# ECE 587 – Hardware/Software Co-Design
# Lecture 20 CUDA IV: Warp Execution

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

March 25, 2026

# Outline

# Reading Assignment

- This lecture: CUDA Warp Execution
- Next lecture: RISC-V and Chipyard

# Outline

Parallel Reduction

Optimizations

# Parallel Reduction

```
sum = 0;
// Summation as reduction using two levels of loops.
// Run outer loop on CPU for simplicity.
for (int ii = 0; ii < N; ii += M) {
  // How to parallelize the inner loop with CUDA?
  partial = 0;
  for (int i = ii; i < ii+M; ++i) {
    partial += input[i];
  }
  sum += partial;
}
```

- Compute reduction, e.g. summation, in parallel.
  - Partition the input into `N/M` groups of length `M`.
  - Run reduction on each group in parallel to obtain partial sum.
  - Run reduction on the partial sums.
- With tree reduction, only $\log_2 M$ parallel steps are needed to compute the partial sum from a group of `M` numbers.
  - Example: for `M=128`, only 7 steps are needed.
  - How can we implement this idea in CUDA?

# Parallel Reduction Kernel

```
__global__ void sum_mod(const float* input, float* partial, int N) {
  extern __shared__ float scratch[];
  const unsigned int M = blockDim.x;
  const unsigned int tid = threadIdx.x;
  const unsigned int i = blockIdx.x * M + tid;

  scratch[tid] = (i < N) ? input[i] : 0.0f;
  __syncthreads();
  ...
  if (tid == 0) partial[blockIdx.x] = scratch[0];
}
```

- Each block of `M` threads reduces one chunk of `M` input values.
- First, each thread loads one input value into shared memory.
  - Use a barrier to make sure all data are ready.
- At the end, only thread `0` writes the partial sum back to the global memory.

# Modulo-Based Iterations and Warp Divergence

```
// sum_mod
for (unsigned int stride = 1; stride < M; stride *= 2) {
  if ((tid % (2 * stride) == 0) && (tid + stride < M)) {
    scratch[tid] += scratch[tid + stride];
  }
  __syncthreads();
}
```

- Thread activation is controlled by the modulo condition.
  - Active threads take time to update `scratch[tid]`.
- Each active thread updates the left element of one pair.
  - The update has to be done in place since `scratch` is used by other active threads as well.
  - Use a barrier to ensure data are ready before the next iteration.
- What happens to inactive threads (`if` condition is `false`)?
  - If all threads in the same warp are inactive, all of them proceed to `__syncthreads()` and wait there immediately.
  - Warp Divergence: if a warp contains both active and inactive threads, then effectively inactive threads have to wait for active ones to finish updating `scratch[tid]`.

# Warp Divergence Example

- `M=128` gives $128/32 = 4$ warps per block.
- Active threads shrink by stride:
  - `stride=1`: 64 threads 0,2,4,...,126 are active
  - `stride=2`: 32 threads 0,4,8,...,124 are active
  - `stride=4`: 16 threads 0,8,16,...,120 are active
  - `stride=8`: 8 threads 0,16,32,48,64,80,96,112 are active
  - `stride=16`: 4 threads 0,32,64,96 are active
  - `stride=32`: 2 threads 0,64 are active
  - `stride=64`: 1 thread 0 is active
- Warp divergence for stride $\leq 16$
  - All 4 warps have both active and inactive threads.
  - CUDA cores with inactive threads are idling, wasting resources.
- Pack all active threads into as few warps as possible.
  - Warps of inactive threads proceed to the barrier immediately.
  - We cannot completely remove warp divergence for stride $\geq 4$ because there is less than 32 active threads.

# Pack Active Threads

```
// sum_idx
for (unsigned int stride = 1; stride < M; stride *= 2) {
  const unsigned int index = 2 * stride * tid;
  if (index + stride < M) {
    scratch[index] += scratch[index + stride];
  }
  __syncthreads();
}
```

- The first `M/(2*stride)` threads should be active.
- For `M=128`,
  - `stride=1`: `tid=0` to `63`
  - `stride=2`: `tid=0` to `31`
  - `stride=4`: `tid=0` to `15`
  - `stride=8`: `tid=0` to `7`
  - `stride=16`: `tid=0` to `3`
  - `stride=32`: `tid=0` or `1`
  - `stride=64`: `tid=0`
- As few warps as possible have active threads now.

