

# CS 623: Performance Bottlenecks in CUDA

**Swarnendu Biswas**

Department of Computer Science and Engineering,  
Indian Institute of Technology Kanpur

Sem 2026-27-I



# Key Ideas for Extracting Performance

- Desired application characteristics for device execution
  - (i) Large data-parallel computation
  - (ii) Complex computation kernel to justify the H2D and D2H data movement costs
    - Keep data on the device to avoid repeated transfers
- Try and reduce resource consumption (check with `-res-usage`)
- Exploit SIMT, reduce thread divergence in a warp
- Strive for good locality, use tiling to exploit shared memory
  - ▶ Improve throughput by reducing global memory traffic
  - ▶ Copy blocks of data from global memory to shared memory and operate on them (e.g., matrix multiplication kernel)
- Optimize memory accesses
  - Global memory memory coalescing
  - Shared memory avoid bank conflicts

# What can we say about this code?

```
__global__ void dkernel(float *vector, int vectorsize) {  
    int id = blockIdx.x * blockDim.x + threadIdx.x;  
    switch (id) {  
        case 0: vector[id] = 0; break;  
        case 1: vector[id] = vector[id] * 10; break;  
        case 2: vector[id] = vector[id - 2]; break;  
        case 3: vector[id] = vector[id + 3]; break;  
        ...  
        case 31: vector[id] = vector[id] * 9; break;  
    }  
}
```

# Dealing with Thread Divergence

Thread divergence renders execution sequential because the SIMD hardware takes multiple passes through the divergent paths

```
if (threadIdx.x / WARP_SIZE > 2) {}
```

Condition evaluating to different truth values may not be bad

- Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path

```
if (threadIdx.x > 2) {}
```

Conditions evaluating to different truth values for threads in a warp is bad

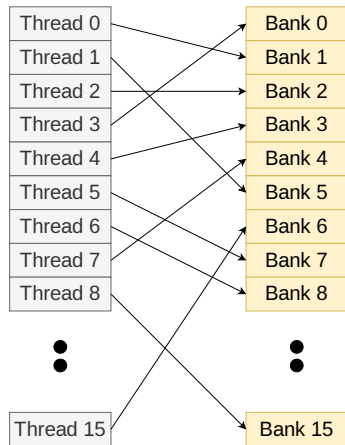
- Creates two different control paths for threads in a block; branch granularity < warp size; threads 0 and 1 follow different path than the rest of the threads in the first warp

# Parallel Memory Architecture

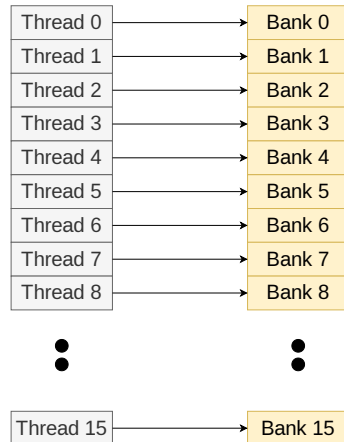
- In a parallel architecture, many threads access memory
- Memory is divided into **banks** to achieve high bandwidth
  - ▶ Each bank can service one address per cycle
  - ▶ A memory can service as many simultaneous accesses as it has banks
- Multiple simultaneous accesses to a bank result in a bank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

# Example of Bank Addressing

## Random access

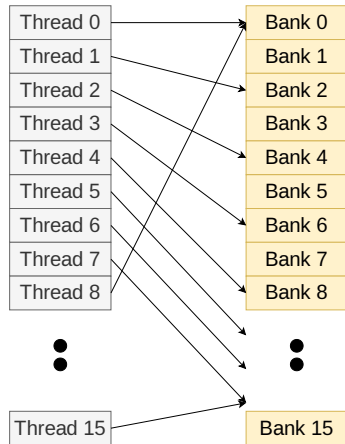


## Linear access, stride=1

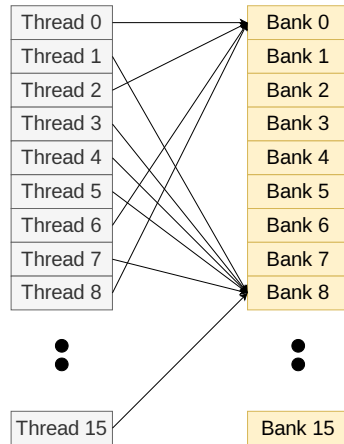


# Example of Bank Addressing

Linear access, stride=2



Linear access, stride=8



# Bank Conflicts in Shared Memory

NVIDIA GPUs have 32 shared memory banks per SM

- Each 4-byte shared memory word  $W$  is mapped to bank  $(\&W/4)\%32$

**Fast case**

- If all threads of a warp access different banks, there is no bank conflict
- If all threads of a warp access an identical address, there is no bank conflict (broadcast)

