More CUDA Examples

Shengren Li
shrli@ucdavis.edu

CUDA on CSIF

CUDA 5.5

/usr/local/cuda-5.5

In .tcshrc, add

```
# CUDA
setenv CUDA /usr/local/cuda-5.5
setenv PATH ${PATH}:${CUDA}/bin
setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:${CUDA}/lib64
```

nvcc -V

`cudaGetDeviceProperties`

CUDA on CSIF

- pc33
    - GeForce GTS 250 (Tesla)
    - Compute capability 1.1
    - Global memory 500MB
    - Shared memory 16KB
    - Max number of threads per block 512
- pc43
    - GeForce GTX 550 Ti (Fermi)
    - Compute capability 2.1
    - Global memory 1GB
    - Shared memory 48KB
    - Max number of threads per block 1024

saxpy: Single-precision A*X Plus Y

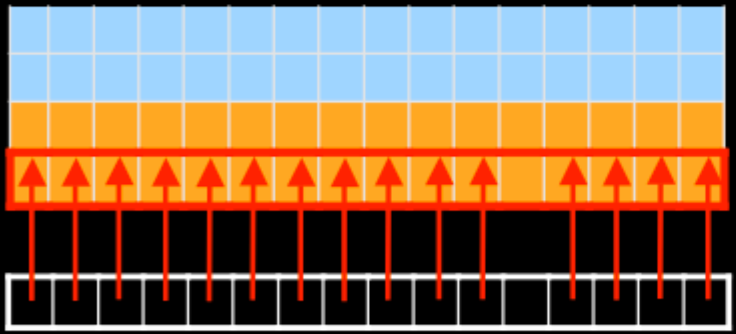
http://devblogs.nvidia.com/parallelforall/easy-introduction-cuda-c-and-c/

```
__global__
void saxpy(int n, float a, float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

int main(void)
{
  int N = 1<<20;
  float *x, *y, *d_x, *d_y;
  x = (float*)malloc(N*sizeof(float));
  y = (float*)malloc(N*sizeof(float));

  cudaMalloc(&d_x, N*sizeof(float));
  cudaMalloc(&d_y, N*sizeof(float));

  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

  // Perform SAXPY on 1M elements
  saxpy<<<(N+255)/256, 256>>>(N, 2.0, d_x, d_y);

  cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);

  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = max(maxError, abs(y[i]-4.0f));
  printf("Max error: %fn", maxError);
}
```

Coalescing
Compute capability 1.0 and 1.1

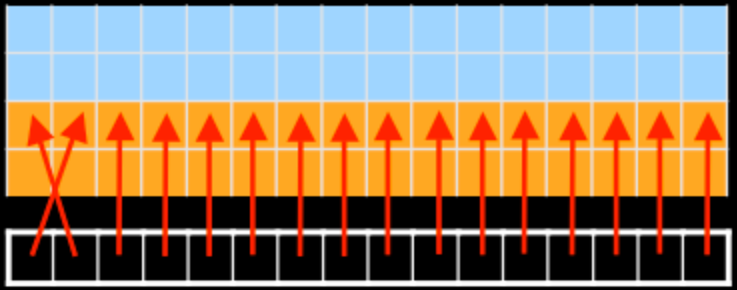
- K-th thread must access k-th word in the segment (or k-th word in 2 contiguous 128B segments for 128-bit words), not all threads need to participate

Coalesces – 1 transaction

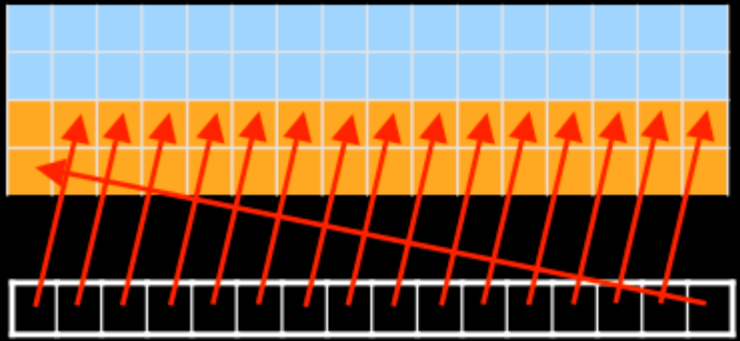


Half-warp of threads

Out of sequence – 16 transactions

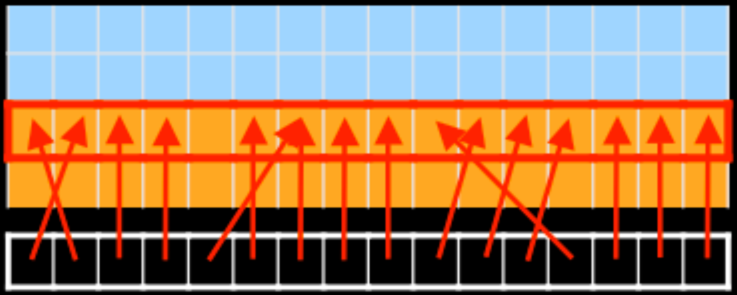


Misaligned – 16 transactions

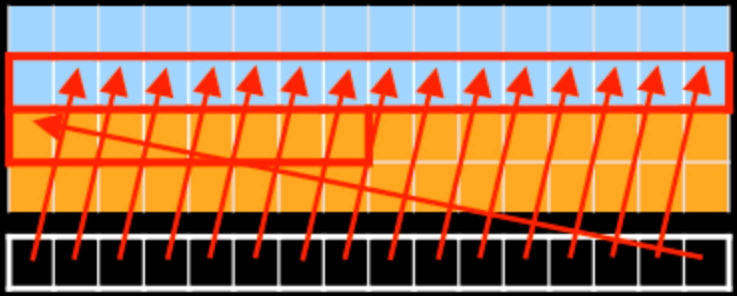


Coalescing
Compute capability 1.2 and higher

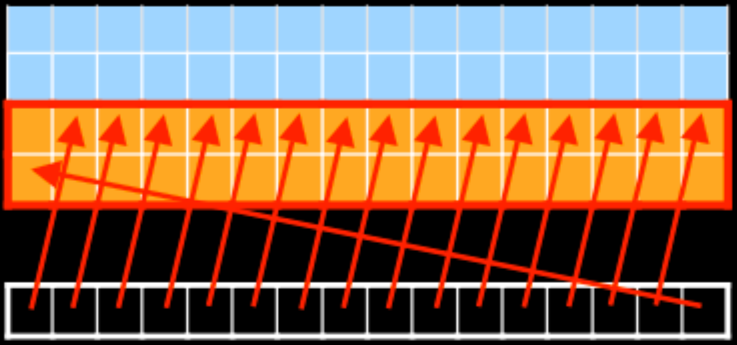
- Coalescing is achieved for any pattern of addresses that fits into a segment of size: 32B for 8-bit words, 64B for 16-bit words, 128B for 32- and 64-bit words
- Smaller transactions may be issued to avoid wasted bandwidth due to unused words


1 transaction - 64B segment

2 transactions - 64B and 32B segments



1 transaction - 128B segment



Matrix transpose

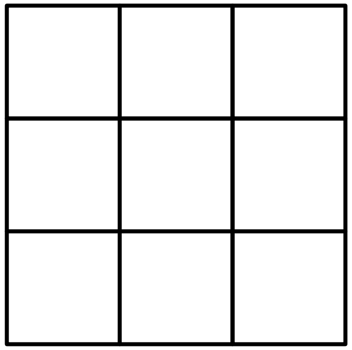
http://devblogs.nvidia.com/parallelforall/efficient-matrix-transpose-cuda-cc/

