

```
//-----
// coalesce vc non coalesce
//-----
```

在 CUDA 中，实现 **memory coalescing**（内存合并访问）是性能优化的关键之一。下面是一个简单的例子，展示了：

1. 一个 **非 coalesced**（低效）访问的 kernel；
2. 一个 **coalesced**（高效）访问的 kernel；
3. 两者性能对比（通过 `cudaEvent` 计时）；

## 场景说明

我们将从全局内存中读取一个二维数组，并进行简单加法。关键是访问模式的对齐方式。

### CUDA 示例代码：Coalesced vs Non-Coalesced

```
#include <cuda_runtime.h>
#include <iostream>

#define N 1024    // rows
#define M 1024    // cols

// 非 coalesced 访问：列优先（慢）
__global__ void non_coalesced_kernel(float* input, float* output) {
    int row = threadIdx.x + blockIdx.x * blockDim.x;
    if (row >= N) return;

    for (int col = 0; col < M; col++) {
        output[row * M + col] = input[col * N + row]; // 不连续访问
    }
}

// coalesced 访问：行优先（快）
__global__ void coalesced_kernel(float* input, float* output) {
```

```

int row = threadIdx.x + blockIdx.x * blockDim.x;
if (row >= N) return;

for (int col = 0; col < M; col++) {
    output[row * M + col] = input[row * M + col]; // 连续访问
}
}

void benchmark(const char* name, void (*kernel)(float*, float*), float* d_in, float* d_out) {
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaMemset(d_out, 0, sizeof(float) * N * M);

    cudaEventRecord(start);
    kernel<<<(N + 255) / 256, 256>>>(d_in, d_out);
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);

    float ms = 0;
    cudaEventElapsedTime(&ms, start, stop);
    std::cout << name << " time: " << ms << " ms\n";
}

int main() {
    float* h_input = new float[N * M];
    float* h_output = new float[N * M];

    // Initialize input
    for (int i = 0; i < N * M; ++i) h_input[i] = i;

    float *d_input, *d_output;
    cudaMalloc(&d_input, sizeof(float) * N * M);
    cudaMalloc(&d_output, sizeof(float) * N * M);
    cudaMemcpy(d_input, h_input, sizeof(float) * N * M, cudaMemcpyHostToDevice);

    // Benchmark
    std::cout << "Benchmarking...\n";

    benchmark("Coalesced", coalesced_kernel, d_input, d_output);
    benchmark("Non-Coalesced", non_coalesced_kernel, d_input, d_output);

    cudaMemcpy(h_output, d_output, sizeof(float) * N * M, cudaMemcpyDeviceToHost);

    cudaFree(d_input);
}

```

```
    cudaFree(d_output);
    delete[] h_input;
    delete[] h_output;
    return 0;
}
```

## 解释

- **non-coalesced kernel** 访问的是列向量 → 不同线程访问相距 N 的内存 → 不连续访问 → 低效；
- **coalesced kernel** 访问的是行向量 → 连续内存访问 → 被合并到一次 memory transaction 中 → 高效。



## 运行效果 (依赖 GPU)

你会观察到：

- coalesced kernel 的运行时间远小于 non-coalesced kernel；
- 在数据规模较大、warp 尺寸合理的条件下，差距明显。

```
//-----
```

请结合以上code, 说明 coalescing的原理, 请结合具体数字, 例子, 流程图

非常好, 我们现在结合上面 CUDA 示例, 用具体数字 + 内存地址 + 流程图 来解释 **memory coalescing** (内存合并访问) 的原理。

## ✓ 什么是 Memory Coalescing?

在 CUDA 中，一个 **warp** (32个线程) 访问 global memory 时，如果 所有线程访问的是连续内存地址，这32次访问可以被合并成 1次 **memory transaction**，极大提高带宽利用率。这种访问叫 **coalesced access** (合并访问)。



## 举个具体的例子

我们设：

- $N = 4, M = 8$ ，即矩阵是 4 行 8 列，线性数组 `input[i * M + j]`
- 数组排布如下 (row-major) :

`less`

内存 layout (地址连续) :

```
Row 0: input[0], input[1], input[2], ..., input[7]
Row 1: input[8], input[9], ..., input[15]
Row 2: input[16], ..., input[23]
Row 3: input[24], ..., input[31]
```

