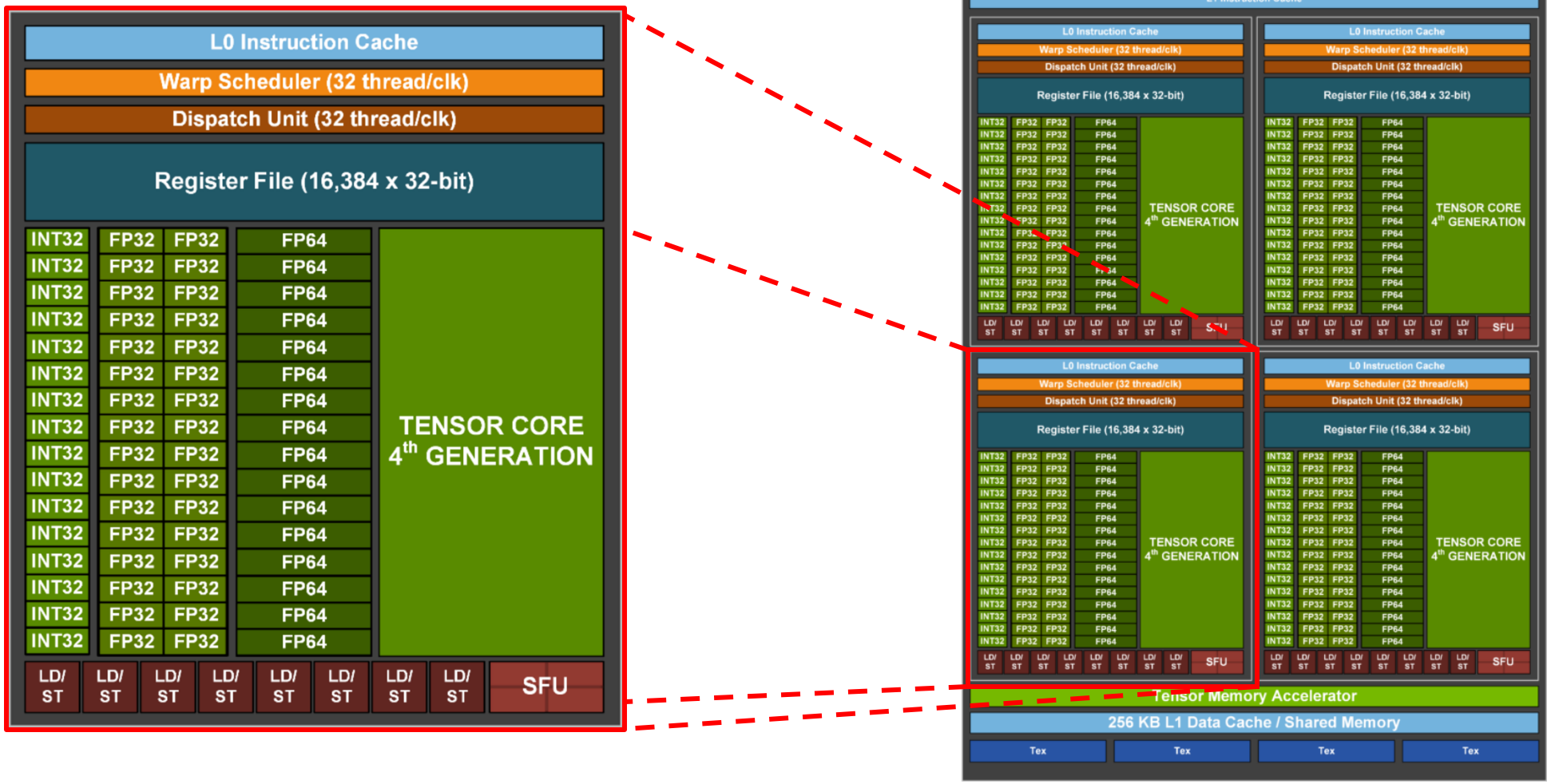
System for Artificial Intelligence

CUDA Programming Case Studies

Siyuan Feng
Shanghai Innovation Institute

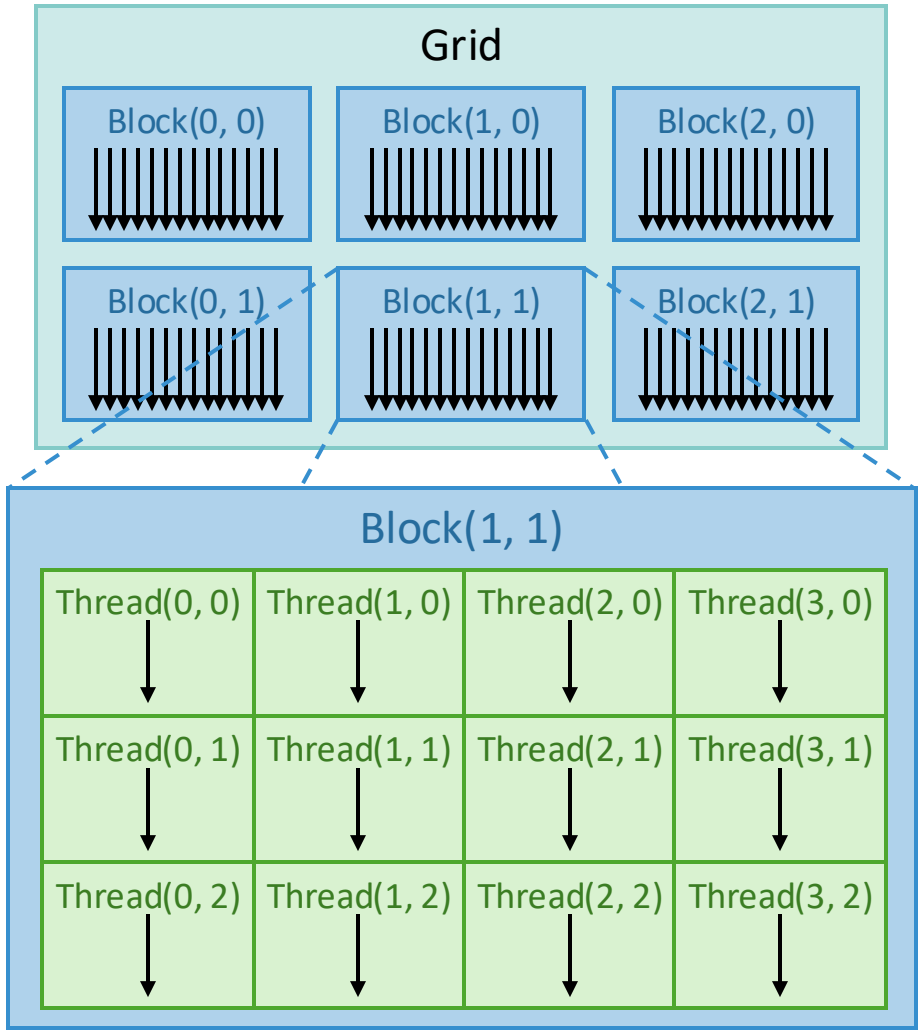
Recap: H100 Streaming Multiprocessor



Recap: CUDA Blocks Map to GPU SM

上海创智学院
Shanghai Innovation Institute

The whole CUDA program runs on whole GPU, while a block runs on a single SM



OUTLINE

01 Convolution 1D

02 Matrix Multiplication

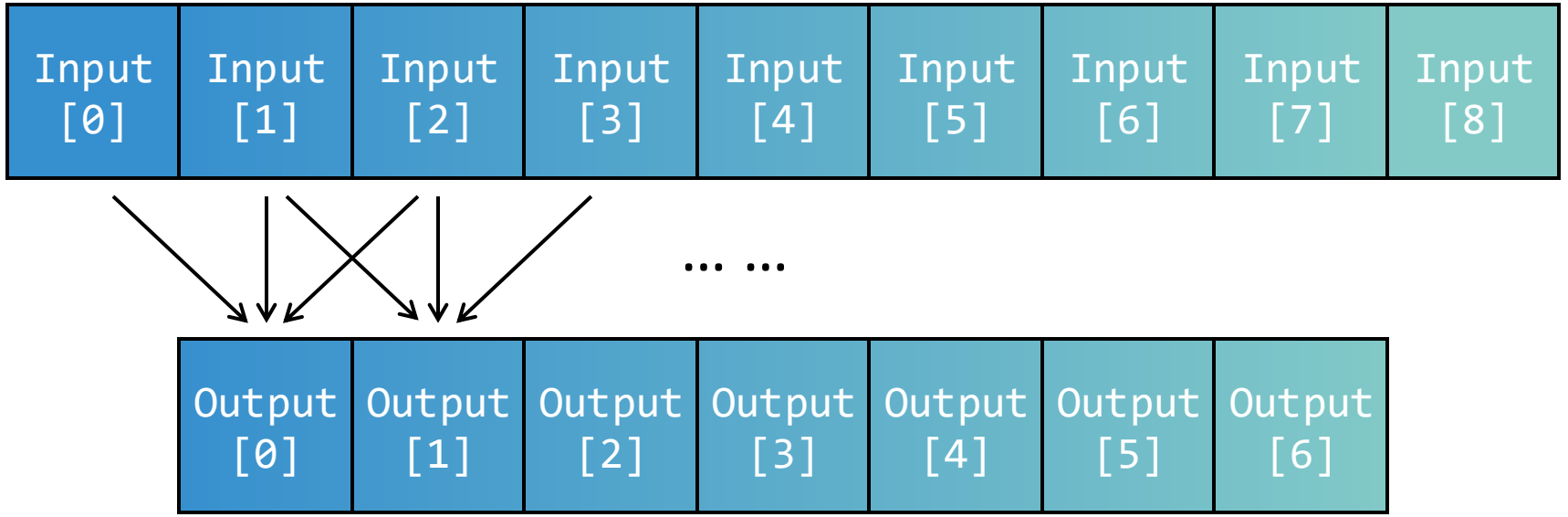
01

Convolution 1D

CUDA Programming Example: 1D Convolution

上海创智学院
Shanghai Innovation Institute

```
output[i] = input[i] + input[i+1] + input[i+2];
```



CUDA 1D Convolution - Naive

上海创智学院 Shanghai Innovation Institute

```
int N = 1024 * 1024, THREADS_PER_BLK = 128
cudaMalloc(&devInput, sizeof(float) * (N + 2));  // allocate
cudaMalloc(&devOutput, sizeof(float) * N);       // allocate

// property initialize contents of devInput here ...

conv1D<<<N/THREADS_PER_BLK, THREADS_PER_BLK>>>(devInput, devOutput, N);
```

Launch
`N/THREADS_PER_BLK` Blocks
`THREADS_PER_BLK` threads

```
__global__ void conv1D(float* input, float* output, int N) {

  int index = blockIdx.x * blockDim.x + threadIdx.x;
  if (index >= N) { return; }  // check index range

  float result = 0.0f;
  for (int i = 0; i < 3; i++)
    result += input[index + i];

  output[index] = result;  // write back to global memory
}
```

each thread computes result for one element

Recap: Memory Load Reuse

上海创智学院
Shanghai Innovation Institute

```
float A[n][n];
float B[n][n];
float C[n][n];
```

```
C[i][j] = sum(A[i][k] * B[j][k], axis=k)
```

Access of A is independent of j,
tile the j dimension by t enables reuse of A for t times.