- Transpose a matrix of single precision values
- Out-of-place, i.e., input and output are separate arrays in memory
- Square matrices whose dimensions are multiples of 32, e.g., 1024 x 1024
- All kernels launch blocks of 32 x 8 threads
- Each thread block processes a tile of size 32 x 32
- Performance metric: effective bandwidth(GB/s)
  = 2 * matrix-size(GB) / execution-time(s)
  higher is better

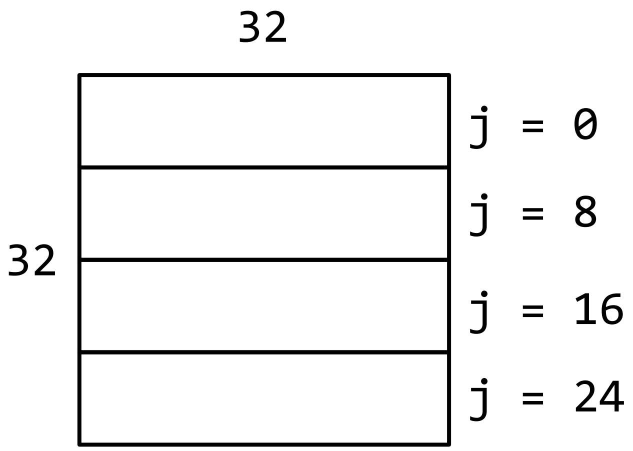
Simple matrix copy

```
__global__ void copy(float *odata, const float *idata)
{
  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;

  for (int j = 0; j < TILE_DIM; j+= BLOCK_ROWS)
    odata[(y+j)*width + x] = idata[(y+j)*width + x];
}
```

```
TILE_DIM = 32
BLOCK_ROWS = 8
```



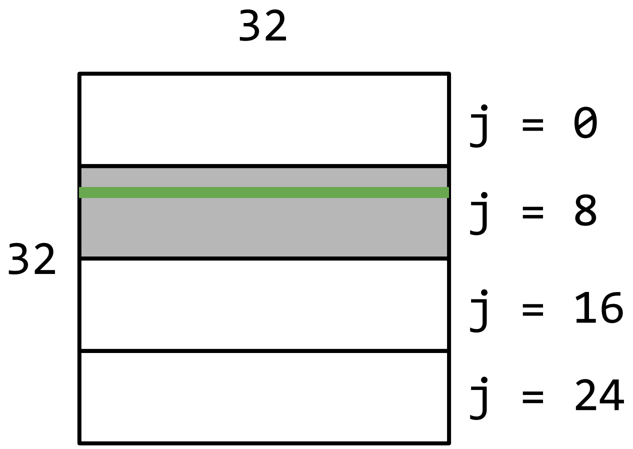
e.g., 3 x 3 tiles

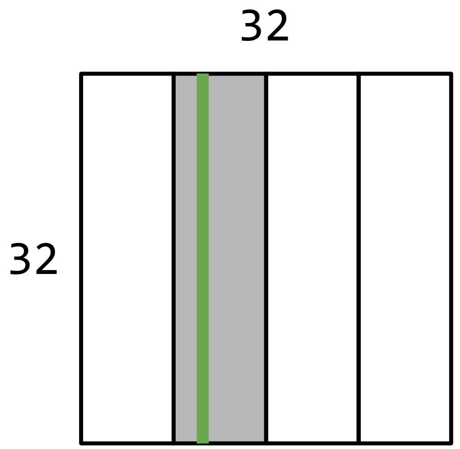


- Both reads from idata and writes to odata are coalesced
- Use copy performance as baseline, i.e., expect matrix transpose to achieve the same performance

Naive Matrix Transpose

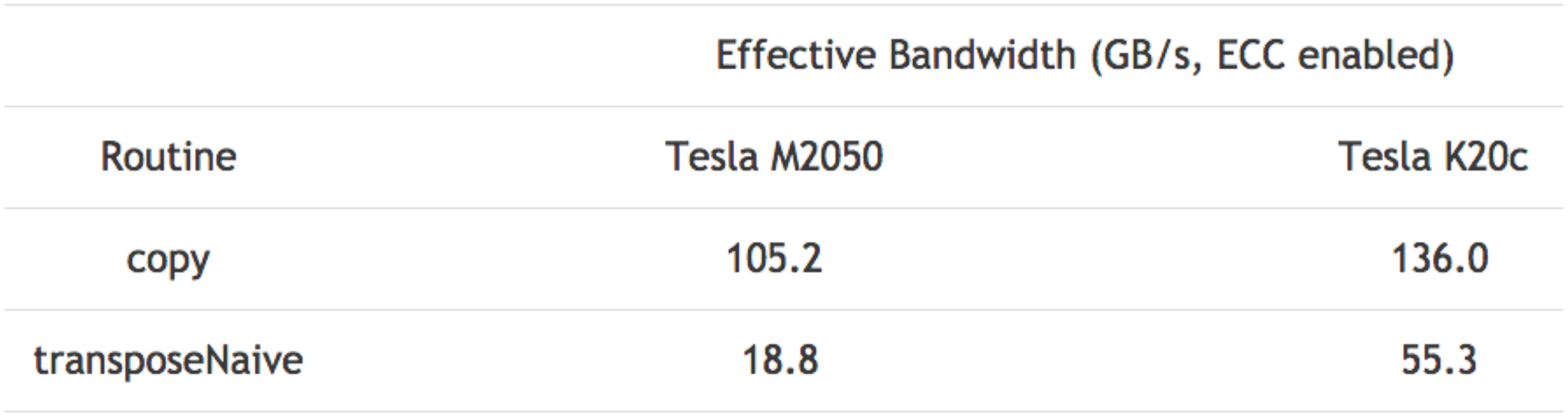
```
__global__ void transposeNaive(float *odata, const float *idata)
{
  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;

  for (int j = 0; j < TILE_DIM; j+= BLOCK_ROWS)
    odata[x*width + (y+j)] = idata[(y+j)*width + x];
}
```





- Reads from idata are coalesced
- Writes to odata are non-coalesced
- A stride of *width* elements between contiguous threads

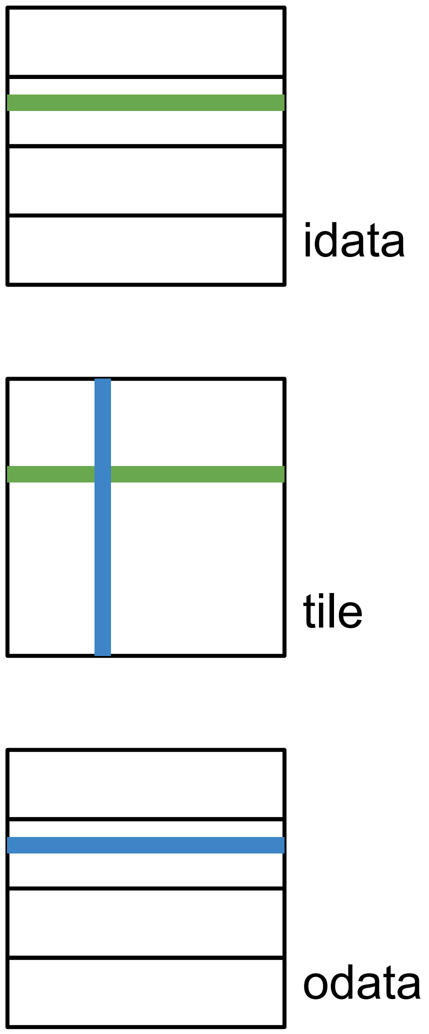
Non-coalesced global memory access hurts performance