# Shared Memory Bank Conflicts

- CUDA hardware organizes shared memory into 32 banks to support concurrent accesses from the 32 threads in a warp.
  - Ideally, no more than 1 address per bank is requested and all requests complete and 32 threads are served concurrently.
- Bank conflicts
  - If at least 2 addresses are requested in a bank, they must be done one after another.
  - At least double the shared memory latency for all threads in the warp since they have to execute the same instruction.
- For example, consider `M=128` and `stride=1`,
  - First warp of threads `0` to `31` access `scratch[0]`, `scratch[2]`, ..., `scratch[62]` together, and `scratch[1]`, `scratch[3]`, ..., `scratch[63]` together.
  - Both groups contain 32 different addresses likely from 16 banks, leading to bank conflicts.
  - Larger strides do not help to remove bank conflicts.

# Resolve Bank Conflicts via Sequential Accesses

```
// sum_seq
for (unsigned int stride = M / 2; stride > 0; stride /= 2) {
  if (tid < stride) {
    scratch[tid] += scratch[tid + stride];
  }
  __syncthreads();
}
```

- Same active threads as `sum_idx` although `stride` updates differently from `M/2` down to `1`.
- Allow threads to access shared memory indexed by their id.
  - Threads in a warp now access consecutive addresses.
  - They won't fall into the same bank so there is no bank conflict.
- For example, consider `M=128` and `stride=64`,
  - First warp of threads `0` to `31` access `scratch[0]`, `scratch[1]`, ..., `scratch[31]` together, and `scratch[64]`, `scratch[65]`, ..., `scratch[95]` together.
  - Both groups access 32 consecutive addresses from shared memory and there is no bank conflict.

# Experimental Results: One Reduction Round

Table: kernel time (ms), warm-up + single run, one reduction round on one Tesla T4 GPU, data transfer time excluded, CPU finishes final sum; $N$ is in units of $2^{20}$.

| kernel / M | N=16 | N=64 | N=256 | N=1024 |
|---|---|---|---|---|
| sum_mod, 256 | 2.10 | 9.76 | 22.62 | 74.22 |
| sum_idx, 256 | 1.75 | 8.00 | 14.06 | 53.13 |
| sum_seq, 256 | 1.35 | 6.25 | 10.40 | 42.46 |
| sum_mod, 1024 | 3.56 | 7.66 | 26.81 | 109.24 |
| sum_idx, 1024 | 2.47 | 5.35 | 18.43 | 73.48 |
| sum_seq, 1024 | 2.06 | 4.50 | 15.56 | 61.37 |

- With the same `N` and `M`,
  - All three kernels complete the same amount of computations.
  - `sum_idx` improves over `sum_mod` by reducing warp divergence.
  - `sum_seq` improves over `sum_idx` by optimizing shared memory access pattern to avoid bank conflicts.
- Note that for the same `N`, larger `M` lead to more reduction steps per block.

# Outline

# Loop Overheads

```
// sum_seq
for (unsigned int stride = M / 2; stride > 0; stride /= 2) {
  if (tid < stride) {
    scratch[tid] += scratch[tid + stride];
  }
  __syncthreads();
}
```

- So far we ignore cost of loop operations.
  - Usually there are a lot more other operations than updating loop variables and taking branches.
  - These operations need to access shared or global memory.
  - Overheads of loop operations are hidden by communications.
- But for the last few iterations of `sum_seq`,
  - Very few instructions and memory transactions in loop body.
  - The overheads of loop operations cannot be ignored.