Recap: Shared Memory

上海创智学院
Shanghai Innovation Institute

Shared Memory / L1 Data Cache

CUDA 1D Convolution - Reused Shared Memory

上海创智学院 Shanghai Innovation Institute

```
int N = 1024 * 1024, THREADS_PER_BLK = 128
cudaMalloc(&devInput, sizeof(float) * (N + 2));  // allocate
cudaMalloc(&devOutput, sizeof(float) * N);       // allocate

// property initialize contents of devInput here ...

conv1D<<<N/THREADS_PER_BLK, THREADS_PER_BLK>>>(devInput, devOutput, N);
```

```
__global__ void conv1D(float* input, float* output, int N) {
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  if (index >= N) { return; }  // check index range

  __shared__ float smem[THREADS_PER_BLK+2]; // per-block allocation
  smem[threadIdx.x] = input[index];
  if (threadIdx.x < 2) {
    smem[THREADS_PER_BLK+threadIdx.x] = input[index+THREADS_PER_BLK];
  }

  __syncthreads();

  float result = 0.0f; // thread-local variable
  for (int i=0; i<3; i++)
    result += smem[threadIdx.x + i];
  output[index] = result / 3.f;
}
```

All threads cooperatively load block's support region from global into shared memory (total of 130 loads instead of 3 * 128 loads)

barrier (all threads in block)

each thread computes result for one element

CUDA 1D Convolution - Reused Shared Memory

上海创智学院 Shanghai Innovation Institute

```
int N = 1024 * 1024, THREADS_PER_BLK = 128
cudaMalloc(&devInput, sizeof(float) * (N + 2));  // allocate
cudaMalloc(&devOutput, sizeof(float) * N);       // allocate

// property initialize contents of devInput here ...

conv1D<<<N/THREADS_PER_BLK, THREADS_PER_BLK>>>(devInput, devOutput, N);
```

```
__global__ void conv1D(float* input, float* output, int N) {
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  if (index >= N) { return; }  // check index range

  __shared__ float smem[THREADS_PER_BLK+2]; // per-block allocation
  smem[threadIdx.x] = input[index];
  if (threadIdx.x < 2) {
    smem[THREADS_PER_BLK+threadIdx.x] = input[index+THREADS_PER_BLK];
  }

  __syncthreads();

  float result = 0.0f; // thread-local variable
  for (int i=0; i<3; i++)
    result += smem[threadIdx.x + i];
  output[index] = result / 3.f;
}
```

Discussion: why need sync all threads?

Answer: Thread 0 depends on the shared memory, which is load by thread 1, so need to ensure all threads finish reading

02

Matrix Multiplication

Naïve of Implementation Matrix Multiplication

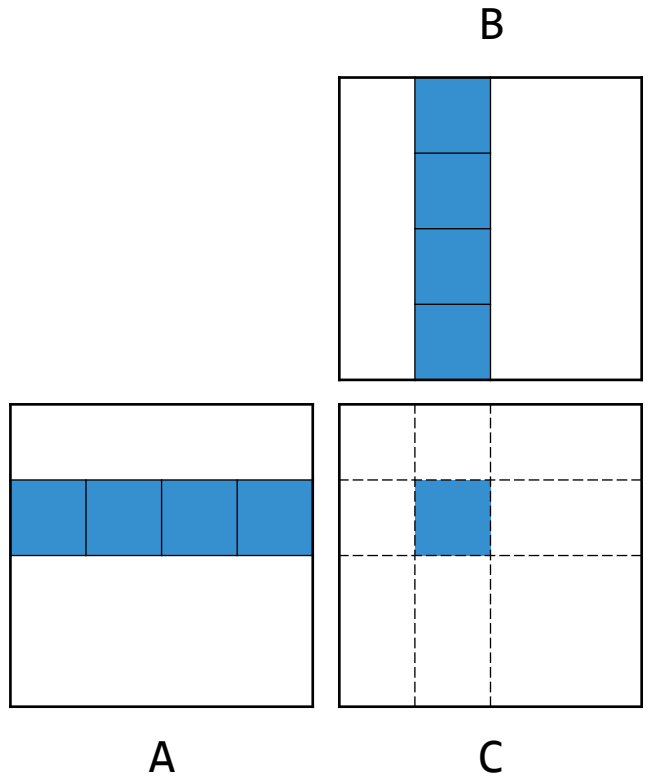
Compute $C = A \times B$

Each thread compute **one element**

```
__global__ void matmul(
  float A[N][N], float B[N][N], float C[N][N]
) {

  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;

  float result = 0;
  for (int k = 0; k < N; ++k) {
    result += A[x][k] * B[k][y];
  }
  C[x][y] = result;
}
```



Global memory access per thread: $2N$
Number of threads: N^2
Total global memory access: $2N^3$