	Effective Bandwidth (GB/s, ECC enabled)	
Routine	Tesla M2050	Tesla K20c
copy	105.2	136.0
transposeNaive	18.8	55.3

Coalesced transpose via shared memory

```
__global__ void transposeCoalesced(float *odata, const float *idata)
{
  __shared__ float tile[TILE_DIM][TILE_DIM];

  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     tile[threadIdx.y+j][threadIdx.x] = idata[(y+j)*width + x];

  __syncthreads();

  x = blockIdx.y * TILE_DIM + threadIdx.x;  // transpose block offset
  y = blockIdx.x * TILE_DIM + threadIdx.y;

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     odata[(y+j)*width + x] = tile[threadIdx.x][threadIdx.y + j];
}
```



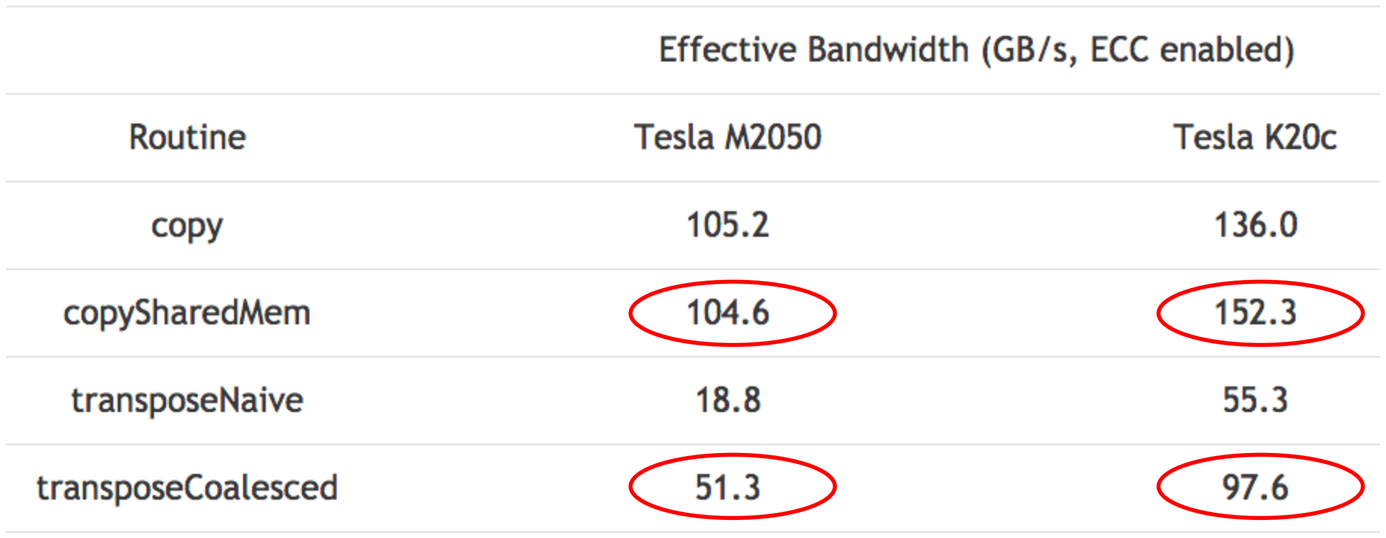
- Both reads from idata and writes to odata are coalesced again

	Effective Bandwidth (GB/s, ECC enabled)	
Routine	Tesla M2050	Tesla K20c
copy	105.2	136.0
transposeNaive	18.8	55.3
transposeCoalesced	51.3	97.6

Better but not good enough

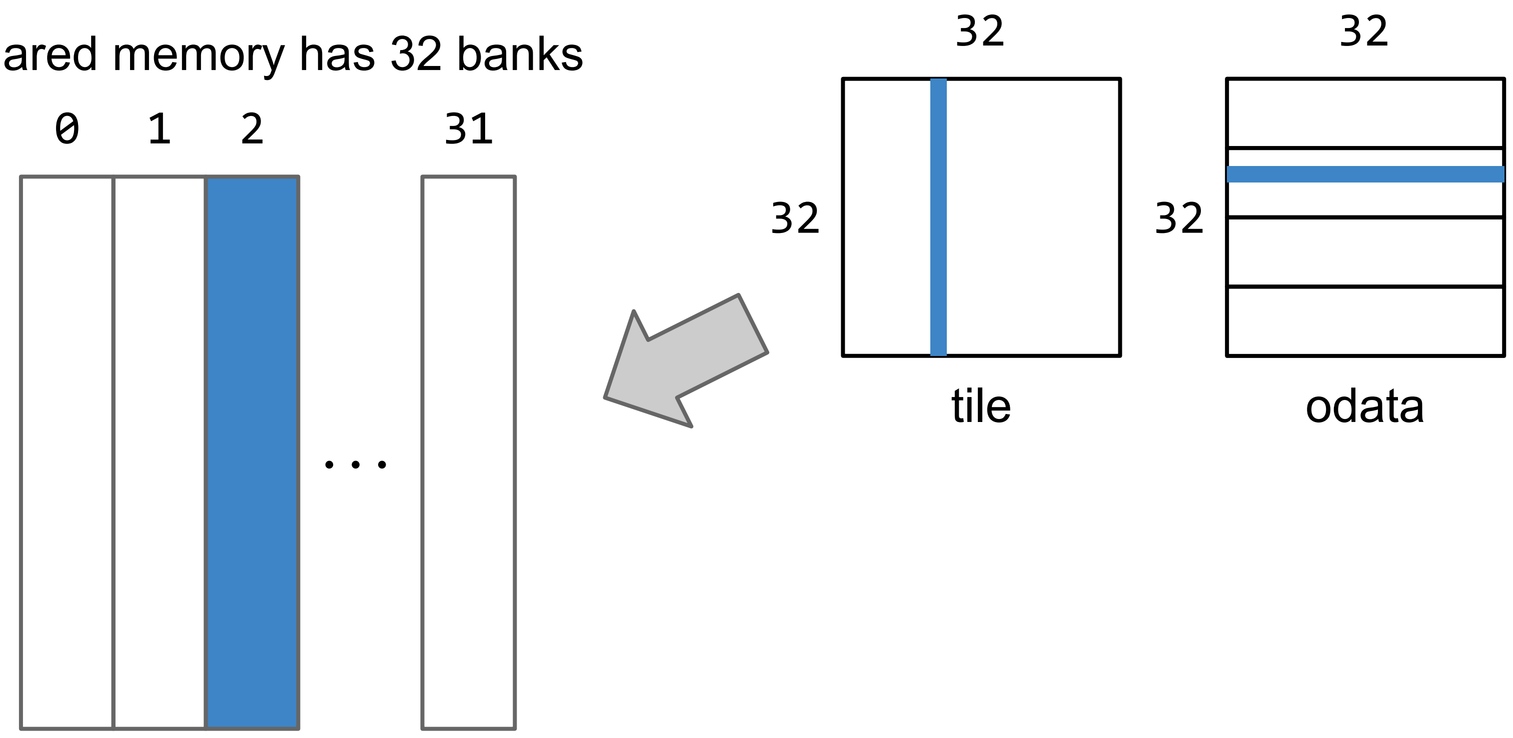
- Overhead associated with using shared memory ?
- Synchronization barrier ?

```
__global__ void copySharedMem(float *odata, const float *idata)
{
  __shared__ float tile[TILE_DIM * TILE_DIM];

  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     tile[(threadIdx.y+j)*TILE_DIM + threadIdx.x] = idata[(y+j)*width + x];

  __syncthreads();

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     odata[(y+j)*width + x] = tile[(threadIdx.y+j)*TILE_DIM + threadIdx.x];
}
```



