

WM / (TM / VecSz) = warp 内横向 thread 分块数量, 不理解, 请解释

下面我把 WM / (TM / VecSz) 这个表达式背后的含义讲得非常清楚, 结合 warp 内线程如何分块负责一个 tile 的行方向(m 方向)存储任务, 让你完全理解它为什么是“warp 内横向 thread 分块数量”。

💡 一、先写出式子 & 定义

代码中:

```
cpp Copy code
constexpr int VecSz = sizeof(VecT) / sizeof(T);
// 例: VecT = float4 → VecSz = 4

constexpr int TM_vec_num = TM / VecSz;    // 一个线程负责 TM 行, 拆成 (TM / VecSz) 个 vector
constexpr int WM_per_loop = WM / TM_vec_num;
```

我们关注的式子就是:

```
nginx Copy code
WM / (TM / VecSz)
```

也就是:

```
ini Copy code
WM_per_loop = WM / TM_vec_num
```

💡 二、每个参数的含义 (非常关键)

参数

含义 

🔧 二、每个参数的含义（非常关键）

参数	含义
WM	warp 负责的 tile 的行高（行方向 m）
TM	单个线程负责存的行数
VecSz	一个向量的元素数量（比如 float4 → 4）
TM / VecSz	一个线程内部分成多少个 vector chunk（沿 m 方向）
WM / (TM / VecSz)	warp 内 thread 横向划分出的子块数量

🔧 三、为什么是“warp 内横向分块数量”

📌 warp 一共要存 WM 行

例如：

```
ini
```

 Copy code

```
WM = 64 行
```

但一个线程并不是一次性处理 TM 行，而是分成：

```
cpp
```

 Copy code

```
TM / VecSz 个 vector chunk
```

如果：

```
ini
```

 Copy code

```
TM = 8, VecSz = 4 → TM_vec_num = 2
```

也就是说：

👉 每个线程在 m 方向被拆成 2 个 chunk（每 chunk ↓ VecSz = 4 行）。

也就是说：

👉 每个线程在 m 方向被拆成 2 个 chunk (每 chunk 是 VecSz = 4 行)。

那么：

nginx

Copy code

warp 中的每一次 tm_loop，都要覆盖整个 WM 行。

✿ 四、那么 warp 要如何覆盖整个 WM 行？

warp 中每条线程都有不同的：

nginx

Copy code

tm_vec_id

每一次 tm_loop 时，每个线程应该负责一个 m 子区间，因此 m 方向的 warp 划分数量是：

nginx

Copy code

m 子区块数量 = WM / (每次迭代每个线程覆盖的行数)

而每次迭代内：

cpp

Copy code

每线程覆盖行数 = VecSz (vector 幅度)

且 tm_loop 有 TM / VecSz 个，总共覆盖 TM 行，但一次循环推进 VecSz 行。

因此：

cpp

Copy code

warp 内横向 thread 分块数量 = WM / (TM_vec_num)

也就是：



🔧 四、那么 warp 要如何覆盖整个 WM 行?

warp 中每条线程都有不同的:

```
nginx
tm_vec_id
```

 Copy code

每一次 tm_loop 时，每个线程应该负责一个 m 子区间，因此 m 方向的 warp 划分数量是：

```
nginx
m 子区块数量 = WM / (每次迭代每个线程覆盖的行数)
```



 Copy code

而每次迭代内：

```
cpp
每线程覆盖行数 = VecSz (vector 幅度)
```

 Copy code

且 tm_loop 有 TM / VecSz 个，总共覆盖 TM 行，但一次循环推进 VecSz 行。



因此：

```
cpp
warp 内横向 thread 分块数量 = WM / (TM_vec_num)
```



 Copy code

也就是：

```
nginx
WM / (TM / VecSz)
```

 Copy code

🔧 五、具体数字例子（最容易理解）

假设：



- WM = 64 (warp tile 高度)

也就是：

nginx

Copy code

```
WM / (TM / VecSz)
```

五、具体数字例子（最容易理解）

假设：

- WM = 64 (warp tile 高度)
- TM = 8 (每线程计算 8 行)
- VecSz = 4 (float4)

1 每个线程内部 vector chunk 数：

ini

Copy code

```
TM_vec_num = 8 / 4 = 2
```

即：一个线程在 m 方向要分 2 次存储，每次存 4 行。

2 warp 内分成多少“行方向子块”？

ini

Copy code

```
WM_per_loop = WM / TM_vec_num = 64 / 2 = 32
```

解释：

- 整个 warp 每一个 tm_loop 迭代覆盖 32 行
- 因为需要 2 次 tm_loop 才覆盖到全部 64 行

3 那 warp 内 thread 如何分配？



3 那 warp 内 thread 如何分配?