Optimization 1: Thread-Level Register Tiling

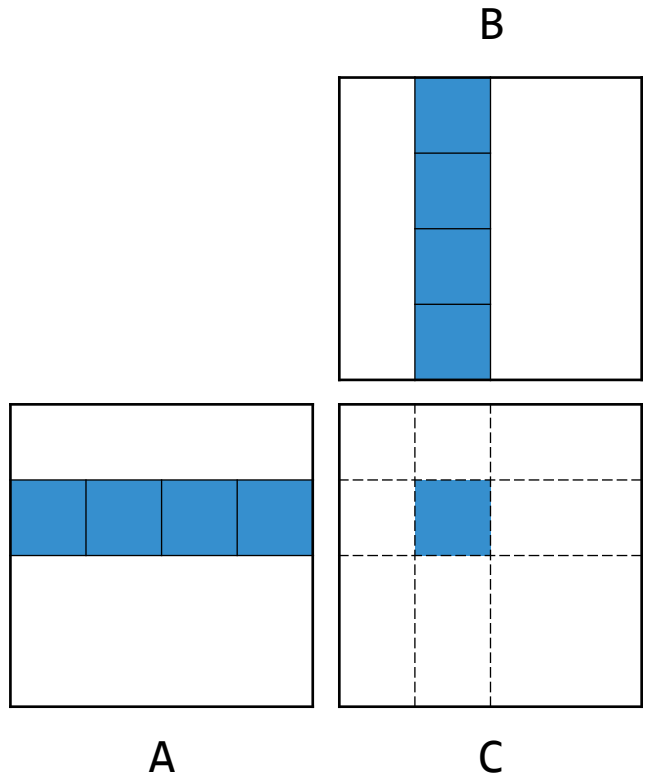
上海创智学院 Shanghai Innovation Institute

Compute $C = A \times B$

Each thread compute a $V \times V$ submatrix

```
__global__ void mm(float A[N][N], float B[N][N], float C[N][N]) {
  int ybase = blockIdx.y * blockDim.y + threadIdx.y;
  int xbase = blockIdx.x * blockDim.x + threadIdx.x;

  float c[V][V] = {0};
  float a[V], b[V];
  for (int k = 0; k < N; ++k) {
    a[:] = A[xbase*V : xbase*V + V, k];
    b[:] = B[k, ybase*V : ybase*V + V];
    for (int y = 0; y < V; ++y) {
      for (int x = 0; x < V; ++x) {
        c[x][y] += a[x] * b[y];
      }
    }
  }
  C[xbase*V : xbase*V + V, ybase*V : ybase*V + V] = c[:];
}
```



Global memory access per thread: $2NV$

Number of threads: $(N/V)^2$

Total global memory access: $2N^3/V$

Optimization 2: Block-Level Shared Memory Tiling

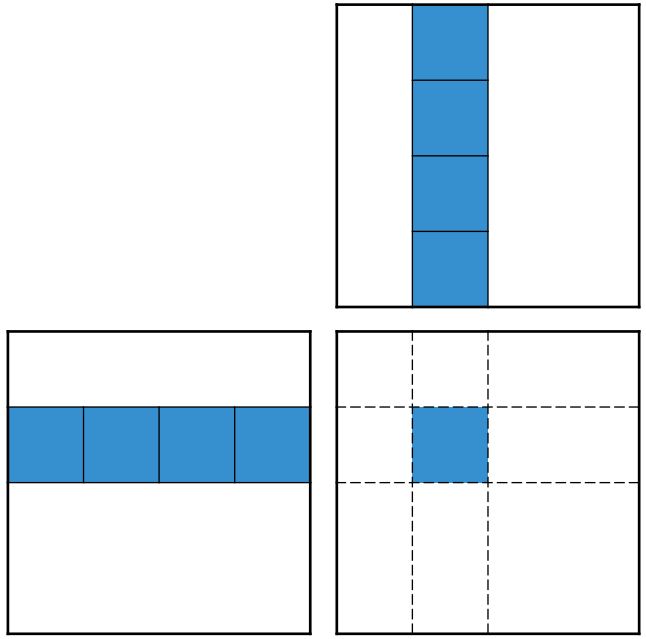
上海创智学院 Shanghai Innovation Institute

```
__global__ void mm(float A[N][N], float B[N][N], float C[N][N]) {
  __shared__ float sA[S][L], sB[S][L];
  float c[V][V] = {0};
  float a[V], b[V];
  int yblock = blockIdx.y;
  int xblock = blockIdx.x;

  for (int ko = 0; ko < N; ko += S) {
    __syncthreads();
    sA[:, :] = A[k : k + S, yblock * L : yblock * L + L];
    sB[:, :] = B[k : k + S, xblock * L : xblock * L + L];
    __syncthreads();
    for (int ki = 0; ki < S; ++ ki) {
      a[:] = sA[ki, threadIdx.y * V : threadIdx.y * V + V];
      b[:] = sB[ki, threadIdx.x * V : threadIdx.x * V + V];
      for (int y = 0; y < V; ++y)
        for (int x = 0; x < V; ++x)
          c[y][x] += a[y] * b[x];
    }
  }
  int ybase = blockIdx.y * blockDim.y + threadIdx.y;
  int xbase = blockIdx.x * blockDim.x + threadIdx.x;
  C[ybase * V : ybase*V + V, xbase*V : xbase*V + V] = c[:];
}
```

Global memory access per thread block: $2LN$
Number of thread blocks:
Total global memory access: $2N^3/L$

Shared memory access per thread: $2VN$
Number of threads:
Total shared memory access: $2N^3/V$



03

Parallel Reduction

Parallel Reduction

上海创智学院
Shanghai Innovation Institute

- Common and important primitive used by many MLSys operators: normalization, softmax, etc.