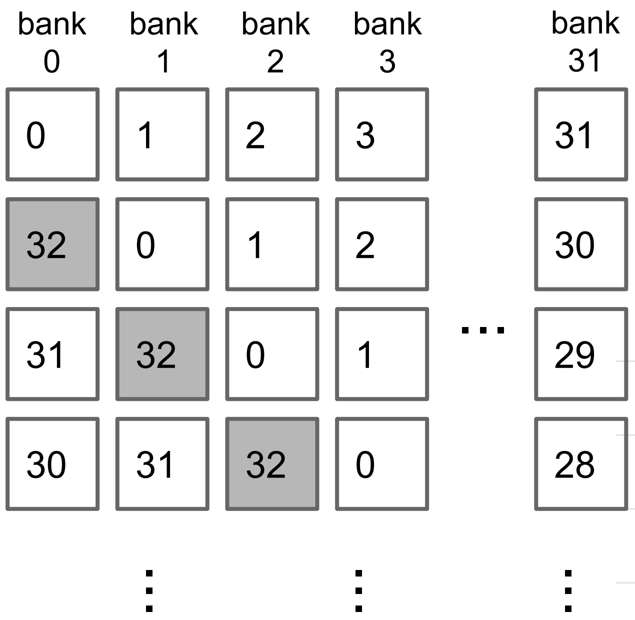
	Effective Bandwidth (GB/s, ECC enabled)	
Routine	Tesla M2050	Tesla K20c
copy	105.2	136.0
copySharedMem	104.6	152.3
transposeNaive	18.8	55.3
transposeCoalesced	51.3	97.6

Shared memory has 32 banks

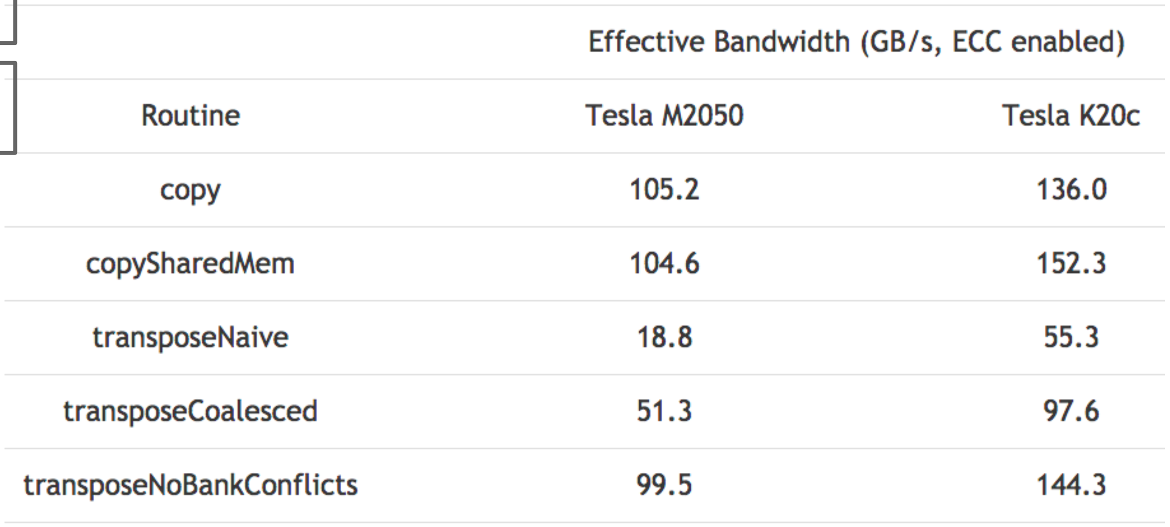


Avoid shared memory bank conflicts

```
__shared__ float tile[TILE_DIM][TILE_DIM+1];
```



Pad arrays to avoid shared memory bank conflicts



	Effective Bandwidth (GB/s, ECC enabled)	
Routine	Tesla M2050	Tesla K20c
copy	105.2	136.0
copySharedMem	104.6	152.3
transposeNaive	18.8	55.3
transposeCoalesced	51.3	97.6
transposeNoBankConflicts	99.5	144.3

Histogram calculation in CUDA

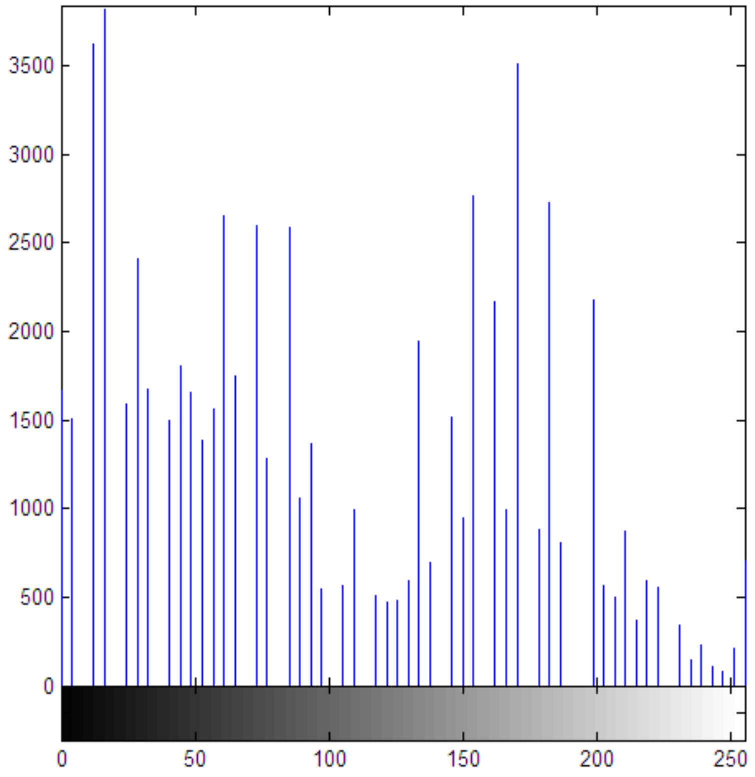
- http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/histogram64/doc/histogram.pdf
- November 2007

Hardware 8 years ago

- G80, G8x, Tesla architecture
- Compute capability 1.0, 1.1
- No atomic shared memory operations
- Maximum amount of shared memory per thread block is 16KB
- Number of shared memory banks is 16
- A single thread block should contain 128-256 threads for efficient execution
- http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities

Histogram

Show the frequency of occurrence of each data element (pixel intensity in image histogram)



[0, 255] 256 bins

http://www.mathworks.com/help/images/contrast-adjustment.html

Sequential program

```
for(int i = 0; i < BIN_COUNT; i++)
    result[i] = 0;

for(int i = 0; i < dataN; i++)
    result[data[i]]++;
```

Parallel program

Naive strategy: all the threads updating a single histogram in global memory using atomicAdd

Strategy:

1. Divide the input array between threads
2. Process the sub-arrays by each dedicated thread and store the result into a certain number of sub-histograms
3. Merge all the sub-histograms into a single histogram

Two strategies

- histogram64
  - per-thread sub-histogram with 64 single-byte bin counters
  - A single thread can process at most 255 bytes input
  - Bank conflicts in shared memory
- histogram256
  - per-warp sub-histogram with 256 4-byte (unsigned int) bin counters
  - Software-implemented atomic additions in shared memory

histogram64

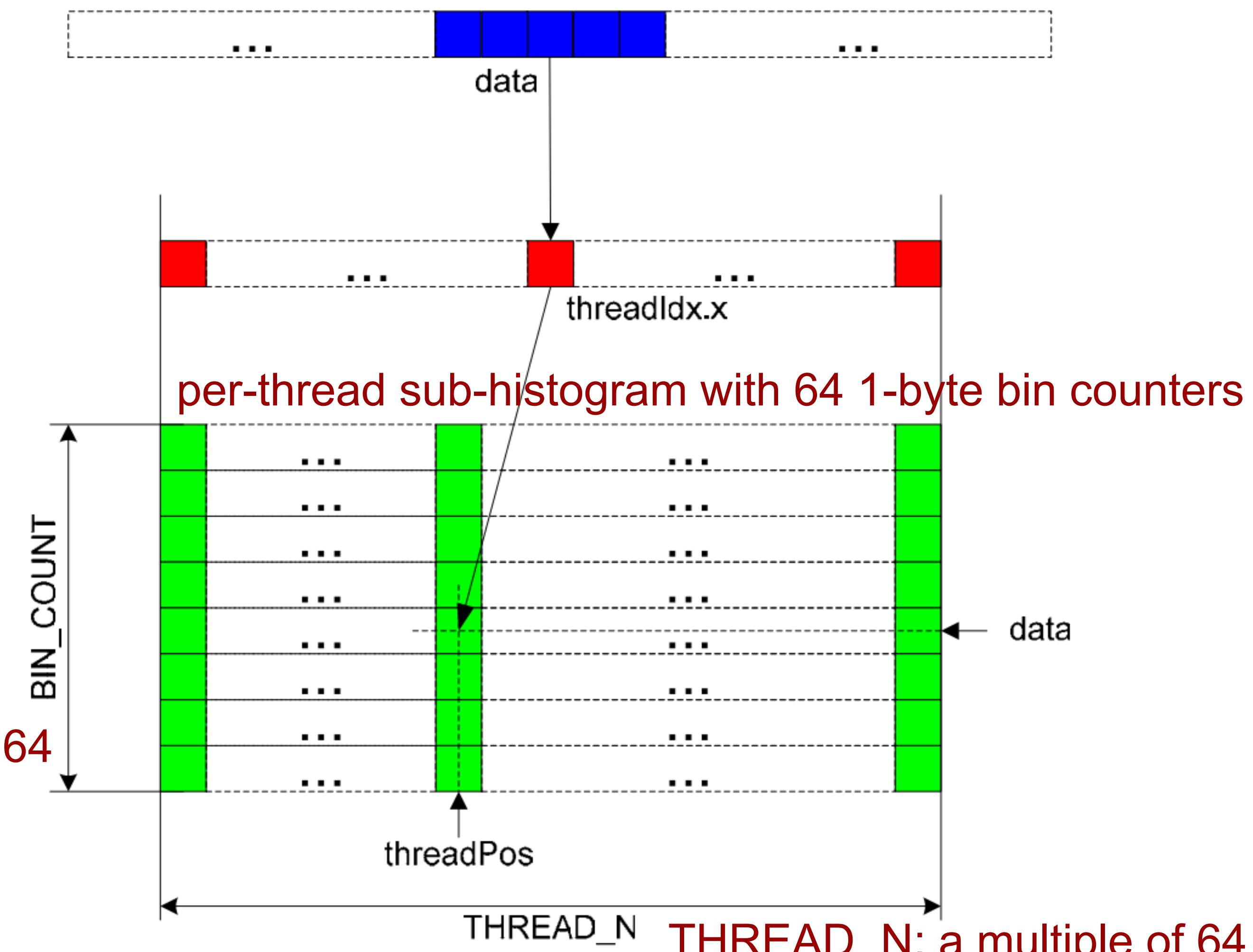
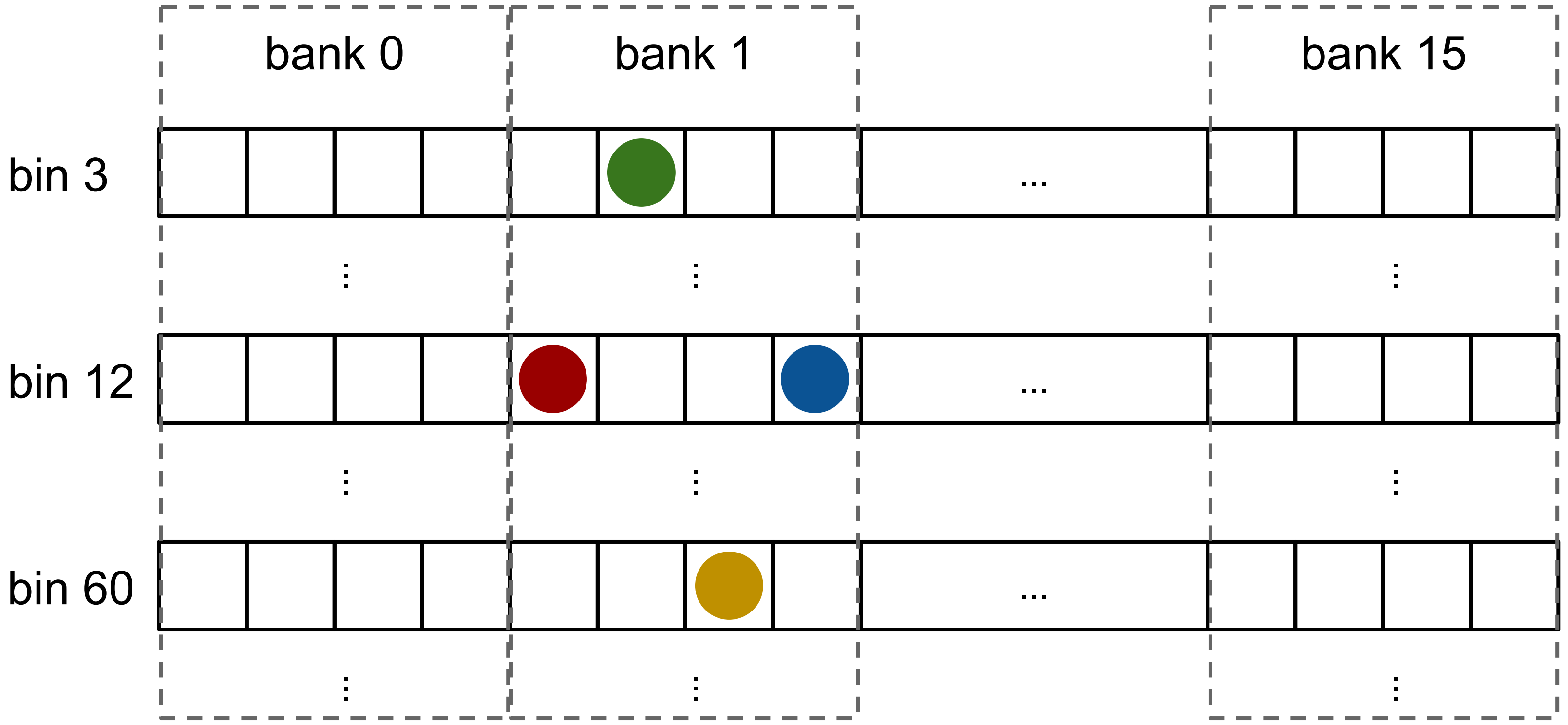