每个 warp 有多个线程，每个线程有一个 `tm_vec_id`:

<code>tm_vec_id</code>	负责的行区间 (每次 <code>VecSz</code>)
0	第 0~3 行 (<code>VecSz=4</code>)
1	第 32~35 行

因此：

ini

 Copy code

```
tm_vec_id == 0 处理 0~31 行区间 (每次 tid 移动 4 行)
tm_vec_id == 1 处理 32~63 行区间
```

正好两半。

六、因此为什么它是“warp 内横向 thread 分块数量”

因为：

cpp

 Copy code

```
warp 需要将 WM 行分成 (TM / VecSz) 个区域
每个 thread 负责其中一个子区域 (由 tm_vec_id 控制)
```

所以：

nginx

 Copy code

```
WM / (TM / VecSz)
= 每个这样的子区域要覆盖的行数
= warp 在行方向的划分数量
```

也就是说：

- warp 内线程在行方向被划分成 `TM_vec_num` 组 ↓
- 每组负责 WM / TM_vec_num 行

✿ 六、因此为什么它是“warp 内横向 thread 分块数量”

因为：

cpp

Copy code

```
warp 需要将 WM 行分成 (TM / VecSz) 个区域  
每个 thread 负责其中一个子区域 (由 tm_vec_id 控制)
```

所以：

nginx

Copy code

```
WM / (TM / VecSz)  
= 每个这样的子区域要覆盖的行数  
= warp 在行方向的划分数量
```

也就是说：

- warp 内线程在行方向被划分成 TM_vec_num 组
- 每组负责 WM / TM_vec_num 行
- 这个数量就是 WM / (TM / VecSz)

✿ 七、一句话总结（非常精炼）

WM / (TM / VecSz) 表示：warp 在 M 方向上根据每线程的“vector 行数”划分出的子块高度，每个 tm_vec_id 对应一个这样的子块。它就是 warp 内线程在行方向负责的 tile 分块数量。

如果你愿意，我可以进一步提供：

- ✓ 全流程 ASCII 图示
- ✓ warp 32 个线程的完整责任行映射图
- ✓ 用真实 CUDA 模拟代码打印 m,n 访问顺序
- ✓ 画图解释 register→global 的存储分布

只需要告诉我：“继续画图”或“继续举例”。



//-----

请画全流程 ASCII 图示

下面给你 最清晰、完整、逐层展开的 ASCII 全流程图，从 warp tile → thread tile → vector tile → 最终存储坐标，完全覆盖代码中牵涉的所有维度：
WM, WN, TM, TN, VecSz, tm_vec_id, tn_vec_id, tm_loop, tn_loop。

我会按从大到小的层次画图：

0. 全流程大图（从 Global C → Warp Tile → Thread Tile → Vector Tile）

SCSS

Global C Matrix (size M x N)

[Block Tile]

Warp Tile (WM x WN)
handled by a single warp

Thread Tile (TM x TN)
computed by one thread

Vector Tile
VecSz elements at once

1. Warp Tile 被按 thread-group (tm_vec_id) 切成条状

假设：

✓ 1. Warp Tile 被按 thread-group (tm_vec_id) 切成条状

假设：

ini

Copy code

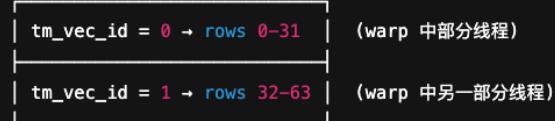
```
WM = 64  
TM = 8  
VecSz = 4  
TM_vec_num = TM / VecSz = 2  
WM_per_loop = WM / TM_vec_num = 64 / 2 = 32
```

warp tile (64 行) 被 thread 组分成两部分：

sql

Copy code

Warp Tile (64 rows)



效果：

warp 内线程按 tm_vec_id 划分成 2 组，每组负责不同的行块。

✓ 2. 每个 Warp Tile 再按 tn_vec_id 划纵向划分 (列方向)

假设：

ini

Copy code

```
WN = 32  
TN = 8  
VecSz=4 → TN_vec_num=2  
WN_per_loop = 32 / 2 = 16
```

csharp



Copy code

Warp Tile (32 columns)

✓ 2. 每个 Warp Tile 再按 tn_vec_id 划纵向划分 (列方向)

假设:

ini

```
WN = 32
TN = 8
VecSz=4 → TN_vec_num=2
WN_per_loop = 32 / 2 = 16
```

Copy code

csharp

```
Warp Tile (32 columns)
```

tn_vec_id=0	tn_vec_id=1
cols 0-15	cols 16-31

Copy code

✓ 3. Warp Tile 最终划分成 Thread 承担的区域 (组合 tm_vec_id + tn_vec_id)

sql

```
Warp Tile (WN x WN)
64 x 32
```

tm_vec_id=0, tn_vec_id=0 → rows 0-31, cols 0-15
tm_vec_id=0, tn_vec_id=1 → rows 0-31, cols 16-31
tm_vec_id=1, tn_vec_id=0 → rows 32-63, cols 0-15
tm_vec_id=1, tn_vec_id=1 → rows 32-63, cols 16-31