Sum = 25

Challenges of Parallel Reduction on GPU

上海创智学院
Shanghai Innovation Institute

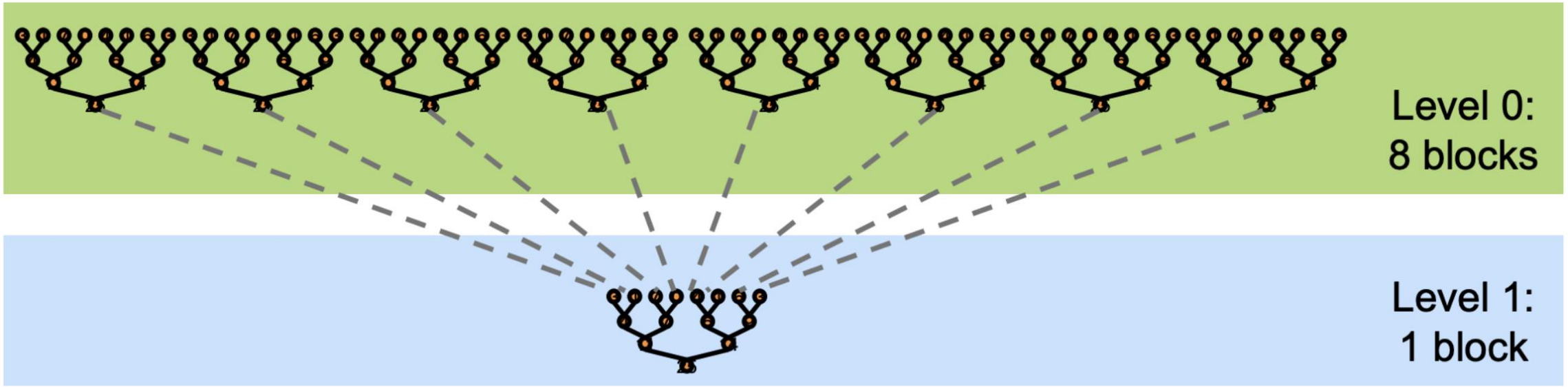
- Task: for a large array of n elements, compute $\sum_{i=1}^{n} A[i]$
- To achieve high GPU utilization
  - Need to use multiple thread blocks (since a block is assigned to one SM)
  - Each thread block reduces a portion of the array
- How to communicate partial results between thread blocks?