...
data
...
...
...
threadIdx.x
per-thread sub-histogram with 64 1-byte bin counters
BIN_COUNT
BIN_COUNT = 64
data
threadPos
THREAD_N
THREAD_N: a multiple of 64

Figure 1. s_Hist[] array layout for histogram64.

4-way bank conflicts

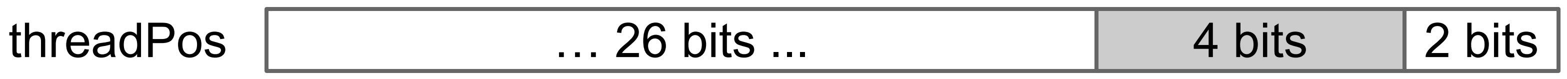


If threadPos = threadIdx.x, thread 4 increments `s_Hist[4 + 12 * THREAD_N]`
thread 5 increments `s_Hist[5 +  3 * THREAD_N]`
thread 6 increments `s_Hist[6 + 60 * THREAD_N]`
thread 7 increments `s_Hist[7 + 12 * THREAD_N]`

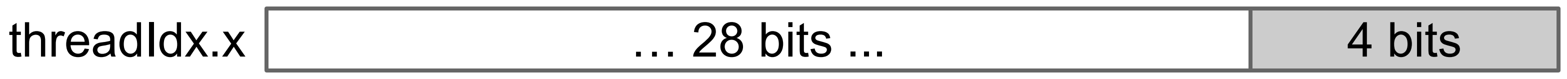
Bank number

bank number

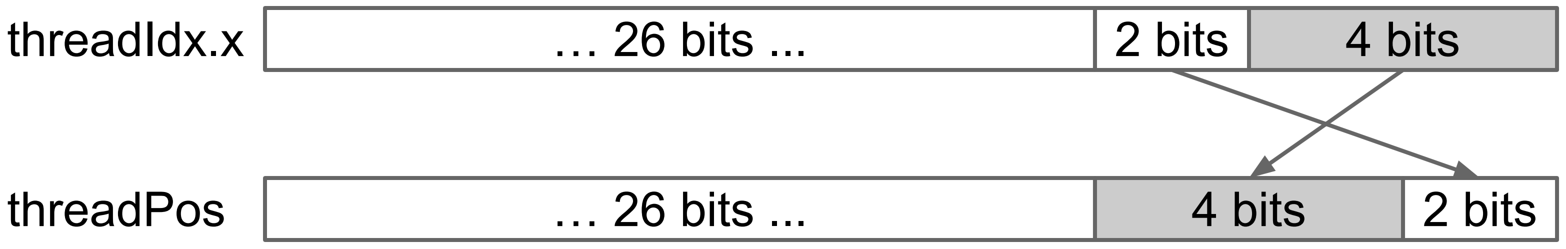
```
=(threadPos + bin * THREAD_N) / 4 % 16
=(threadPos / 4) % 16
```

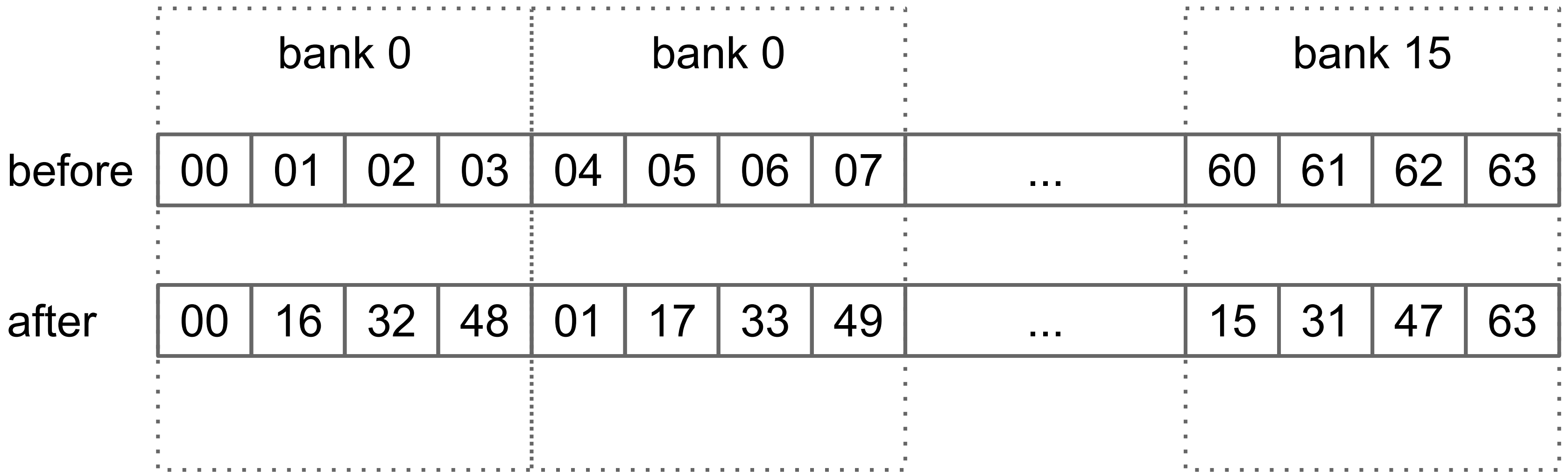


For threads within half-warp (16 threads), only the last 4 bits of threadIdx.x are different



