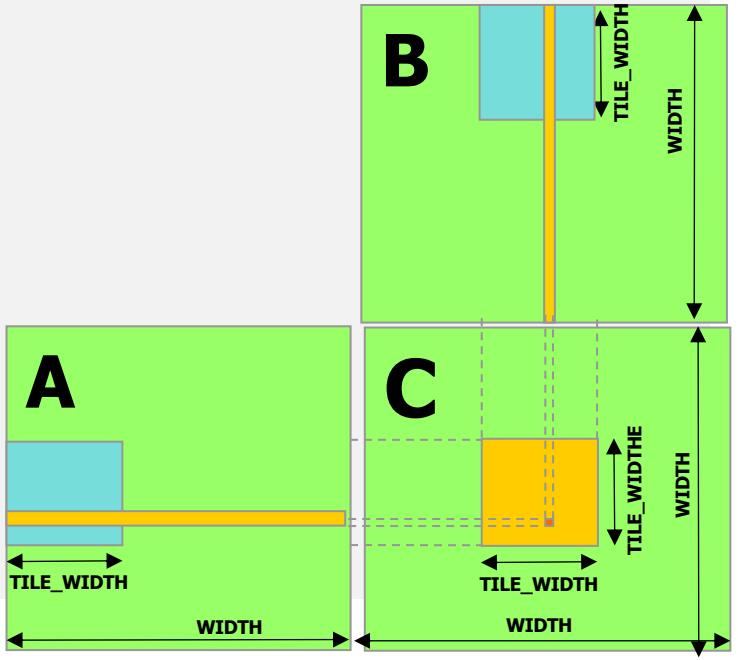
Bank indices with padding





Review: Tiled Matrix Multiplication Kernel

```
__global__ void matmul(float* g_C, const float* g_A, const float* g_B, const int width) {
    __shared__ float s_A[TILE_WIDTH][TILE_WIDTH];
    __shared__ float s_B[TILE_WIDTH][TILE_WIDTH];
    int by = blockIdx.y; int bx = blockIdx.x;
    int ty = threadIdx.y; int tx = threadIdx.x;
    int gy = by * TILE_WIDTH + ty; // global y index
    int gx = bx * TILE_WIDTH + tx; // global x index
    float sum = 0.0F;
    for (register int m = 0; m < width / TILE_WIDTH; ++m) {
        // read into the shared memory blocks
        s_A[ty][tx] = g_A[gy * width + (m * TILE_WIDTH + tx)];
        s_B[ty][tx] = g_B[(m * TILE_WIDTH + ty) * width + gx];
        __syncthreads();
        // use the shared memory blocks to get the partial sum
        for (register int k = 0; k < TILE_WIDTH; ++k) {
            sum += s_A[ty][k] * s_B[k][tx];
        }
        __syncthreads();
    }
    g_C[gy * width + gx] = sum;
}
```



Bank Conflicts in the Tiled Matrix Multiplication?

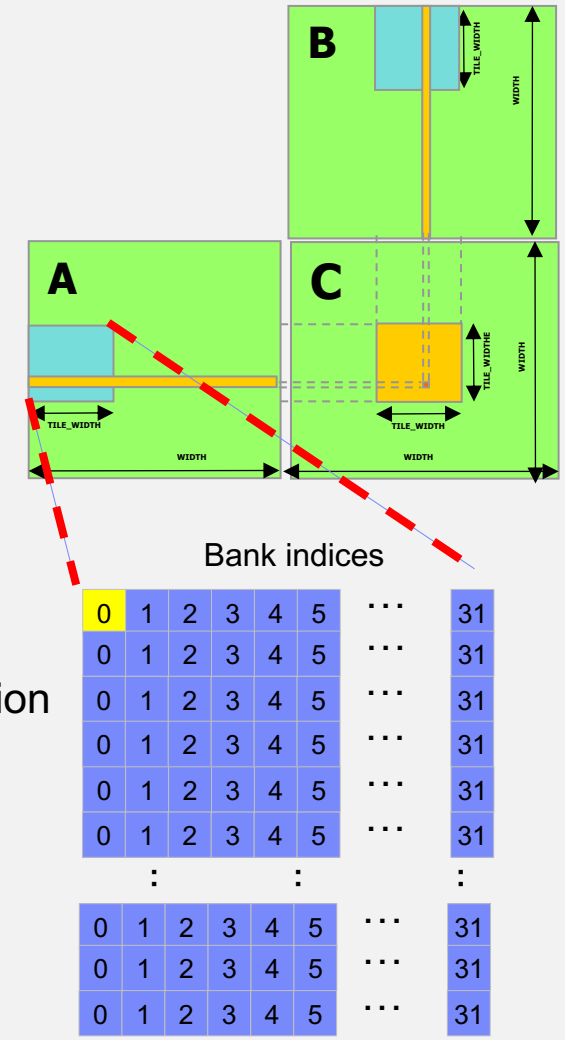
```
__global__ void matmul(float* g_C, const float* g_A, const float* g_B, const int width) {
  ......
      // use the shared memory blocks to get the partial sum
      for (register int k = 0; k < TILE_WIDTH; ++k) {
         sum += s_A[ty][k] * s_B[k][tx];
      }
  ……
}
```