### 1. Coalesced 访问 (行优先)

`cpp`

```
output[row * M + col] = input[row * M + col];
```

- 如果启动 `blockDim.x = 32`，每个线程 `threadIdx.x = 0...31` 对应 `row = threadIdx.x`。
- 第一个 `col = 0` 时：

`cpp`

```
input[0], input[8], input[16], input[24] ← 线程0、1、2、3分别访问
```

看似不连续对吧？但我们假设 每个线程处理整行：**col 0~7**：

thread 0: input[0~7]

thread 1: input[8~15]

thread 2: input[16~23]

...

这时候，一个 warp 的线程访问的是连续地址段：input[0~255]，这就可以 coalesce。

 多个线程同时访问一个连续的内存段 → coalesced

## 2. Non-Coalesced 访问 (列优先)

cpp

CopyEdit

```
output[row * M + col] = input[col * N + row];
```

以 col = 0 为例：

cpp

CopyEdit

```
thread 0 (row=0): input[0*4 + 0] = input[0]
```

```
thread 1 (row=1): input[0*4 + 1] = input[1]
```

```
thread 2 (row=2): input[0*4 + 2] = input[2]
```

```
thread 3 (row=3): input[0*4 + 3] = input[3]
```

...

- 对于一个 warp 来说，它访问的是地址：input[0], input[1],  
input[2], input[3]...

- 在逻辑上是连续的，但物理上在 transpose 的二维矩阵中是不连续的。

- 每个线程从不同的 row、相同的 col 中读取 → 实际在 memory 中是跳跃的

✗ 非连续内存访问 → non-coalesced → 多次 memory transaction

## 内存访问对比流程图

✓ Coalesced (行优先)

线程访问模式:

```
less  
CopyEdit  
Thread 0: input[0, 1, 2, 3, 4, 5, 6, 7]  
Thread 1: input[8, 9, 10, 11, 12, 13, 14, 15]  
Thread 2: input[16, 17, ...]
```

图示:

```
less  
CopyEdit  
GPU Threads → |||||||  
Memory Addr → [0, 1, 2, 3, ..., 31, ...] (连续块)  
→ 一次 burst access 即可读入多个值
```

✗ Non-Coalesced (列优先)

线程访问模式:

```
less  
CopyEdit  
Thread 0: input[0, 4, 8, 12, 16, 20, ...]  
Thread 1: input[1, 5, 9, 13, 17, 21, ...]  
...  
图示:
```

```
less  
CopyEdit  
GPU Threads → |||||||  
Memory Addr → [0, 4, 8, 12, ...] (跳跃块)
```

→ 每个地址独立，需要多个 **memory transaction**，访问慢

## 🎯 总结

方式	内存访问模式	Bandwidth 利用率	是否合并访问	说明
Row-major (行优先)	连续 (coalesced)	✓ 高	✓ Yes	建议使用
Column-major (列优先)	不连续 (non-coalesced)	✗ 低	✗ No	通常要改为 tile / transpose 优化

## 📌 优化建议

- 始终让 **thread id** 访问连续地址；
- 可结合 **shared memory** 做 tile；
- 如果数据天然不连续（如转置），可先加载到 shared memory、再重排访问。

//-----

继续加上 shared memory 优化版本

下面是基于 **CUDA shared memory** 优化 **memory coalescing** 的完整示例，模拟 **transpose** 操作，并解决由于列优先带来的 **non-coalesced global memory** 访问问题。

## 🧠 背景场景

- 目标：将一个  $N \times M$  矩阵转置

- baseline: global memory 直接读写 → 非 coalesced
- 优化: 通过 **shared memory tile + transpose**, 让读/写都尽可能 coalesced