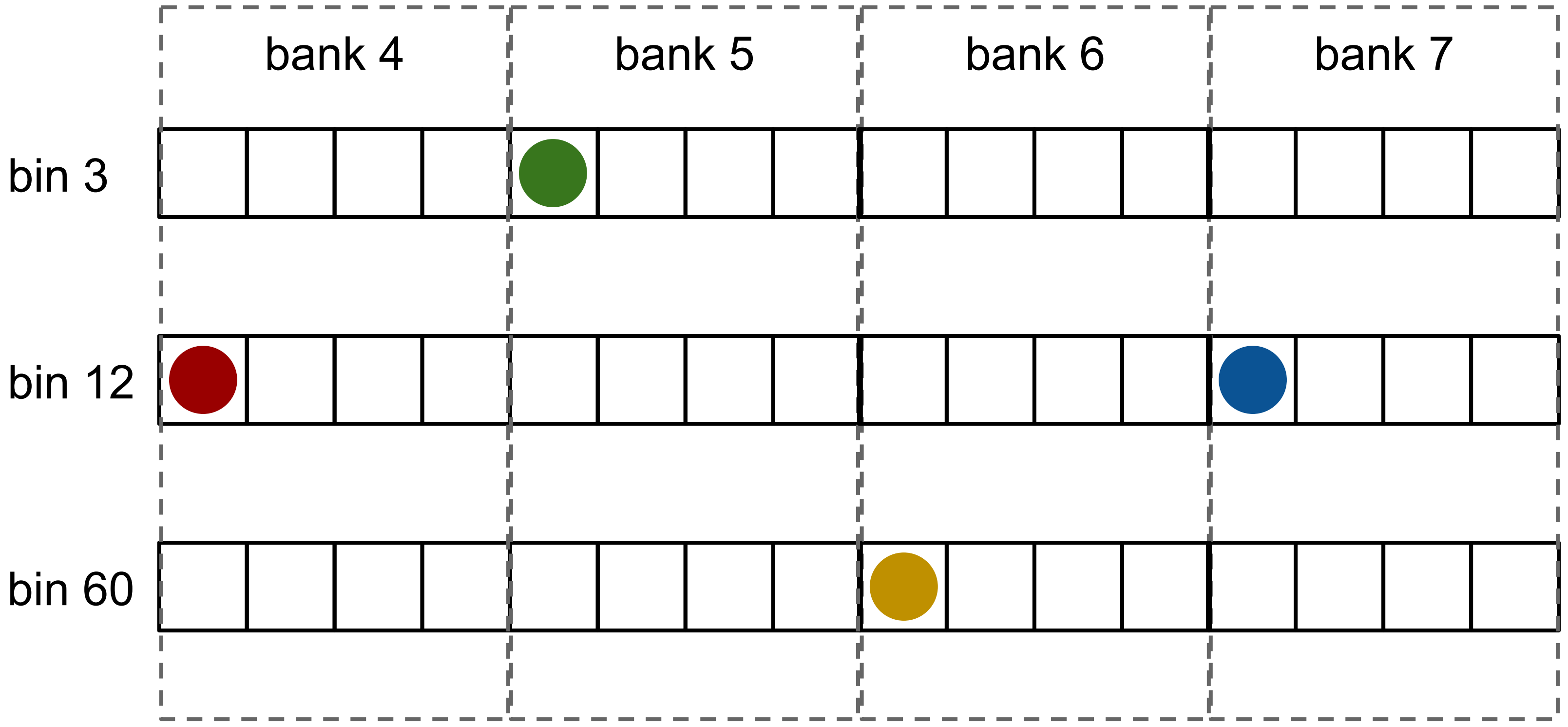
Shuffle [5:4] and [3:0] bit ranges





bank-conflict-free access

Threads within half-warp will access 16 different banks, e.g., thread 4, 5, 6, 7



per-block sub-histogram shift starting positions

- One thread per bin (row in `s_Hist[]`)
- If the threads within half-warp all start from the first per-thread sub-histogram, there will be 16-way bank conflicts
- bank number
  `= (accumPos + threadIdx.x * THREAD_N) / 4 % 16`
- `accumPos = (threadIdx.x % 16) * 4`
- Threads within half-warp start from 16 different banks

bank-conflict-free access

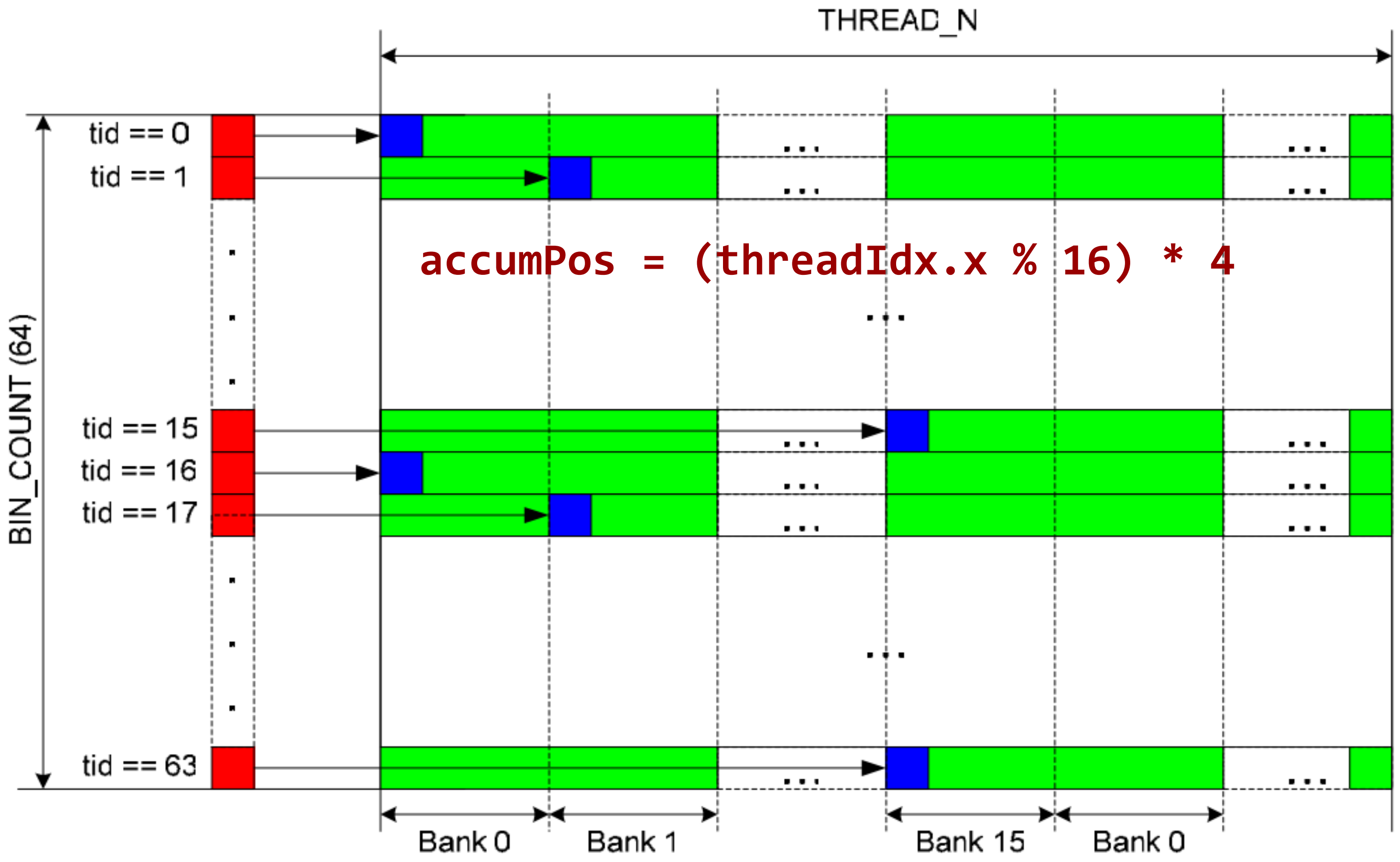


Figure 2. Shifting start accumulation positions (blue) in order to avoid bank conflicts during the reduction stage in histogram64.