Sum = 25

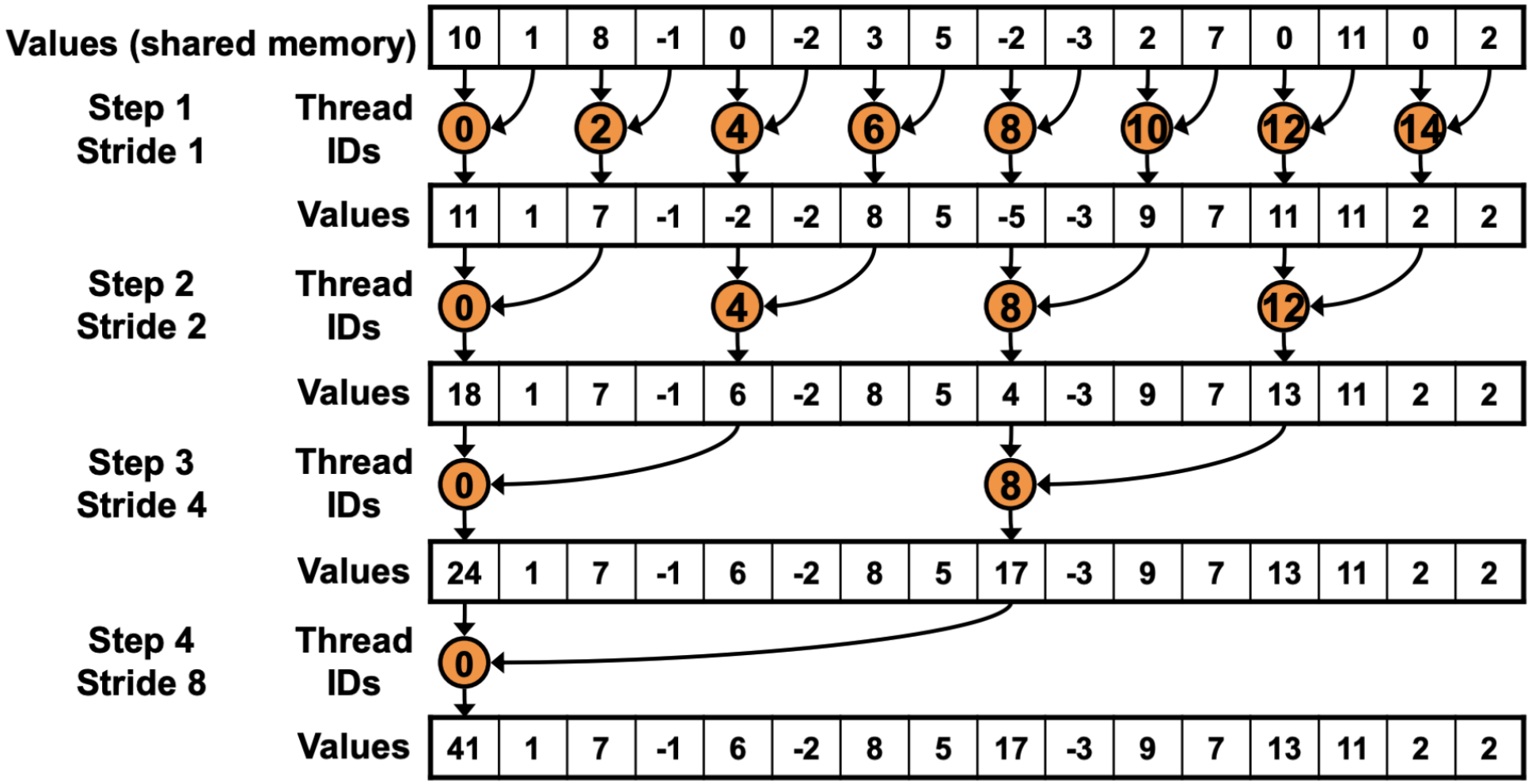
Multiple Kernels

上海创智学院
Shanghai Innovation Institute

- Avoid global synchronization by decompose computation into multiple kernel invocations
- Code for all levels is the same



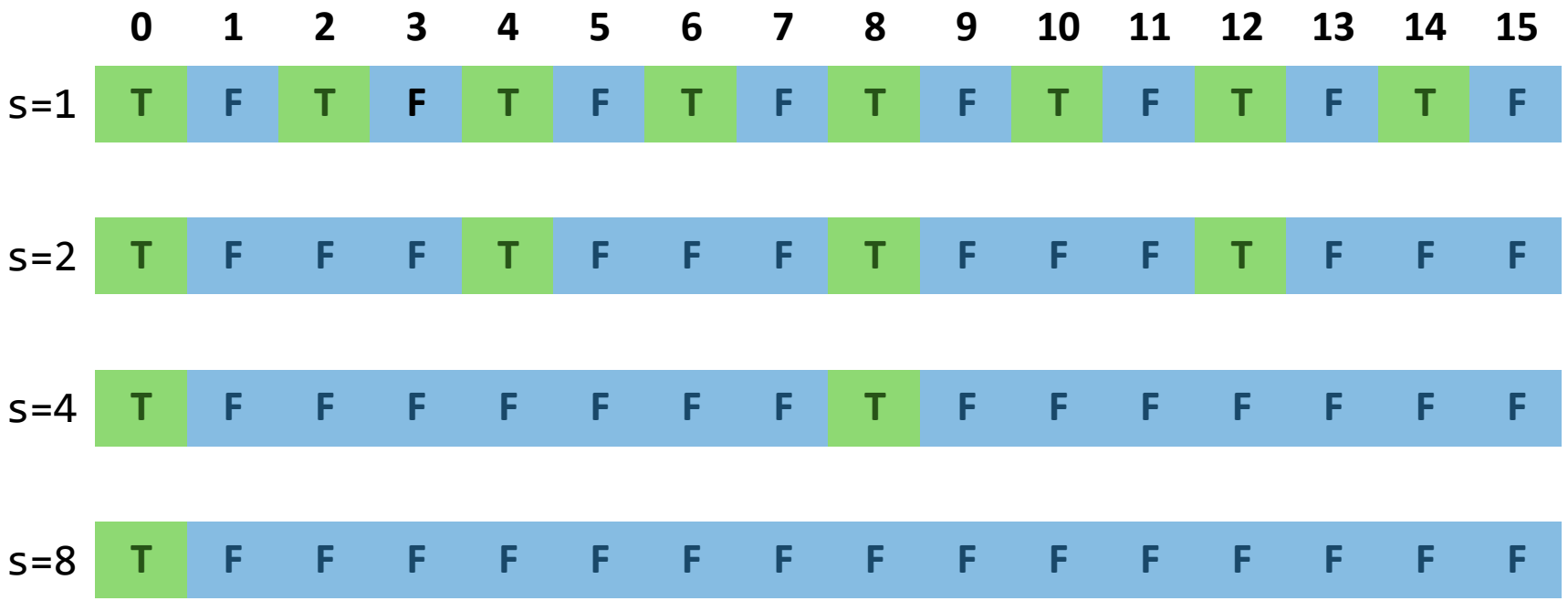
Version 1: Interleaved Addressing



Version 1: Divergent Warps

上海创智学院
Shanghai Innovation Institute

```
for(unsigned int s=1; s < blockDim.x; s *= 2) {
  if (tid % (2*s) == 0)
    sdata[tid] += sdata[tid + s];
  __syncthreads();
}
```