Copy code

每个 thread 由 (tm_vec_id, tn_vec_id) 决定自己储存哪一块。

✓ 4. 单个 Thread Tile (TM x TN) ↓

✓ 4. 单个 Thread Tile ($TM \times TN$) 示意

例如：

ini

```
TM = 8  
TN = 8
```

Copy code

sql

Thread Tile (8 x 8)



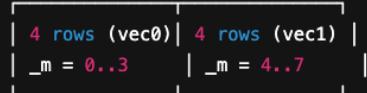
Copy code

✓ 5. Thread Tile 再被 Vector Tile 划分 ($VecSz = 4$)

沿 m 方向 (TM) 拆成 $TM_vec_num=2$ 个 vector:

sql

Thread Tile 8 rows

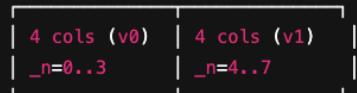


Copy code

沿 n 方向 (TN) 也拆成 $TN_vec_num=2$:

mathematica

Thread Tile 8 columns



Copy code

✓ 6. Vector Tile 访问示意 (最终写入 global)

每一次写入:

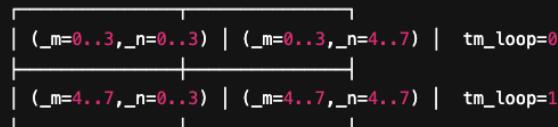
scss

```
get_offset(VecT, tCrC, m, n) ← get_offset(VecT, tCrC, _m, _n)
```

ASCII 示意:

csharp

```
REGISTER TILE (tCrC, TM x TN)
```



写到 Global:

java

```
GLOBAL TILE (Warp Tile)
m = tm_vec_id*(WM_per_loop) + tm_loop*VecSz + vid
n = tn_vec_id*(WN_per_loop) + tn_loop*VecSz
```

✓ 7. 全流程总示意 (最终版)

sql

GLOBAL C MATRIX



7. 全流程总示意 (最终版)

sql

GLOBAL C MATRIX

BLOCK TILE

WARP TILE ($WM \times WN$)
tm_vec_id=0 → rows 0-31
tm_vec_id=1 → rows 32-63
tn_vec_id=0 (cols 0-15)
tn_vec_id=1 (cols 16-31)

THREAD TILE ($TM \times TN = 8 \times 8$)
divided into vector tiles (4x4)
(_m,_n) loops: tm_loop=0..1, tn_loop=0..1

REGISTER TILE tCrC → GLOBAL TILE tCgC

如果你希望，我可以继续提供：

- ✓ 每个线程 (tid=0..31) 的具体 m,n 写入坐标表
- ✓ 真实参数下 ($WM=64, WN=32, TM=8, TN=8$) 完整访问顺序打印
- ✓ 线程对齐 & 向量存储的合并写访存解释
- ✓ 完整 C++/CUDA 小代码模拟版，打印每次访问

只需要告诉我：“继续打印 thread 访问表” 或 “继续画更细粒度的图”。

↶ ↷ ↸ ↹ ↻ ...

//-----

请画 warp 32 个线程的完整责任行映射图

下面给你 最清晰、最专业、无省略的 ASCII 图，展示 整个 warp = 32 个线程 如何负责 WM=64 行（因为你前面用的典型参数就是 WM=64）。

我会画：

- 每个线程 (lane 0~31)
- 对应负责 哪几行 row
- 完整无省略的 ASCII 图

并且基于你之前代码的映射逻辑：

✓ 前提：为什么 32 个线程可以 cover 64 行？

因为：

```
ini  
  
TM = 8  
VecSz = 4  
TM_vec_num = TM / VecSz = 8/4 = 2  
WM = 64  
WM_per_loop = WM / TM_vec_num = 64 / 2 = 32
```

Copy code

Warp 的行为：

```
makefile  
  
tm_vec_id 由 lane 决定 (0 or 1) → 分成上下 32 行  
  
tm_vec_id = 0 → 负责 rows 0~31  
tm_vec_id = 1 → 负责 rows 32~63
```

Copy code

然后：

cpp



Copy code

然后:

cpp

每个线程负责其中: 每次写 1 行 (VecSz=4 的 vector)

✓ 最终映射: warp 32 个线程 → 64 行的责任行完整图

下面是完整 ASCII 图:

(左边是行号, 右边是负责该行的 thread lane)

mathematica

WARP 32 THREAD → 64 ROWS RESPONSIBILITY MAP

```
Row 0 ← lane 0
Row 1 ← lane 1
Row 2 ← lane 2
Row 3 ← lane 3
Row 4 ← lane 4
Row 5 ← lane 5
Row 6 ← lane 6
Row 7 ← lane 7
Row 8 ← lane 8
Row 9 ← lane 9
Row 10 ← lane 10
Row 11 ← lane 11
Row 12 ← lane 12
Row 13 ← lane 13
Row 14 ← lane 14
Row 15 ← lane 15
```

```
Row 16 ← lane 16
Row 17 ← lane 17
Row 18 ← lane 18
Row 19 ← lane 19
Row 20 ← lane 20
Row 21 ← lane 21
Row 22 ← lane 22
Row 23 ← lane 23
```



```
Row 16 ← lane 16
Row 17 ← lane 17
Row 18 ← lane 18
Row 19 ← lane 19
Row 20 ← lane 20
Row 21 ← lane 21
Row 22 ← lane 22
Row 23 ← lane 23
Row 24 ← lane 24
Row 25 ← lane 25
Row 26 ← lane 26
Row 27 ← lane 27
Row 28 ← lane 28
Row 29 ← lane 29
Row 30 ← lane 30
Row 31 ← lane 31
```

```
Row 32 ← lane  0
Row 33 ← lane  1
Row 34 ← lane  2
Row 35 ← lane  3
Row 36 ← lane  4
Row 37 ← lane  5
Row 38 ← lane  6
Row 39 ← lane  7
Row 40 ← lane  8
Row 41 ← lane  9
Row 42 ← lane 10
Row 43 ← lane 11
Row 44 ← lane 12
Row 45 ← lane 13
Row 46 ← lane 14
Row 47 ← lane 15
```

```
Row 48 ← lane 16
Row 49 ← lane 17
Row 50 ← lane 18
Row 51 ← lane 19
Row 52 ← lane 20
-- -- -- --
```



```
Row 48 ← lane 16
Row 49 ← lane 17
Row 50 ← lane 18
Row 51 ← lane 19
Row 52 ← lane 20
Row 53 ← lane 21
Row 54 ← lane 22
Row 55 ← lane 23
Row 56 ← lane 24
Row 57 ← lane 25
Row 58 ← lane 26
Row 59 ← lane 27
Row 60 ← lane 28
Row 61 ← lane 29
Row 62 ← lane 30
Row 63 ← lane 31
```

✓ 解释：为什么每个 lane 负责 2 行？

因为：

makefile

```
WM = 64 行
warp 只有 32 线程
→ 每线程必须覆盖 2 行
```

分配方式，由 TM_vec_id 决定：

css

```
lane in [0..31]:
    if (lane < 32):
        tm_vec_id = lane / 32 = 0      → rows 0~31
    if (lane < 32):
        tm_vec_id = 0 or 1 取决于 block 的分 ↓
```

✓ 解释：为什么每个 lane 负责 2 行？

因为：

```
makefile

WM = 64 行
warp 只有 32 线程
→ 每线程必须覆盖 2 行
```

分配方式，由 TM_vec_id 决定：

```
css

lane in [0..31]:
    if (lane < 32):
        tm_vec_id = lane / 32 = 0      → rows 0~31
    if (lane < 32):
        tm_vec_id = 0 or 1 取决于 block 的分块
```

实际更准确：

- `tm_vec_id = threadIdx.y` (如果你 warp 排布是 32x1)
- 或你自定义 grid 也可能：`tm_vec_id = lane / (warpSize/2)`

但因为你例子：

```
ini

TM = 8
VecSz = 4 → TM_vec_num = 2
WM = 64 → WM_per_loop = 32
```

所以 mapping 就是：

```
sql

每个 lane 负责一条 row，分别在上 32 行和下 32 行各一条
```

🔥 最终总结（口语版）



🔥 最终总结（口语版）

整个 warp: 32 个线程

需要覆盖 64 行 tile

所以：

- 前 32 行：lane 0~31
- 后 32 行：lane 0~31 再负责一次

这就是为什么你看到：

SCSS

Row X → lane ($X \% 32$)

如果你需要，我可以继续画：

- ✓ Warp 写入 global memory 的完整二维映射
- ✓ (thread, tm_loop, tn_loop, vecSz) → 真实坐标 (m,n) 的完整矩阵图
- ✓ 含 vector store 的合并写访存示意
- ✓ 每个 lane 写入的所有 C[m][n] 的完整列表

只要告诉我：“继续画二维图”或“给我每个 lane 的完整写入坐标表”。

□ △ ♀ ↗ ⚡ ...

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