Merge sub-histograms

```
const int value = threadIdx.x;
```

value: bin index; one bin per thread

```
#if ATOMICS
    atomicAdd(d_Result + value, sum);
#else
    d_Result[IMUL(BIN_COUNT, blockIdx.x) + value] = sum;
#endif
```

`d_Result[]`: histogram in global memory

`d_Result[]`: per-block sub-histograms in global memory

histogram256

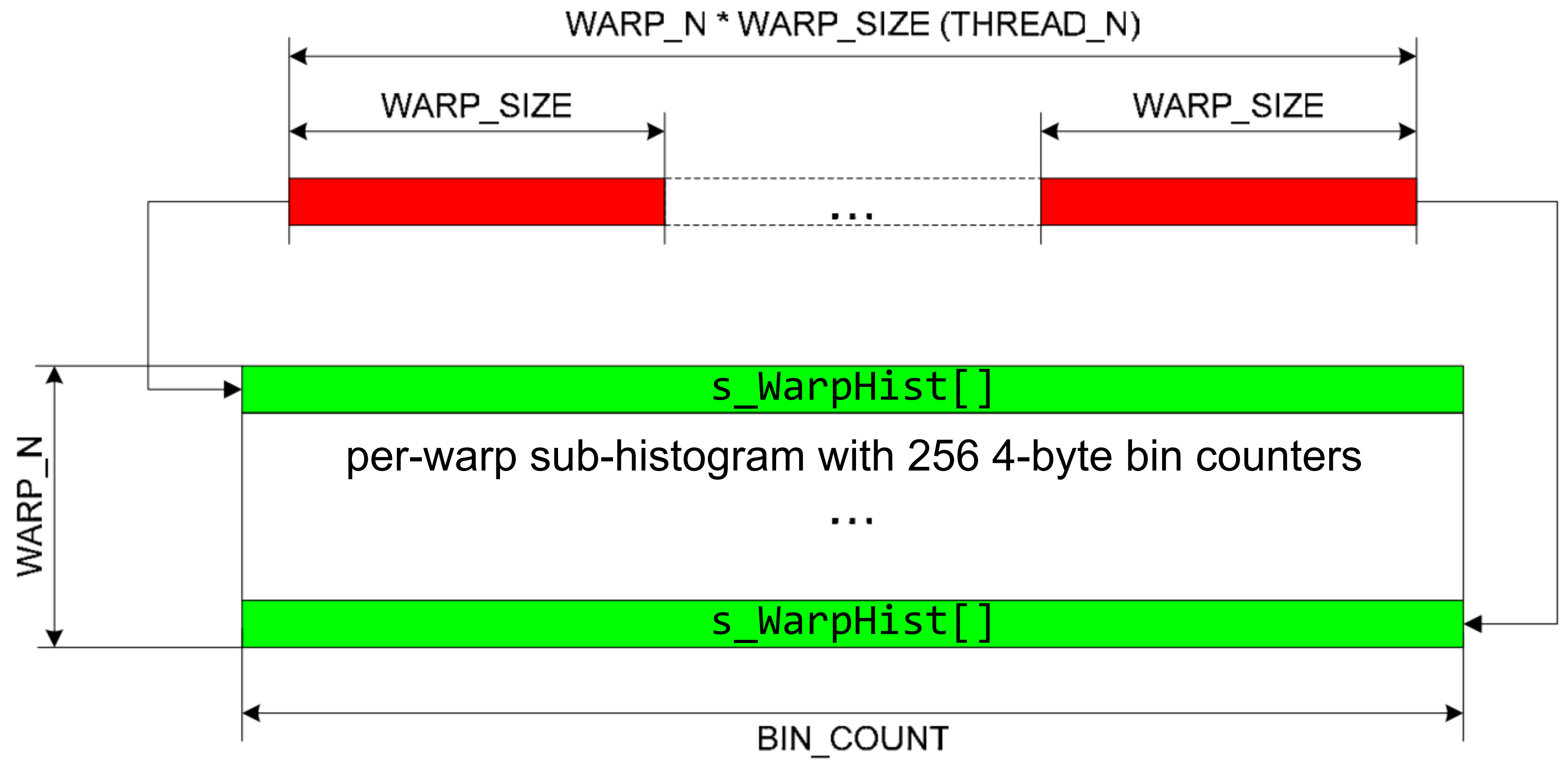

WARP_N * WARP_SIZE (THREAD_N)
WARP_SIZE
WARP_SIZE
...
s_WarpHist[]
per-warp sub-histogram with 256 4-byte bin counters
...
s_WarpHist[]
WARP_N
BIN_COUNT

Figure 3. s_Hist[] layout for histogram256.

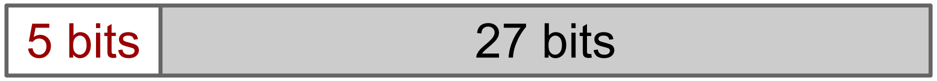
Intra-warp shared memory collisions

Each group of 32 threads (warp) shares one `s_WarpHist[]`, thus two or more threads may collide on the same bin counter

e.g., thread 0, 3, 12 read 3 different pixels with the same intensity `data` (within 0 .. 255 range), then they try to increment `s_WarpHist[data]` at the same time

Software implementation of atomic shared memory operations

```
__device__ void addData256(
      volatile unsigned int *s_WarpHist,
      unsigned int data,
      unsigned int threadTag threadIdx.x % 32 << 27   [0, 31] 5 bits
){
    unsigned int count;

    do{
        count = s_WarpHist[data] & 0x07FFFFFFU;
        count = threadTag | (count + 1);
        s_WarpHist[data] = count;
    }while(s_WarpHist[data] != count);
}
```



The hardware performs shared memory write combing, which accepts 'count' from an arbitrary thread and rejects all the others.

The first 5 bits records the last writer.

Example: thread 0, 3, 12 increment `s_WarpHist[data]`

```
s_WarpHist[data] = (31 << 27) | 10
```

thread 0	thread 3	thread 12
`read  (31 << 27) | 10`	`read  (31 << 27) | 10`	`read  (31 << 27) | 10`
`write  (0 << 27) | 11`	`write  (3 << 27) | 11`	`write (12 << 27) | 11`
`read   (3 << 27) | 11`	`read   (3 << 27) | 11`	`read   (3 << 27) | 11`
`write  (0 << 27) | 12`		`write (12 << 27) | 12`
`read   (0 << 27) | 12`		`read   (0 << 27) | 12`
		`write (12 << 27) | 13`
		`read  (12 << 27) | 13`

```
s_WarpHist[data] = (12 << 27) | 13
```

Merge sub-histograms

```
for(int pos = threadIdx.x; pos < BIN_COUNT; pos += blockDim.x)
{
    unsigned int sum = 0;

    for(int base = 0; base < BLOCK_MEMORY; base += BIN_COUNT)
        sum += s_Hist[base + pos] & 0x07FFFFFFU;

#if ATOMICS
        atomicAdd(d_Result + pos, sum);
#else
        d_Result[IMUL(BIN_COUNT, blockIdx.x) + pos] = sum;
#endif
}
```

One bin (one column of `s_Hist[]`) per thread

sum: a bin counter of per-block sub-histogram

`d_Result[]`: histogram in global memory

`d_Result[]`: per-block sub-histograms in global memory