s_A[ty*TILE_WIDTH + k] **s_B[k*TILE_WIDTH + tk]**

Bank Conflicts in the Tiled Matrix Multiplication?

```
__global__ void matmul(float* g_C, const float* g_A, const float* g_B, const int width) {
  ......
    // use the shared memory blocks to get the partial sum
    for (register int k = 0; k < TILE_WIDTH; ++k) {
      sum += s_A[ty][k] * s_B[k][tx];
    }
  ……
}
```

s_A[ty*TILE_WIDTH + k]



For TILE_WIDTH =32,

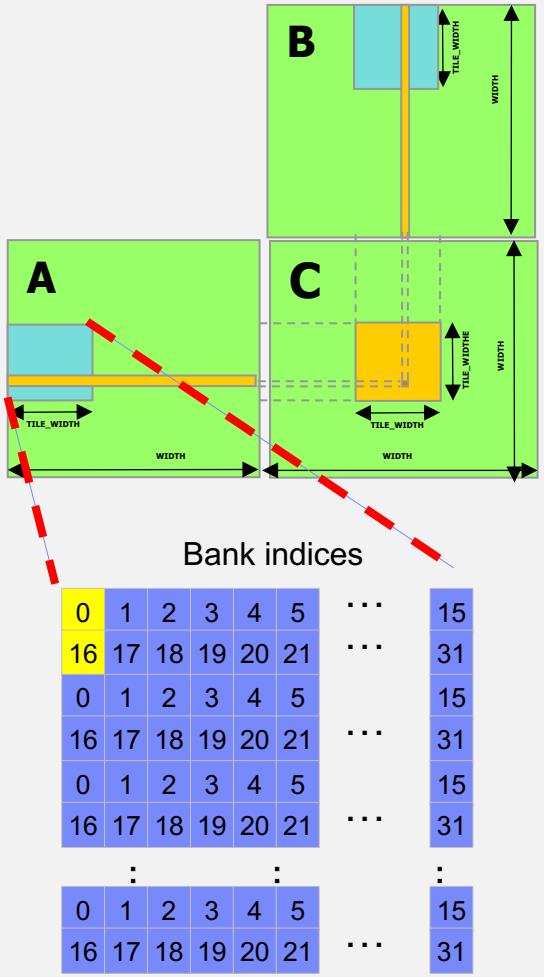
The whole warp is accessing the same shared memory location

→ **Boardcasting the same value → No bank conflict**

Bank Conflicts in the Tiled Matrix Multiplication?

```
__global__ void matmul(float* g_C, const float* g_A, const float* g_B, const int width) {
  ......
    // use the shared memory blocks to get the partial sum
    for (register int k = 0; k < TILE_WIDTH; ++k) {
      sum += s_A[ty][k] * s_B[k][tx];
    }
  ……
}
```

s_A[ty*TILE_WIDTH + k]