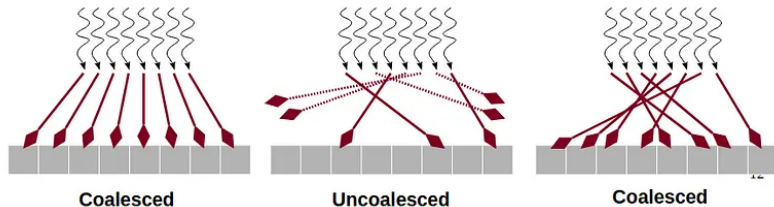
**Slow case**

- Bank conflict arises when multiple threads access the same bank
  - ▶ Must serialize the accesses
- Cost = max # of simultaneous accesses to a single bank

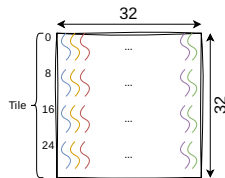
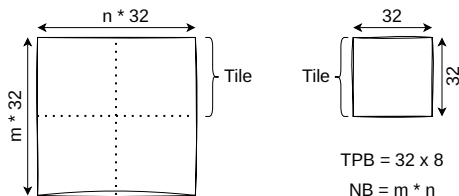
Give low priority to fix low-degree bank conflicts since resolving it will increase instructions

# Memory Coalescing

- Issuing a memory instruction from a single warp can generate up to 32 data cache accesses
- Coalescing reduces the number of memory requests by merging accesses from multiple lanes into cache-line-sized chunks when there is spatial locality across the warp
  - ▶ Coalesced memory accesses imply a warp accesses adjacent data in a cache line
  - ▶ In the best case, this results in one memory transaction
- Uncoalesced memory accesses imply a warp accesses scattered data in different cache lines leading to memory divergence
  - ▶ This may result in 32 different memory transactions



# Copying a Matrix



Each  $32 \times 8$  thread block copies a  $32 \times 32$  tile

```
__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) // BLOCK_ROWS is 8  
        odata[(y+j)*width + x] = idata[(y+j)*width + x];  
}
```

reads from idata and writes to odata are coalesced

# 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];  
}
```

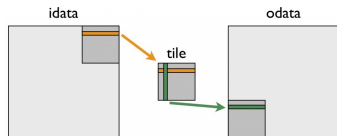
reads from idata are coalesced, but  
writes to odata have a stride of 1024

# Optimizing Matrix Transpose

```
__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];  
}
```

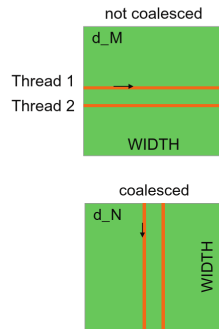


transpose-nvidia.cu



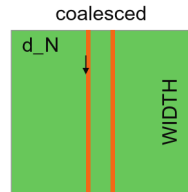
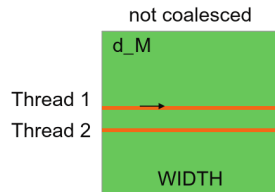
# Matrix Multiplication Example

```
__global__ void matmulKernel(float* A, float* B, float* C) {  
    int row = blockIdx.y * blockDim.y + threadIdx.y;  
    int col = blockIdx.x * blockDim.x + threadIdx.x;  
    float tmp = 0;  
    if (row < N && col < N) {  
        // Each thread computes one element of the matrix  
        for (int k = 0; k < N; k++) {  
            tmp += A[row * N + k] * B[k * N + col];  
        }  
        C[row * N + col] = tmp;  
    }  
}
```

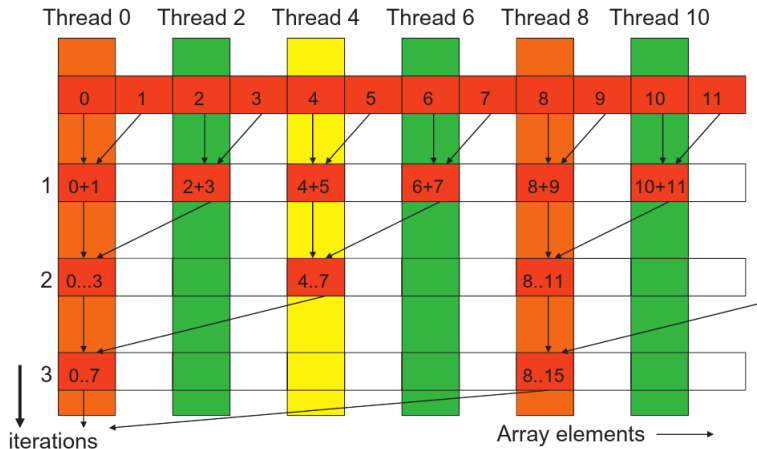


# Optimizing Global Memory Accesses

- Try to ensure that memory requests from a warp can be coalesced
  - ▶ Using optimizations like tiling to make use of the faster shared memory
  - ▶ Stride-one access across threads in a warp is good
  - ▶ Use structure of arrays rather than array of structures



# Implementing a Reduction Kernel in CUDA



# Reduction Kernel