## ✓ Shared Memory 优化版本代码 (含 Coalesced 读写)

```

cpp
CopyEdit
// block size
#define TILE_DIM 32
#define BLOCK_ROWS 8

__global__ void transpose_shared_memory(float* odata, const
float* idata, int width, int height) {
    __shared__ float tile[TILE_DIM][TILE_DIM + 1]; // +1 to
avoid bank conflicts

    int x = blockIdx.x * TILE_DIM + threadIdx.x; // col
    int y = blockIdx.y * TILE_DIM + threadIdx.y; // row

    // transpose block index
    int width_in = width;
    int height_in = height;

    // ---- 1. Coalesced read from idata to shared memory
-----
    if (x < width && y < height)
        tile[threadIdx.y][threadIdx.x] = idata[y * width +
x];

    __syncthreads();

    // transpose thread indices
    x = blockIdx.y * TILE_DIM + threadIdx.x; // 注意交换 x/y
以完成转置
    y = blockIdx.x * TILE_DIM + threadIdx.y;

    // ---- 2. Coalesced write from shared memory to odata
-----

```

```

    if (x < height && y < width)
        odata[y * height + x] = tile[threadIdx.x]
[threadIdx.y];
}

```

## 🎯 如何优化了 memory access?

操作阶段	内存类型	Access Pattern	是否 Coalesced
global → shared	global memory	thread (0..31) 连续访问行元素	✓ 是
shared → global	global memory	通过 shared memory 转置后再写	✓ 是

## ✓ 为什么需要 `tile[TILE_DIM][TILE_DIM+1]`?

- 加 +1 避免 shared memory bank conflict
- CUDA 的 shared memory 是按 32 bank 分布，访问 `tile[i][i]` (对角线) 时加1 避免多个线程访问同一个 bank

## 📌 Kernel 启动方式示例

```

cpp
CopyEdit
int width = 1024;
int height = 1024;
dim3 gridDim((width + TILE_DIM - 1) / TILE_DIM,
              (height + TILE_DIM - 1) / TILE_DIM);
dim3 blockDim(TILE_DIM, BLOCK_ROWS);

transpose_shared_memory<<<gridDim, blockDim>>>(odata,
idata, width, height);

```

## 🔍 可视化流程图

wasm

[CopyEdit](#)

global input:

A00	A01	A02	A03
A10	A11	A12	A13
A20	A21	A22	A23

-> 使用 shared memory tile[32][32+1] 加载一个 block

-> 线程对 tile[][] 做转置 (swap threadIdx)

-> 写回 global memory:

global output:

A00	A10	A20	A30
A01	A11	A21	A31
A02	A12	A22	A32

## ✓ 性能对比 (理论)

方法	Rea d	Writ e	Shared Memory	Coalesce d	性能预期
直接 global transpose	✗	✗	无	否	最慢
shared memory + tile transpose	✓	✓	有	是	快 ~5x
使用 CUDA cuBLAS 优化接口	✓	✓	内部优化	是	最快 (推荐)

```
//-----  
//what is coalescence in cuda kernel function development? please use detailed code examples  
//-----
```

In CUDA kernel development, coalesced memory access refers to a pattern where consecutive threads in a warp access consecutive memory locations, allowing the GPU to combine these memory requests into a single transaction. This is crucial for achieving high memory bandwidth.

What is Coalescing?

When threads in a warp (32 threads) access global memory:

Coalesced: Threads access consecutive 32-bit words → 1 memory transaction

Non-coalesced: Threads access scattered locations → up to 32 separate transactions

Code Examples

### 1. Coalesced Memory Access (Good)

```
__global__ void coalescedKernel(float* output, const float* input, int size) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (tid < size) {
        // Coalesced: consecutive threads access consecutive memory locations
        output[tid] = input[tid] * 2.0f;
    }
}

// Launch configuration
int size = 1024;
int threadsPerBlock = 256;
int blocksPerGrid = (size + threadsPerBlock - 1) / threadsPerBlock;

float *d_input, *d_output;
cudaMalloc(&d_input, size * sizeof(float));
cudaMalloc(&d_output, size * sizeof(float));

// Initialize input data
// ...

coalescedKernel<<<blocksPerGrid, threadsPerBlock>>>(d_output, d_input, size);
```

Memory access pattern:

Thread 0: input[0], output[0]

Thread 1: input[1], output[1]

Thread 2: input[2], output[2]

... → Perfectly coalesced!