For TILE_WIDTH =16,

Two different shared memory location is accessed

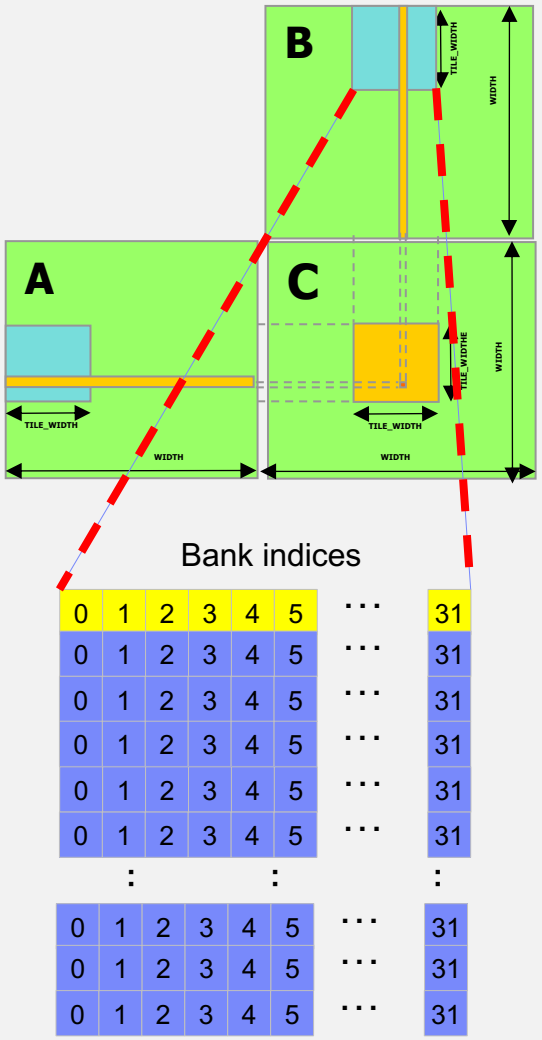
→ **16-way bank conflict** on device of compute capability 1.0

→ On device with compute capability 2.0 and above, **multicasting** the same value → no bank conflict

Bank Conflicts in the Tiled Matrix Multiplication?

```
__global__ void matmul(float* g_C, const float* g_A, const float* g_B, const int width) {
  ......
    // use the shared memory blocks to get the partial sum
    for (register int k = 0; k < TILE_WIDTH; ++k) {
      sum += s_A[ty][k] * s_B[k][tx];
    }
  ……
}
```

s_B[k*TILE_WIDTH + tx]



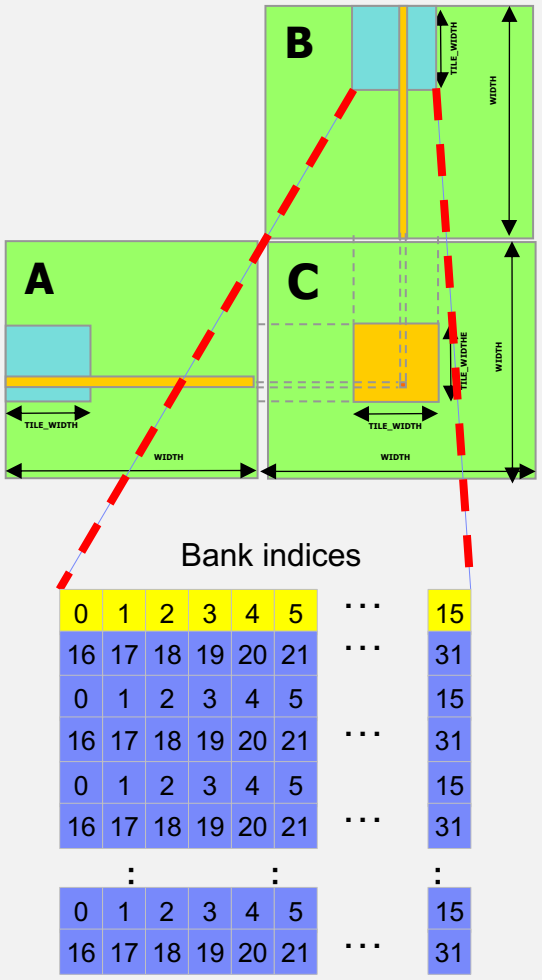
For TILE_WIDTH =32,

Each thread in a warp is accessing different shared memory location → No bank conflict

Bank Conflicts in the Tiled Matrix Multiplication?

```
__global__ void matmul(float* g_C, const float* g_A, const float* g_B, const int width) {
  ......
    // use the shared memory blocks to get the partial sum
    for (register int k = 0; k < TILE_WIDTH; ++k) {
      sum += s_A[ty][k] * s_B[k][tx];
    }
  ……
}
```

s_B[k*TILE_WIDTH + tx]



For TILE_WIDTH =16,

Two threads in a warp access the same value in a bank

→2-way bank conflict on device of compute capability 1.0

→ On device with compute capability 2.0 and above,

multicasting the same value → no bank conflict

Next?

- **Memory Optimizations**
  - **More about Global Memory**
  - **Memory Coalescing** to fully utilize global memory bandwidth
    - **Memory Coalescing-aware Memory Allocation**
  - **Reducing Bank Conflict** to fully utilize shared memory bandwidth

- **Considering Control-Flow Divergence**
  - Warps and SIMD Hardware

- **Considering Occupancy**
  - Dynamic Partitioning of Resources

- **Considering Thread Granularity**