Version 2: Strided Index and Non-divergent Warp

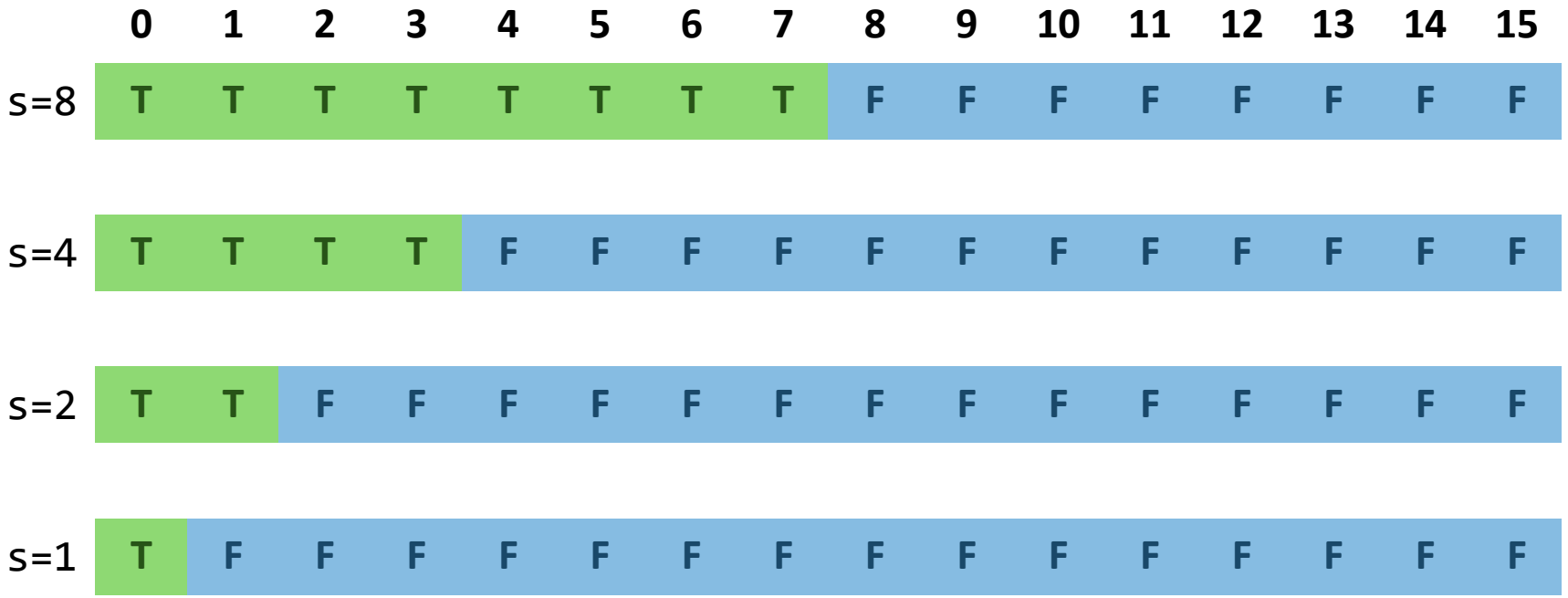
上海创智学院
Shanghai Innovation Institute

```
for(unsigned int s=1; s < blockDim.x; s *= 2) {
  if (tid % (2*s) == 0)
    sdata[tid] += sdata[tid + s];
  __syncthreads();
}
```

```
for(unsigned int s=blockDim.x / 2; s > 0; s /= 2) {
  if (threadIdx.x < s) {
    sdata[threadIdx.x] += sdata[threadIdx.x + s];
  }
  __syncthreads();
}
```

Version 2: Non-Divergent Warps

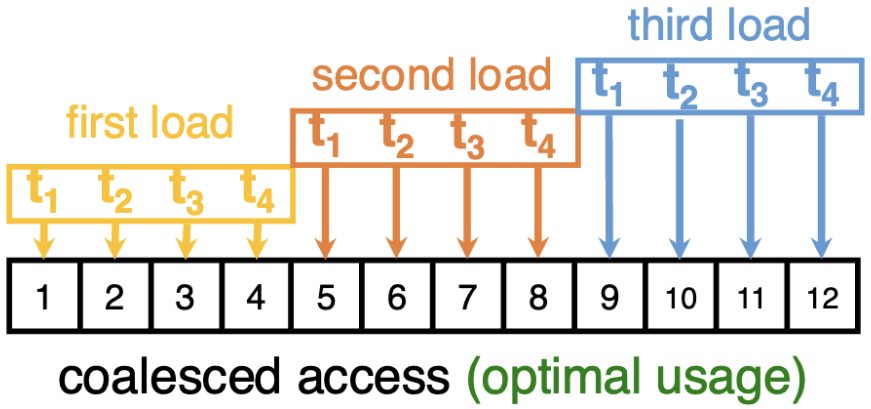
```
for(unsigned int s=blockDim.x / 2; s > 0; s /= 2) {
  if (threadIdx.x < s) {
    sdata[threadIdx.x] += sdata[threadIdx.x + s];
  }
  __syncthreads();
}
```