```
__global__ void reduction1(const uint64_t *d_input, uint64_t *d_output) {
    uint64_t tid = threadIdx.x;
    uint64_t pindex = blockIdx.x * blockDim.x + threadIdx.x;
    __shared__ uint64_t partialSum[TPB];
    partialSum[tid] = d_input[pindex];
    __syncthreads();
    unsigned int t = threadIdx.x;
    for (unsigned int stride = 1; stride < blockDim.x; stride *= 2) {
        if (t % (2*stride) == 0)
            partialSum[t] += partialSum[t+stride];
        __syncthreads();
    }
    if (tid == 0) {
        d_output[blockIdx.x] = partialSum[tid];
    }
}
```

only even threads  
are active

# Possible Optimizations on the Reduction Kernel



Optimizing Parallel Reduction in CUDA

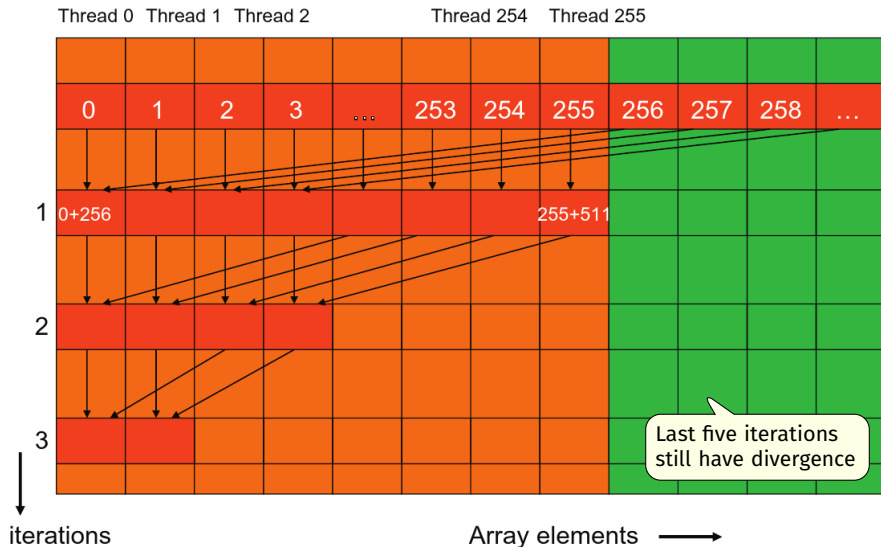
Mark Harris  
NVIDIA Developer Technology

- (i) basic implementation with modulo operator
- (ii) strided access starting with each thread accessing two adjacent locations
- (iii) strided access with reversed loop index
- (iv) halve the number of threads
- (v) unroll the last few loop iterations and avoid synchronization
- (vi) ...

# Reduction Kernel

```
__global__ void reduction3(const uint64_t *d_input, uint64_t *d_output) {
    uint64_t tid = threadIdx.x;
    uint64_t pindex = blockIdx.x * blockDim.x + threadIdx.x;
    __shared__ uint64_t partialSum[TPB];
    partialSum[tid] = d_input[pindex];
    __syncthreads();
    for (uint64_t stride = (blockDim.x >> 1); stride >= 1; stride >>= 1) {
        if (tid < stride) {
            partialSum[tid] += partialSum[tid + stride];
        }
        __syncthreads();
    }
    if (tid == 0) {
        d_output[blockIdx.x] = partialSum[tid];
    }
}
```

# Execution of the Revised Reduction Kernel



# Avoid Divergence in the Last Few Iterations

```
for (unsigned int stride = blockDim.x/2; stride > 32; stride /= 2) {  
    if (t < stride)  
        partialSum[t] += partialSum[t+stride];  
    __syncthreads();  
}  
if (tid < 32) {  
    partialSum[tid] += partialSum[tid+32];  
    partialSum[tid] += partialSum[tid+16];  
    partialSum[tid] += partialSum[tid+8];  
    partialSum[tid] += partialSum[tid+4];  
    partialSum[tid] += partialSum[tid+2];  
    partialSum[tid] += partialSum[tid+1];  
}
```

# References

---



NVIDIA Corporation. CUDA C++ Programming Guide, Release 12.9.2.



NVIDIA Corporation. CUDA C++ Best Practices Guide, Release 12.9.