## 2. Non-Coalesced Memory Access (Bad)

```
__global__ void nonCoalescedKernel(float* output, const float* input, int size) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (tid < size) {
        // Non-coalesced: consecutive threads access scattered locations
        output[tid * 2] = input[tid * 2] * 2.0f;
    }
}
```

Memory access pattern:

Thread 0: input[0], output[0]

Thread 1: input[2], output[2]

Thread 2: input[4], output[4]

... → Strided access, poor coalescing

## 3. 2D Coalesced Access Example

```
// Coalesced 2D access
__global__ void coalesced2DKernel(float* output, const float* input,
                                    int width, int height) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x < width && y < height) {
        // Coalesced: x dimension is contiguous in memory
        int index = y * width + x;
        output[index] = input[index] * 2.0f;
    }
}

// Non-coalesced 2D access (bad)
__global__ void nonCoalesced2DKernel(float* output, const float* input,
                                      int width, int height) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x < width && y < height) {
        // Non-coalesced: y dimension causes strided access
        int index = x * height + y; // Wrong order for coalescing
        output[index] = input[index] * 2.0f;
    }
}

// Launch 2D kernel
dim3 threadsPerBlock(16, 16);
```

```

dim3 blocksPerGrid((width + 15) / 16, (height + 15) / 16);
coalesced2DKernel<<<blocksPerGrid, threadsPerBlock>>>(d_output, d_input, width, height);

```

#### 4. Shared Memory for Coalescing

```

__global__ void sharedMemoryKernel(float* output, const float* input, int size) {
    __shared__ float tile[256];

    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int localIdx = threadIdx.x;

    // Coalesced read from global memory to shared memory
    if (tid < size) {
        tile[localIdx] = input[tid];
    }
    __syncthreads();

    // Process data in shared memory (fast)
    if (tid < size) {
        float result = tile[localIdx] * 2.0f;

        // Coalesced write back to global memory
        output[tid] = result;
    }
}

```

#### 5. Transpose with Coalescing

```

// Coalesced transpose
__global__ void transposeCoalesced(float* output, const float* input,
                                    int width, int height) {
    __shared__ float tile[32][33]; // Padding to avoid bank conflicts

    int x = blockIdx.x * 32 + threadIdx.x;
    int y = blockIdx.y * 32 + threadIdx.y;

    // Coalesced read
    if (x < width && y < height) {
        tile[threadIdx.y][threadIdx.x] = input[y * width + x];
    }
    __syncthreads();

    // Transposed write with coalescing
    x = blockIdx.y * 32 + threadIdx.x;
    y = blockIdx.x * 32 + threadIdx.y;

    if (x < height && y < width) {
        output[y * height + x] = tile[threadIdx.x][threadIdx.y];
    }
}

```

## 6. Structure of Arrays (SoA) for Coalescing

```
// Good: Structure of Arrays (SoA)
struct SoA {
    float* x;
    float* y;
    float* z;
};

__global__ void processSoA(SoA data, int size) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (tid < size) {
        // Coalesced: all threads access consecutive x, then y, then z
        data.x[tid] *= 2.0f;
        data.y[tid] *= 2.0f;
        data.z[tid] *= 2.0f;
    }
}

// Bad: Array of Structures (AoS)
struct AoS {
    float x, y, z;
};

__global__ void processAoS(AoS* data, int size) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (tid < size) {
        // Non-coalesced: threads access scattered structure members
        data[tid].x *= 2.0f;
        data[tid].y *= 2.0f;
        data[tid].z *= 2.0f;
    }
}
```

### Best Practices for Coalescing

Thread Access Pattern: Ensure consecutive threads access consecutive memory locations

Data Alignment: Start addresses should be aligned to 128-byte boundaries

Access Size: Use naturally aligned data types (4, 8, 16 bytes)

Shared Memory: Use for data reuse and to enable coalescing

SoA vs AoS: Prefer Structure of Arrays for coalesced access

### Performance Impact

Coalesced memory access can provide 10-20x performance improvement compared to non-coalesced access due to reduced memory transactions and better bandwidth utilization.

Always use CUDA profilers (nvprof, Nsight Compute) to analyze memory access patterns and identify coalescing issues in your kernels.

//-----  
//  
//-----

//-----  
//  
//-----

//-----  
//  
//-----

//-----  
//  
//-----

//-----  
//  
//-----

//-----  
//  
//-----

//-----  
//  
//-----