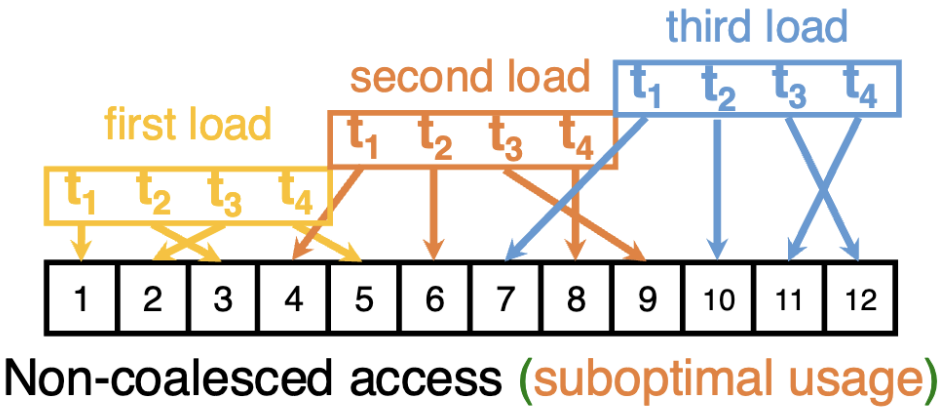


Coalesced Memory Access

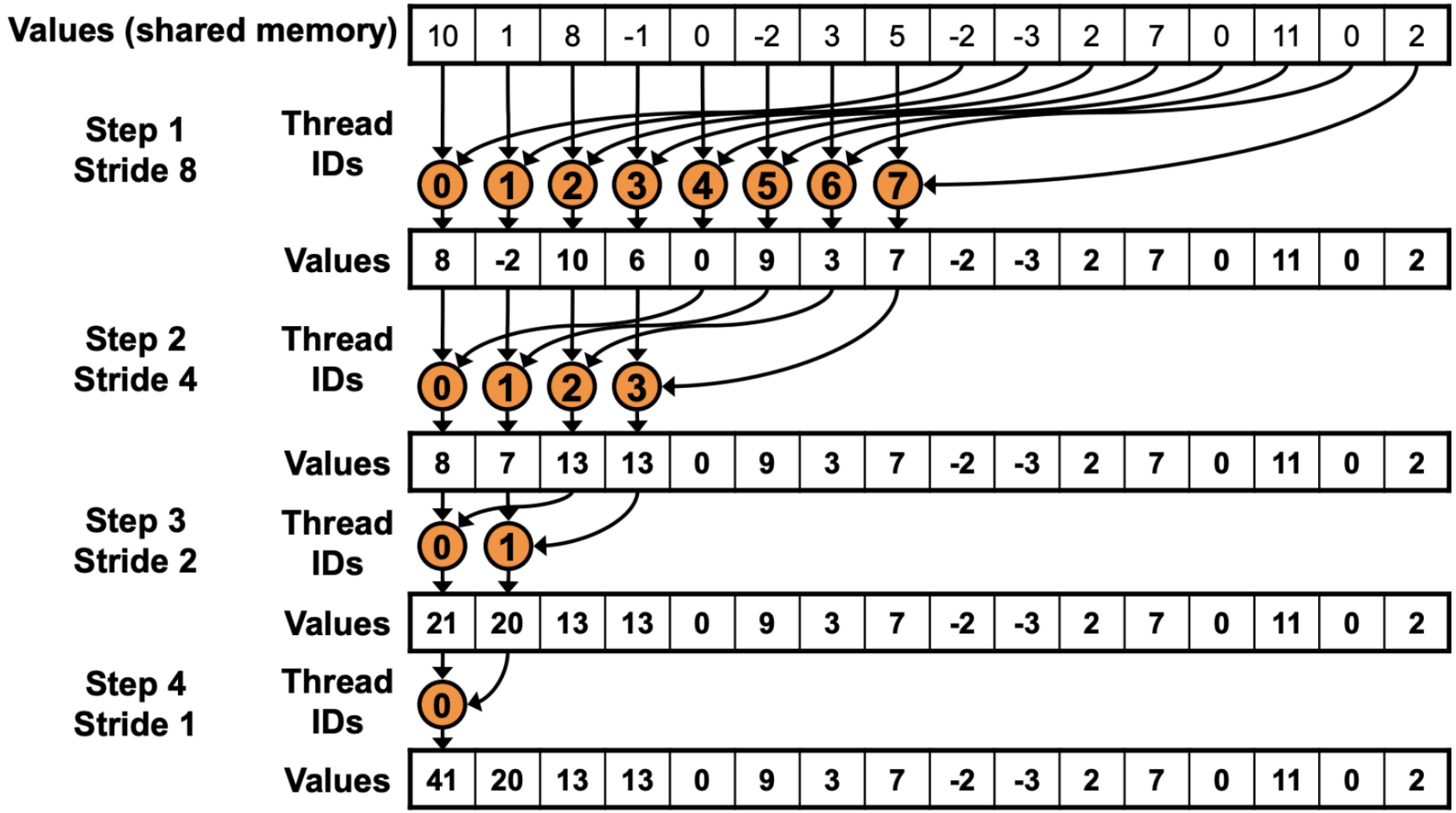
上海创智学院
Shanghai Innovation Institute

- Multiple GPU threads access consecutive memory addresses
- Maximize GPU memory bandwidth





Version 2: Sequential Addressing



Further Optimization

上海创智学院
Shanghai Innovation Institute

- Leverage **Tensor Cores** to speed up register level operation
- Leverage **TMA** to boost memory copy from global to shared memory
- Use **swizzle** to reduce bank conflict in shared memory
- Enable **multi-stage software pipeline** to hide data movement latency
- …

Acknowledgement

The development of this course, including its structure, content, and accompanying presentation slides, has been significantly influenced and inspired by the excellent work of instructors and institutions who have shared their materials openly. We wish to extend our sincere acknowledgement and gratitude to the following courses, which served as invaluable references and a source of pedagogical inspiration:

- Machine Learning Systems[15-442/15-642], by **Tianqi Chen** and **Zhihao Jia** at **CMU**.
- Advanced Topics in Machine Learning (Systems)[CS6216], by **Yao Lu** at **NUS**

While these materials provided a foundational blueprint and a wealth of insightful examples, all content herein has been adapted, modified, and curated to meet the specific learning objectives of our curriculum. Any errors, omissions, or shortcomings found in these course materials are entirely our own responsibility. We are profoundly grateful for the contributions of the educators listed above, whose dedication to teaching and knowledge-sharing has made the creation of this course possible.

System for Artificial Intelligence

Thanks

Siyuan Feng
Shanghai Innovation Institute