# Loop Unrolling

```
// sum_unroll
for (unsigned int stride = M / 2; stride > 32; stride /= 2) {
  if (tid < stride) scratch[tid] += scratch[tid + stride];
  __syncthreads();
}
if (tid < 32) scratch[tid] += scratch[tid + 32]; __syncthreads();
if (tid < 16) scratch[tid] += scratch[tid + 16]; __syncthreads();
if (tid <  8) scratch[tid] += scratch[tid +  8]; __syncthreads();
if (tid <  4) scratch[tid] += scratch[tid +  4]; __syncthreads();
if (tid <  2) scratch[tid] += scratch[tid +  2]; __syncthreads();
if (tid <  1) scratch[tid] += scratch[tid +  1]; __syncthreads();
```

- Unroll a loop to reduce loop overheads.
  - Replace loop variables with constants.
  - Repeat the loop body in the code explicitly.
- Still, the loop may need to iterate a number of rounds that is controlled by other variables.
  - Keep the original loop and modify it for the unrolled part.

# When Only One Warp Remains Active

```
if (tid < 32) scratch[tid] += scratch[tid + 32]; __syncthreads();
if (tid < 16) scratch[tid] += scratch[tid + 16]; __syncthreads();
if (tid <  8) scratch[tid] += scratch[tid +  8]; __syncthreads();
if (tid <  4) scratch[tid] += scratch[tid +  4]; __syncthreads();
if (tid <  2) scratch[tid] += scratch[tid +  2]; __syncthreads();
if (tid <  1) scratch[tid] += scratch[tid +  1]; __syncthreads();
```

- We unroll the reduction loop when only the first warp contains active threads.
- Other warps need to wait on `__syncthreads()` for 6 times.
  - They do nothing else.
  - It doesn't cost much but still wastes resource.
- Threads in one warp always execute the same instruction.
  - If only the threads from the first warp will modify data, then they are synchronized without `__syncthreads()`.
- We can safely remove `__syncthreads()` when the actual work is done entirely within one warp.

# Single Warp Optimization

```
// sum_warp
for (unsigned int stride = M / 2; stride > 32; stride /= 2) {
  if (tid < stride) scratch[tid] += scratch[tid + stride];
  __syncthreads();
}
if (tid < 32) {
  volatile float* vscratch = scratch;
  vscratch[tid] += vscratch[tid + 32];
  vscratch[tid] += vscratch[tid + 16];
  vscratch[tid] += vscratch[tid + 8];
  vscratch[tid] += vscratch[tid + 4];
  vscratch[tid] += vscratch[tid + 2];
  vscratch[tid] += vscratch[tid + 1];
}
```

- Use `volatile` to force compiler to generate code to read and write shared memory in program order.
- Remove individual `tid` check to reduce instruction count.
  - This does change program behavior – not a problem for reduction since `scratch[0]` remains the same.
  - This does incur more additions and shared memory accesses but the benefit outweighs the extra work.

# Experimental Results: Further Optimizations

Table: kernel time (ms), warm-up + single run, one reduction round on one Tesla T4 GPU, data transfer time excluded, CPU finishes final sum; $N$ is in units of $2^{20}$.

| kernel / M | N=16 | N=64 | N=256 | N=1024 |
|---|---|---|---|---|
| sum_seq, 256 | 1.35 | 6.25 | 10.40 | 42.46 |
| sum_unroll, 256 | 1.23 | 5.68 | 9.13 | 38.00 |
| sum_warp, 256 | 0.84 | 3.76 | 6.74 | 27.99 |
| sum_seq, 1024 | 2.06 | 4.50 | 15.56 | 61.37 |
| sum_unroll, 1024 | 1.94 | 4.24 | 14.65 | 57.55 |
| sum_warp, 1024 | 1.35 | 3.00 | 10.36 | 41.01 |

- `sum_unroll` reduces loop overheads.
- `sum_warp` further removes unnecessary barriers.

# Summary

- It is critical to reduce warp divergence for threads running different code paths.
- Pay attention to bank conflicts when threads in a warp access shared memory concurrently.
- Consider loop unrolling to reduce loop overheads, in particular for loops with small body.
- Reason with the computation for additional optimization opportunities.
- Can we build specific hardware to speed up specific computations further?